colección DataPro

Inteligencia Artificial

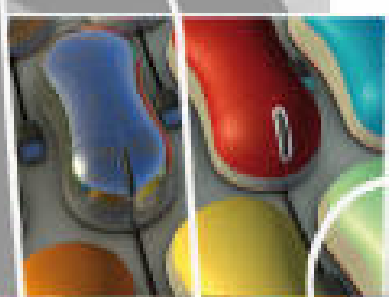
para desarrolladores

Conceptos e implementación en Java

Virginie MATHIVET

2ª edición

Archivos complementarios para descarga



eni

Inteligencia Artificial para desarrolladores

Conceptos e implementación en Java (2ª edición)

Este libro sobre **Inteligencia Artificial** está dirigido particularmente a los **desarrolladores** y no requiere profundos conocimientos en matemáticas. Al hilo de los distintos capítulos, la autora presenta las **principales técnicas de Inteligencia Artificial** y, para cada una de ellas, su inspiración biológica, física e incluso matemática, así como los distintos conceptos y principios (**sin entrar en detalles matemáticos**), con **ejemplos** y **gráficos** e **imágenes** para cada uno de ellos. Los dominios de aplicación se ilustran mediante **aplicaciones reales y actuales**. Cada capítulo contiene **un ejemplo de implementación genérico**, que se completa con una **aplicación práctica, desarrollada en Java**. Estos **ejemplos de código genéricos** son fácilmente adaptables a numerosas aplicaciones Java 10, sin plug-in externos. Las técnicas de Inteligencia Artificial descritas son:

- **Los sistemas expertos**, que permiten aplicar reglas para tomar decisiones o descubrir nuevos conocimientos.
- **La lógica difusa**, que permite controlar sistemas informáticos o mecánicos de manera mucho más flexible que con los programas tradicionales.
- **Los algoritmos de búsqueda de rutas**, entre los cuales el algoritmo A* se utiliza con frecuencia en videojuegos para encontrar los mejores caminos.
- **Los algoritmos genéticos**, que utilizan la potencia de la evolución para aportar soluciones a problemas complejos.
- **Los principales maetaheurísticos**, entre ellos la **búsqueda tabú**, que permiten encontrar soluciones óptimas a problemas de optimización, con o sin restricciones.
- **Los sistemas multi-agentes**, que simulan elementos muy simples que permiten conseguir comportamientos emergentes a partir de varios agentes muy sencillos.
- **Las redes neuronales (y el deep learning)**,, capaces de descubrir y reconocer modelos en series históricas, en imágenes o incluso en conjuntos de datos.

Para ayudar al lector a **pasar de la teoría a la práctica**, la autora proporciona para su descarga en esta página siete proyectos Java (realizados con Netbeans) uno por cada técnica de Inteligencia Artificial. Cada proyecto contiene un paquete genérico y uno o varios paquetes específicos a la aplicación propuesta.

El libro termina con una bibliografía que permite al lector encontrar más información acerca de las diferentes técnicas, una webgrafía que enumera algunos artículos que presentan aplicaciones reales, un anexo y un índice.

Virginie MATHIVET

Tras obtener el título de ingeniera en el INSA y un DEA en "Documentos, Imágenes y Sistemas de Información y Comunicaciones", **Virginie MATHIVET** ha realizado su tesis doctoral en el laboratorio LIRIS en Inteligencia Artificial, dedicada a los algoritmos genéticos y las redes neuronales. Después de enseñar inteligencia artificial, robótica y temas relacionados con el desarrollo durante más de 10 años, actualmente es directora de I+D en TeamWork. A través de este libro comparte su pasión por el dominio de la Inteligencia Artificial y la pone al nivel de todos los desarrolladores para que puedan explotar todo su potencial.

Objetivos de este libro

La inteligencia artificial, o I.A., es un dominio que apasiona a los aficionados a la ciencia ficción. No obstante, en el mundo actual, muchos desarrolladores no utilizan las técnicas asociadas a ellas, muchas veces por falta de conocimientos sobre ellas.

Este libro presenta las principales técnicas de inteligencia artificial, comenzando por los conceptos principales que deben comprenderse, y aportando algunos ejemplos de código escritos en Java a continuación.

Público objetivo y requisitos previos

Este libro está dirigido a todos aquellos lectores que deseen descubrir la inteligencia artificial. Cada capítulo detalla una técnica.

No son necesarios conocimientos previos en matemáticas, pues las fórmulas y ecuaciones se han limitado al mínimo imprescindible. En efecto, este libro se orienta, principalmente, a los conceptos y los principios subyacentes de las distintas técnicas.

La segunda parte de cada capítulo presenta ejemplos de código en Java. Es necesario tener un conocimiento previo mínimo del lenguaje. Este libro se dirige, por tanto, a los desarrolladores, en particular:

- A **estudiantes de escuelas técnicas** que deseen comprender mejor la inteligencia artificial y estudiar ejemplos de código.
- A **desarrolladores** que tengan que utilizar una tecnología particular y que quieran encontrar una explicación de los principios empleados, así como extractos de código reutilizables o adaptables.
- A **apasionados** que deseen descubrir la inteligencia artificial y codificar programas que hagan uso de ella.
- A todos aquellos **curiosos** interesados en este dominio.

Estructura del libro

Este libro empieza con una **introducción** que permite explicar qué es la inteligencia en general y la inteligencia artificial en particular. Se presentan sus principales dominios.

El libro incorpora, a continuación, siete capítulos. Cada uno de ellos aborda una técnica o un conjunto de técnicas. En su interior, encontrará en primer lugar una explicación de los principios y conceptos. A continuación, se incluyen ejemplos de la aplicación de estos algoritmos, así como un código comentado y explicado.

El lector curioso o que quiera aprender varias técnicas podrá leer el libro en el orden sugerido. No obstante, el lector que busque información acerca de una técnica particular podrá consultar directamente el capítulo correspondiente, pues son independientes.

El primer capítulo presenta los **sistemas expertos**, que permiten aplicar reglas para realizar un diagnóstico o ayudar a un profesional.

El segundo capítulo se centra en la **lógica difusa**, que permite tener controladores con un comportamiento más flexible y próximo al de los seres humanos.

El tercer capítulo aborda la **búsqueda de rutas**, en particular las rutas más cortas, en un mapa o en un grafo. Se presentan para ello varios algoritmos (búsqueda en profundidad, en anchura, Bellman-Ford, Dijkstra y A*).

El cuarto capítulo es relativo a los **algoritmos genéticos**. Estos algoritmos se inspiran en la evolución biológica para hacer evolucionar sus potenciales soluciones a los problemas, hasta encontrar soluciones adecuadas tras varias generaciones.

El quinto capítulo presenta varios **metaheurísticos de optimización**. Se presentan y comparan cinco algoritmos (algoritmo voraz, descenso por gradiente, búsqueda tabú, recocido simulado y optimización por enjambre de partículas) que permiten mejorar las soluciones obtenidas.

El sexto capítulo se centra en los **sistemas multiagentes**, en los que varios individuos artificiales con un comportamiento relativamente simple logran, de manera conjunta, resolver problemas complejos.

El último capítulo aborda las **redes neuronales**, que permiten aprender a resolver problemas cuya función subyacente no se conoce con precisión. Este capítulo también presenta la machine learning y el deep learning, dos concep-tos normalmente relacionados con las redes neuronales.

Al final del libro se incluyen:

- Una **webgrafía**, que presenta artículos relativos al uso real de estos algoritmos.
- Un **anexo** que permite instalar y utilizar SWI Prolog, que complementa a Java en el capítulo dedicado a los sistemas expertos.
- Un **índice**.

Código para descargar

El código de los distintos capítulos está disponible para su descarga desde la página Información. Hay disponible un proyecto realizado en NetBeans (versión 9.0) para cada capítulo.

Para abrir estos proyectos puede descargar de manera gratuita NetBeans. Está disponible en la siguiente dirección: https://netbeans.apache.org/download/

La versión de Java utilizada es la 11 (JDK 1.11). Los programas pueden, de este modo, ejecutarse en cualquier equipo que posea una JVM (máquina virtual de Java). Además, el código se ha separado en al menos dos paquetes: uno que contiene las clases que componen el núcleo de los algoritmos, genérico y reutilizable, y otro que contiene el programa principal y los ejemplos de uso.

La versión 11 de Java está disponible desde hace poco tiempo respecto al momento en el que es escribe este libro, sin embargo no es compatible con la versión 8.2 de NetBeans (ni Java 10). Aunque los códigos se escriban para Java 11 y NetBVeans 9, es posible reconstruir un proyecto Java 8 con NetBeans 8 sin modificar el código (retro-compatibilidad).

A excepción del capítulo Sistemas multiagentes, que incluye aplicaciones gráficas, los demás proyectos son de tipo «consola».

Las variables y métodos se han nombrado en español.

La visibilidad de las clases, métodos y atributos se ha limitado a lo estrictamente necesario para obtener un mejor nivel de seguridad del conjunto. No obstante, las reglas de diseño que exigen una estructuración en capas de tipo MVC no siempre se han respetado, de manera voluntaria. En efecto, esto habría agregado complejidad a los proyectos, y por tanto habría empeorado la legibilidad del código.

Del mismo modo, los distintos algoritmos presentados no se han optimizado cuando dichas optimizaciones iban en detrimento de la facilidad de lectura del código. En la medida de lo posible, los métodos más complejos se han dividido en métodos más pequeños, también con el objetivo de mejorar la legibilidad.

¡Les deseo una feliz lectura!

Presentación del capítulo

La **inteligencia artificial** consiste en volver inteligente un sistema artificial, principalmente informático. Esto supone que existe una definición precisa de la **inteligencia**, aunque no sea necesariamente el caso.

Esta introducción se interesa, en primer lugar, en la inteligencia humana y la manera en la que se define. A continuación, se explica cómo esta definición puede aplicarse a otras formas de vida, bien sean animales o vegetales, dado que, si la inteligencia no estuviera vinculada más que a la humanidad, sería inútil tratar de reproducirla en sistemas artificiales.

Una vez planteado el hecho de que la inteligencia puede encontrarse en cualquier ser vivo, veremos cómo definir la inteligencia artificial, así como las grandes corrientes de pensamiento que encontramos. Por último, esta introducción termina con un vistazo al panorama de sus dominios de aplicación.

Definir la inteligencia

Es importante comprender, en primer lugar, qué se entiende por **inteligencia**. Existen muchas ideas en torno a este asunto que pueden suponer un inconveniente para la comprensión (e incluso hacerla imposible) del campo de la inteligencia artificial.

El término inteligencia viene del latín "intelligentia", que significa la facultad de comprender y de establecer relaciones entre elementos.

La inteligencia es, no obstante, múltiple, y todos los autores actuales se ponen de acuerdo en el hecho de que no existe una sino varias inteligencias, y que cada uno de nosotros puede presentar fortalezas o debilidades en las distintas formas de inteligencia. La teoría de las inteligencias múltiples, propuesta inicialmente por Howard Gardner en 1983 (profesor en Harvard dedicado al estudio del fracaso escolar en niños), enumera siete formas de inteligencia, a las que se han agregado algunas otras hasta obtener la lista actual de las nueve formas de inteligencia:

- **Inteligencia lógica-matemática**: capacidad de trabajar con números, analizar situaciones y elaborar razonamientos. Se emplea en el ámbito científico, en particular en física y matemáticas.
- **Inteligencia visual-espacial**: capacidad para representar un objeto o un entorno en 3D; se utiliza para orientarse en un mapa, recordar una ruta o imaginar el resultado de una forma espacial a partir de su plano. La necesitan, por ejemplo, artistas, arquitectos o conductores de taxi.
- **Inteligencia verbal-lingüística**: capacidad para comprender y enunciar ideas a través del lenguaje. Requiere un buen conocimiento y dominio del vocabulario, así como de la sintaxis y los recursos estilísticos. Ayuda a abogados, políticos o escritores.
- **Inteligencia intrapersonal**: capacidad de formarse una imagen fiel de uno mismo, lo que significa ser capaz de determinar el propio estado emocional, las propias fortalezas y debilidades.
- **Inteligencia interpersonal**: capacidad para comprender a los demás y reaccionar de la manera adecuada. Está vinculada, por tanto, con las nociones de empatía, tolerancia y sociabilidad. También permite manipular a los demás, de modo que la utilizan, por ejemplo, los líderes de las principales sectas. Inspira, también, técnicas comerciales y de negociación.
- **Inteligencia corporal/cinestésica**: capacidad de formarse una representación mental del propio cuerpo en el espacio y ser capaz de llevar a cabo un movimiento particular. Muy utilizada por los atletas, es la que permite mantener un buen gesto en todo momento. Se utiliza en trabajos manuales y de precisión (por ejemplo, un cirujano), y permite también la expresión corporal de las emociones, de modo que es muy necesaria en el trabajo de bailarines y actores.
- **Inteligencia naturalista**: capacidad para ordenar, organizar y establecer jerarquías entre objetos de nuestro entorno. Permite, de este modo, definir especies, subespecies, o llevar a cabo clasificaciones. Se utiliza mucho, por ejemplo, en botánica, paleontología o biología.
- **Inteligencia musical**: es la capacidad de reconocer o crear melodías, notas musicales y harmonías. Es necesaria en compositores y cantantes, y se expresa en todos los melómanos.
- **Inteligencia existencial o espiritual**: es la capacidad de plantearse cuestiones acerca del sentido de la vida, sobre nuestro objetivo en el mundo. Se aproxima a nuestra noción de moralidad. No está, necesariamente, vinculada a la noción de religión, sino más bien a nuestra posición respecto al resto del universo.

Existen numerosos tests pensados para medir la inteligencia; el más conocido es el **test de C.I.** (cociente intelectual). Son, no obstante, muy criticados. En efecto, estas pruebas no son capaces de medir la inteligencia en toda su amplitud de formas y se centran, principalmente, en las inteligencias lógica-matemática y visual-espacial (incluso aunque se compruebe, en parte, la inteligencia verbal-lingüística). Todas las demás formas de inteligencia se ignoran.

Además, los principales tests de C.I. que encontramos están sesgados por la experiencia: basta con hacer y repetir varias veces los entrenamientos para estas pruebas para obtener resultados significativamente mejores. Por lo tanto, ¿se ha vuelto uno más inteligente? La repetición y el estudio apresurado no producen más que hábitos, reflejos y buenas prácticas para este tipo de problemas precisamente, pero este aprendizaje carece de valor alguno.

El sistema escolar en sí da prioridad a tres formas de inteligencia (lógica-matemática, visual-espacial y verbal-lingüística). Las demás formas de inteligencia se dejan a un lado y se estudian en materias llamadas "complementarias" (deporte, música, tecnología...), e incluso algunas no se abordan (inteligencias intra, interpersonal y existencial).

Debemos, por tanto, aceptar que la inteligencia no es fácilmente medible, ni fácilmente definible, pues cubre demasiados dominios. Además, ciertos tipos de inteligencia se encuentran subestimados o incluso ignorados.

La mejor definición es, también, la más vasta: **la inteligencia es la capacidad de adaptarse**. Permite, de este modo, resolver los problemas a los que nos enfrentamos. Esta es la definición que da, por ejemplo, Piaget (biólogo de formación y psicólogo) en 1963 en el libro "The Origin of Intelligence in Children".

La inteligencia de los seres vivos

La inteligencia se asocia, quizás demasiado, a la propia del ser humano. En efecto, el Hombre trata de mostrarse superior a los animales, y todo aquello que pueda diferenciarlo de ellos es conveniente para distinguirse de las "bestias". Este término es, por otro lado, muy significativo: designa todos los animales y personas que se consideran poco inteligentes.

Por lo tanto, la definición de inteligencia como capacidad para adaptarse permite tener en cuenta numerosos comportamientos que encontramos en la vida animal e incluso en otros seres vivos.

Cuando hablamos de la "**inteligencia de los seres vivos**", pensamos a menudo en los grandes primates (capaces de aprender el lenguaje de signos y de comunicarse gracias a él), en los perros o en los delfines. Podemos citar, también, el caso de Hans el listo, un célebre caballo que "sabía" contar y era capaz de responder golpeando con sus cascos en el suelo (por ejemplo, a la pregunta "¿cuánto suman 3 + 4?", golpearía siete veces). En realidad, era capaz de detectar pequeños movimientos muy sutiles en los rostros del público para determinar cuándo debía parar: había adaptado su comportamiento a su entorno para obtener golosinas y caricias.

Podemos hablar, también, de animales que demuestran una **inteligencia colectiva** destacable. Tenemos, por ejemplo, a las termitas, capaces de construir nidos inmensos y climatizados, compuestos por numerosos pasillos y celdas. Las hormigas son otro buen ejemplo, con la presencia de roles: reina, obreras, nodrizas, guardianas, combatientes e incluso incubadoras, dado que ciertas especies "crían" a las larvas y las protegen de los escarabajos para, a continuación, "traicionarlas" y consumir el néctar que han producido.

En el caso de las abejas, la **inteligencia lingüística** es muy destacable. En efecto, cuando una abeja vuelve a la colmena tras encontrar una fuente de polen, lo indica a las demás abejas mediante una danza. En función de la forma y la velocidad de esta, las demás abejas serán capaces de deducir la distancia. Observando el ángulo respecto al suelo, tendrán una indicación de la dirección. La propia danza es capaz de informar acerca de la fuente de alimento (el tipo de flor, por ejemplo).

Pero la inteligencia está, en realidad, presente en todas las formas de vida. Muchas especies vegetales se han adaptado para atraer a ciertas presas (como ocurre con las plantas carnívoras) o a insectos que se encargarán de diseminar su polen. Por el contrario, otras especies se protegen mediante jugos tóxicos o espinas.

Otras atraen a los depredadores naturales de sus propios depredadores. Por ejemplo, algunas plantas de la familia de las Acacias atraen a las hormigas para protegerse de los animales herbívoros. Aplican, de este modo, el famoso dicho "los enemigos de mis enemigos son mis amigos".

Podríamos ir más allá. Algunas setas han desarrollado estrategias muy complejas para su supervivencia y reproducción, y ciertas especies presentes en las selvas amazónicas de Brasil son capaces de utilizar las hormigas como huésped (al ser ingeridas, por ejemplo, mediante esporas) y tomar control temporalmente (mediante secreciones que atacan las funciones cerebrales y conducen a la hormiga afectada a salir del nido, trepar lo más alto posible y, finalmente, unirse a una hoja), y servir como alimento inicial y punto de partida para una nueva generación de esporas esparcidas.

Efectivamente, no todos los individuos de una misma especie tienen el mismo nivel de inteligencia, pero es imposible negar la posibilidad de encontrar formas de inteligencia en todas las especies vivas.

La inteligencia artificial

La naturaleza presenta numerosos casos de inteligencia: esta no es específica al ser humano. De hecho, no es ni siquiera propia de los seres vivos: cualquier sistema capaz de adaptarse y ofrecer una respuesta adecuada a su entorno podría considerarse como inteligente. Hablamos, en este caso, de **inteligencia artificial** (I.A.). Este término lo acuñó John McCarthy en 1956 (la I.A. tiene una historia larga e interesante).

El dominio de la inteligencia artificial es muy vasto y permite cubrir numerosas técnicas diferentes. La capacidad de cálculo cada vez mayor de los ordenadores, una mejor comprensión de ciertos procesos naturales vinculados a la inteligencia y el progreso de los investigadores en las ciencias fundamentales han permitido realizar grandes avances.

Por otro lado, no todas las facultades que podemos atribuir a un ordenador se pueden considerar como parte de la inteligencia artificial. De este modo, un ordenador capaz de resolver ecuaciones complejas en muy poco tiempo (mucho más rápido de lo que podría realizar un ser humano) no se considera, no obstante, como inteligente.

Como ocurre con los seres humanos (o con los animales), existen ciertas pruebas que permiten determinar si se puede considerar que el programa es, o no, inteligente. El más conocido es el **test de Turing** (descrito en 1950 por Alan Turing), que consiste en realizar una comunicación entre un ser humano encargado de realizar la prueba con dos pantallas. Detrás de una pantalla tenemos a otro ser humano encargado de escribir. Detrás de la otra pantalla tenemos el programa que queremos comprobar. Se solicita, pasada una fase en la que el encargado de realizar la prueba discute con ambos sistemas, determinar cuál era el ser humano. Si no se es capaz de diferenciar a la máquina del ser humano, entonces se ha superado la prueba.

Esta prueba ha sufrido, como ocurre con los tests de C.I., numerosas críticas. En primer lugar, no se aplica a todas las formas de inteligencia y no permite comprobar todos los programas (solo aquellos destinados a la comunicación). Además, un programa "no inteligente" que no hace más que responder ciertas frases, sin comprenderlas, es capaz de superar esta prueba, como fue el caso del programa ELIZA creado en 1966.

Este programa reconocía estructuras de frases para extraer las palabras más importantes y reutilizarlas en las siguientes preguntas. Por ejemplo, a la frase "Me gusta el chocolate" el programa respondía "¿Por qué dice que le gusta el chocolate?". Encontramos este programa como psicólogo en el editor de texto Emacs.

Por último, un programa demasiado inteligente capaz de responder a todas las preguntas de manera correcta, sin cometer ninguna falta de ortografía, de gramática o, simplemente, de tecleo sería reconocido rápidamente y suspendería el test.

Resulta, por tanto, difícil definir exactamente la inteligencia artificial: es preciso, sobre todo, que **la máquina dé la impresión de ser inteligente** cuando resuelve un problema, imitando por ejemplo el comportamiento humano o implementando estrategias más flexibles que las propias permitidas por la programación clásica. Incluso en este caso, encontramos cierta noción de **adaptabilidad**.

Es importante, en cambio, destacar que no existe ninguna noción de tecnología, de lenguaje o de dominio de la aplicación en esta definición. Se trata de un campo demasiado vasto, que podemos separar no obstante en dos grandes corrientes:

- **El enfoque simbólico**: el entorno se describe con la mayor precisión posible, así como las leyes aplicadas, y el programa es capaz de seleccionar la mejor opción. Este enfoque se utiliza en sistemas expertos o en la lógica difusa, por ejemplo.
- **El enfoque conexionista**: se le da al sistema un medio para evaluar si lo que hace está bien o no, y se le deja encontrar soluciones por sí solo, por emergencia. Este enfoque es propio de las redes neuronales o los sistemas multiagentes.

Estos dos enfoques no son del todo contradictorios y pueden resultar complementarios para resolver ciertos

problemas: podemos, por ejemplo, partir de una base simbólica que puede complementarse con un enfoque conexionista.

Dominios de aplicación

La inteligencia artificial se asocia, a menudo, con la **ciencia ficción**. La encontramos, por tanto, en muchas películas y libros, como por ejemplo el ordenador HAL 9000 de *Una odisea en el espacio*, de Stanley Kubrick (1968). Desgraciadamente (para nosotros, humanos), estos sistemas de I.A. tienen la mala costumbre de revelarse, o querer someter a los hombres, en ocasiones "por su propio bien", como ocurre en el film *I, Robot*, de Alex Proyas (2004).

En la actualidad, la inteligencia artificial se utiliza, efectivamente, en el mundo de la **robótica** para permitir a los robos interactuar de manera más flexible con los seres humanos a los que deben ayudar. Las tareas que deben realizar son, en ocasiones, muy sencillas, como limpiar el suelo, o mucho más complejas como ocurre con los "robots de compañía" que deben ayudar en la vida cotidiana a personas que no tienen todas sus facultades (por ejemplo, personas mayores o con alguna minusvalía). Existen muchos trabajos en este dominio, y las posibilidades son casi infinitas.

Los **militares** lo han comprendido bien: muchos robots se encargan o subvencionan mediante sus fondos destinados a la investigación. Hablamos de drones inteligentes, capaces de encontrar enemigos en zonas de combate, de soldados mecánicos, armas más inteligentes y también de robots que permiten encontrar y salvar a las víctimas de catástrofes naturales.

Otro gran dominio de la inteligencia artificial es el mundo de los **videojuegos**. En efecto, para obtener un juego más realista, es necesario que los personajes (enemigos o aliados) tengan un comportamiento que parezca lo más coherente posible a los ojos de los jugadores. En un juego del tipo *Metal Gear*, un enemigo que se acercara a usted con un pequeño cuchillo en mano en una zona despejada no sería realista, mientras que aquel escondido por los rincones y que ataca por la espalda sí parece "vivo". Además, si bien los monstruos de *Súper Mario* tienen rutas predefinidas, este tipo de estrategia no puede aplicarse en aquellos juegos donde la inmersión sea algo importante.

Incluso en el videojuego *Pac-Man*, los distintos fantasmas están dotados, cada uno, de cierta inteligencia artificial para controlar sus movimientos: Blinky (el fantasma rojo) intenta alcanzar la casilla en la que se encuentra actualmente el jugador; Pinky (rosa) intenta desplazarse cuatro casillas por delante (para pillarlo) mientras que Clyde (naranja) alterna entre intentar atrapar al personaje y alejarse (esto lo hace menos peligroso). Por último, Inky (azul) intenta bloquear al jugador de una manera similar a Pinky pero utilizando además la posición de Blinky, para "hacerle un sándwich".

Si bien la robótica y los videojuegos son dominios evidentes para la aplicación de técnicas de inteligencia artificial, no son, no obstante, más que la punta del iceberg. Existen muchos otros dominios que utilizan la I.A., desde el ámbito **bancario** hasta la **medicina**, pasando por la informática industrial.

En efecto, los sistemas expertos que permiten tomar una decisión basándose en reglas más o menos avanzadas se utilizan para detectar fraudes (por ejemplo, fraude con una tarjeta bancaria) o para detectar modificaciones en el comportamiento (así es como le proponen contratos de telefonía o de energía mejor adaptados cuando cambia su estilo de vida). Se utilizan, también, a menudo en medicina para ayudar en el diagnóstico, en función de los síntomas del paciente, de manera más rápida y completa que un médico (incluso aunque él sea el encargado de tomar las decisiones).

La I.A. puede encontrarse en dominios de la vida cotidiana como la **informática personal**: Clippy, el ayudante de la suite Microsoft Office, es un buen ejemplo. El correo postal pasa, también, por máquinas dotadas de cierta inteligencia artificial. En efecto, las direcciones manuscritas se leen y se reconocen, y a continuación se traducen y marcan en los sobres con un código de barras escrito en tinta naranja fosforescente. La velocidad de lectura es impresionante, con la posibilidad de codificar hasta 50.000 cartas por hora, ¡unas 14 cartas por segundo!

La **informática industrial** utiliza también, en gran medida, la I.A., por ejemplo en el campo de la logística para optimizar los trayectos de los camiones de reparto o la carga de estos. El orden en los almacenes también puede mejorarse gracias a algoritmos de inteligencia artificial. Es posible situar las piezas más eficazmente cambiando su forma, y los circuitos impresos se optimizan para limitar la cantidad de "puentes" o de material conductor.

Para terminar, estos últimos años se observa una explosión del **IoT** (*Internet of Things*) y de los objetos conectados. Si algunas veces es suficiente con devolver los valores de los sensores y elaborar estadísticas, cada vez se utilizan más ciertos datos para ofrecer al usuario información más relevante, como consejos sobre salud o alertas en caso de próximos fallos en el hardware.

Resumen

La inteligencia es un concepto difícil de definir con precisión, porque puede adoptar muchas formas. Resulta difícil, también, medirla, y las pruebas de C.I. están sesgadas. Podría resumirse como la capacidad de adaptación al entorno para resolver los problemas que se le presentan.

El reino animal está, por tanto, dotado de inteligencia, si bien diferente en cada caso, pero presente. En términos generales, todos los seres vivos, por su adaptación al entorno y la creación de estrategias de supervivencia complejas muestran pruebas de inteligencia.

Esta puede "implantarse" en las máquinas. La inteligencia artificial permite dotar a un sistema de un mecanismo que le permite simular el comportamiento de un ser vivo, comprenderlo mejor o incluso adaptar su estrategia a cambios y modificaciones en su entorno. Incluso en este caso resulta difícil determinar con precisión si un sistema presenta alguna forma de inteligencia; las pruebas del tipo "test de Turing" presentan, como ocurre con los tests de C.I., limitaciones.

Las tecnologías, los lenguajes y los algoritmos son tan numerosos como los dominios de aplicación, y la I.A. no está reservada a la robótica o a los videojuegos. En efecto, podemos encontrarla en casi todos los dominios informatizados. Nos envuelve, incluso sin darnos cuenta, y mejora prácticamente cualquier tipo de sistema.

Se trata de un dominio en pleno desarrollo y las capacidades crecientes de los ordenadores han permitido implementar algoritmos hasta el momento imposibles. Sin duda alguna, la I.A. va a formar parte de nuestro futuro y, por lo tanto, es importante para cualquier desarrollador, y en general para cualquier informático, comprender los mecanismos subyacentes para poder aplicarlos.

Presentación del capítulo

Con frecuencia, nos gustaría que un ordenador fuera capaz de darnos cierta información que no conocemos a partir de hechos conocidos.

Los propios seres humanos no siempre son capaces de deducir nuevos hechos a partir de otros conocidos y necesitan la ayuda de un **experto**. Por ejemplo, en el caso de un fallo en el automóvil, la mayoría de las personas no son capaces de determinar el origen de la avería y se dirigen al mecánico (el experto). Este, gracias a su conocimiento, encontrará el problema (y lo reparará, por lo general).

Muchos puestos de trabajo los ocupan estos expertos. Los médicos, los consultores o los agentes inmobiliarios no son más que algunos ejemplos.

La inteligencia artificial puede ayudarnos creando un programa informático llamado **sistema experto** que jugará el rol de este profesional. En ciertos casos, como en medicina, esta herramienta podrá convertirse en una ayuda para el propio especialista, puesto que el campo de estudio es muy vasto. Es poco habitual, en efecto, que el experto humano pueda remplazarse por completo, y estará ahí, a menudo, para confirmar la conclusión del sistema, ahorrándole en cualquier caso un tiempo precioso. En otros casos, el sistema proporcionará un primer diagnóstico, que complementará una persona física en caso de que el error sea desconocido en la base de datos utilizada (como ocurre con los fallos raros, por ejemplo). En la mayoría de los casos, el sistema experto será suficiente.

Este capítulo presenta, por tanto, los sistemas expertos, comenzando por sus conceptos principales. Se utiliza un ejemplo como hilo conductor, para comprender bien cómo se comunican entre sí todas las partes de un sistema experto.

A continuación, se presentan los grandes dominios de aplicación. Luego, se aborda el caso en que el conocimiento del dominio presente cierto grado de incertidumbre, y se explican las modificaciones necesarias para gestionar estos casos.

Por último, la implementación de sistemas más o menos complejos, en Java y en Prolog, completa este capítulo, que termina con un pequeño resumen.

Ejemplo: un sistema experto en polígonos

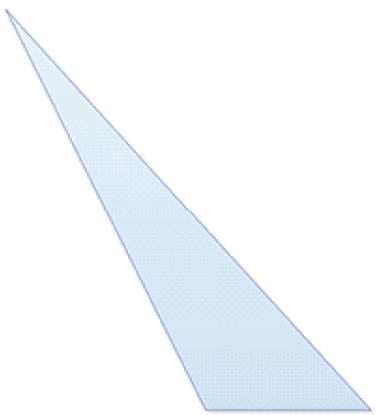
Esta sección permite ver el funcionamiento detallado de un sistema experto cuyo objetivo es determinar el nombre de un polígono (por ejemplo, un rectángulo) en función de características relativas a su forma (el número de lados, los ángulos rectos...). Este capítulo empieza con un breve repaso de geometría.

Un polígono se define como una forma geométrica que posee al menos tres lados. El **orden** de un polígono se corresponde con su número de lados.

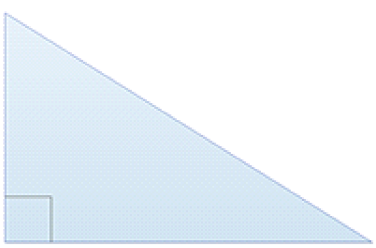
1. Triángulos

Si el orden de un polígono vale 3, entonces posee tres lados y se trata de un triángulo. Puede tratarse de un triángulo cualquiera: rectángulo, isósceles, rectángulo isósceles o equilátero.

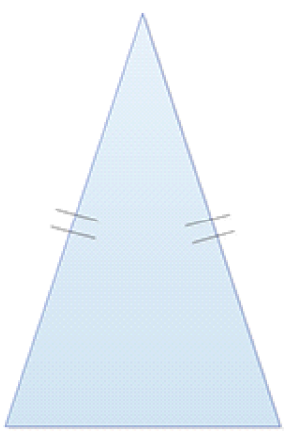
Las siguientes figuras nos recuerdan las particularidades de cada uno.



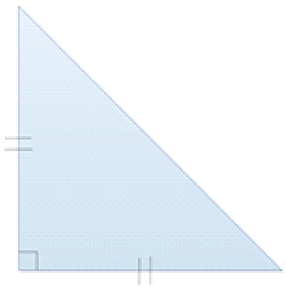
Triángulo cualquiera: posee tres lados de tamaños diferentes y ningún ángulo recto.



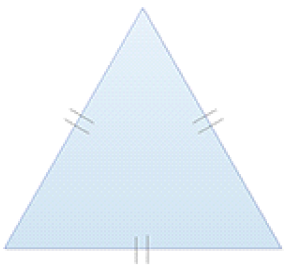
Triángulo rectángulo: posee tres lados de tamaños diferentes y un ángulo recto.



Triángulo isósceles: posee dos lados del mismo tamaño, pero ningún ángulo recto.



Triángulo rectángulo isósceles: conjuga los dos lados iguales de un triángulo isósceles y el ángulo recto del triángulo rectángulo.

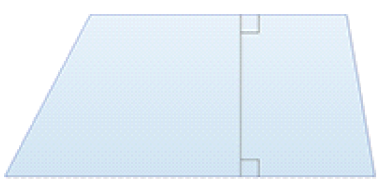


Triángulo equilátero: posee tres lados del mismo tamaño (no puede poseer ningún ángulo recto).

2. Cuadriláteros

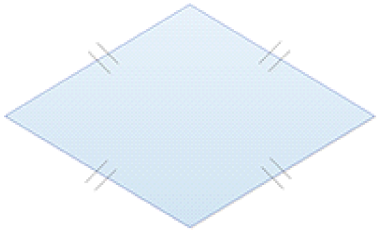
Cuando el orden de un polígono vale 4, hablamos de cuadriláteros. Pueden ser de cualquier tipo: un trapecio, un paralelogramo, un rombo, un rectángulo o un cuadrado. Las siguientes figuras presentan los distintos tipos de cuadrilátero y sus propiedades.

Cuadrilátero cualquiera: posee cuatro lados no paralelos.

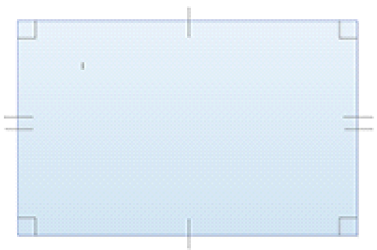


Trapecio: posee dos lados (y solo dos) paralelos.

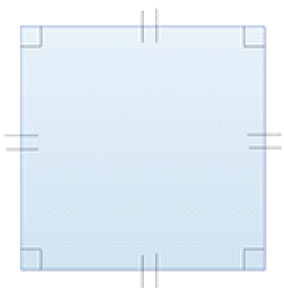
Paralelogramo: posee cuatro lados paralelos dos a dos. Posee, también, lados opuestos del mismo tamaño.



Rombo: paralelogramo cuyos lados son todos del mismo tamaño.



Rectángulo: paralelogramo con los cuatro ángulos rectos.



Cuadrado: paralelogramo que conjuga los cuatro lados del mismo tamaño del rombo con todos los ángulos rectos del rectángulo.

3. Otros polígonos

Cuando el orden es superior a 4, el nombre del polígono está determinado según la siguiente tabla, para los casos más corrientes. Hablamos de polígono regular cuando todos los lados son del mismo tamaño.

Orden	Nombre del polígono
5	Pentágono
6	Hexágono
8	Octógono
10	Decágono
12	Dodecágono
20	Icoságono

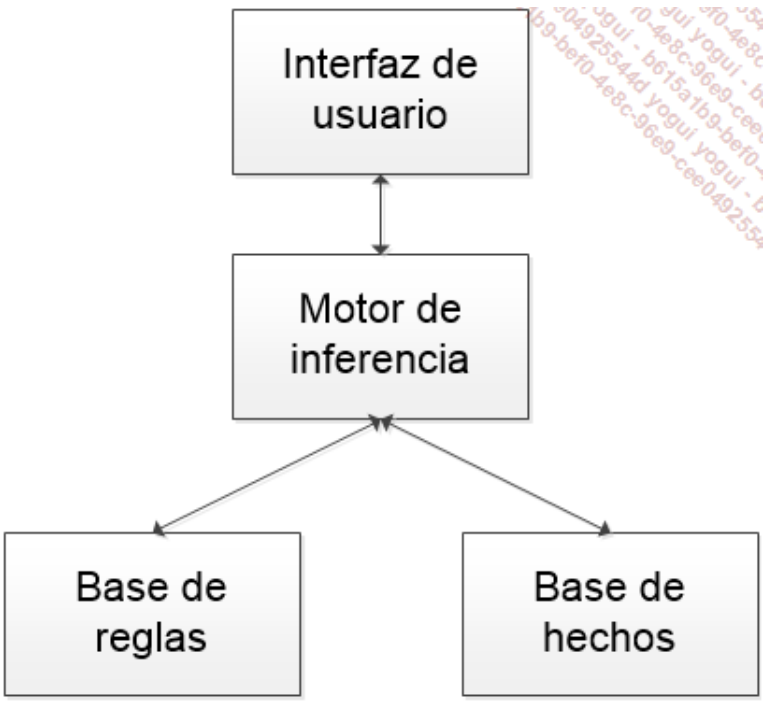
Todos los polígonos, sea cual sea su orden, poseen un nombre particular. De este modo un polígono de orden 26 se denomina "hexaicoságono". No obstante, la tabla anterior muestra únicamente los nombres más comunes.

Contenido de un sistema experto

Un sistema experto está formado por diferentes partes vinculadas entre sí:

- Una **base de reglas** que representa el conocimiento experto.
- Una **base de hechos** que representa el conocimiento actual del sistema acerca de un caso concreto.
- Un **motor de inferencia** que permite aplicar las reglas.
- Una **interfaz** de usuario.

El siguiente esquema indica los vínculos entre las distintas partes, que se detallan a continuación.



1. Base de reglas

Un sistema experto contiene un conjunto de reglas llamado **base de reglas**. Estas representan el conocimiento del experto en el dominio.

Estas reglas son siempre de la forma:

```
SI (conjunto de condiciones) ENTONCES nuevo conocimiento
```

Las condiciones de aplicación de una regla se denominan las **premisas**. Pueden existir varias premisas, en cuyo caso están vinculadas por una coordinación Y, lo que quiere decir que deben ser ciertas todas ellas para poder aplicar la regla.

Si la regla necesita una coordinación O, se dividirá en dos reglas distintas equivalentes. Así, la regla:

```
Si (A O B) entonces C
```

Se convertirá en:

```
Si A entonces C
Si B entonces C
```

Los nuevos conocimientos se denominan **conclusiones**.

Para nuestro sistema experto relativo a las formas geométricas, he aquí las reglas correspondientes a los triángulos:

```
SI (orden vale 3) ENTONCES es un triángulo

SI (triángulo Y 1 ángulo recto) ENTONCES es un triángulo rectángulo

SI (triángulo Y 2 lados del mismo tamaño) ENTONCES es un triángulo isósceles

SI (triángulo rectángulo Y triángulo isósceles) ENTONCES es un triángulo rectángulo isósceles

SI (triángulo Y lados todos iguales) ENTONCES es un triángulo equilátero
```

Existirán otras reglas para los cuadriláteros y los polígonos de orden superior. Vemos rápidamente que el número de reglas puede ser importante.

Además, según el sistema utilizado, cada regla debe seguir una sintaxis precisa e impuesta.

En particular, las premisas y las conclusiones pueden obtenerse bajo la forma *atributo(valor)*, por ejemplo orden(3) o anguloRecto(1). La regla para el triángulo rectángulo en dicha representación podría ser:

```
SI (orden(3) Y anguloRecto(1)) ENTONCES poligono(TrianguloRectangulo)
```

Cuando el sistema experto está formado por todas las piezas, es posible seleccionar el formato de las reglas de manera que resulte lo más próximo posible al dominio estudiado. Esto facilitará la implementación y la creación del motor.

2. Base de hechos

Las premisas de una regla pueden ser de dos tipos:

- Conocimiento relativo al problema que proporciona el usuario del sistema: se trata de las **entradas**.
- Conocimiento derivado de la aplicación de las reglas: se trata de los **hechos inferidos**.

Ambos tipos de conocimiento deben registrarse en una **base de hechos** que contiene, en este caso, toda la información acerca del problema, sea cual sea su origen. Cuando se ejecuta un sistema experto la base no contiene, inicialmente, más que el conocimiento del usuario (las entradas) y se completa poco a poco con hechos inferidos.

Supongamos que contamos con una forma de orden 4, que posee 4 lados paralelos, del mismo tamaño, y 4 ángulos rectos. Este será nuestro conocimiento inicial.

A estos hechos de entrada se les suma el hecho de que la figura es un cuadrilátero (es de orden 4) y un paralelogramo (un cuadrilátero con 4 lados paralelos), que se trata, con más precisión, de un rombo (un

paralelogramo cuyos 4 lados son del mismo tamaño) y que se trata, a su vez, de un rectángulo (un paralelogramo con los ángulos rectos). Por último, se agrega el hecho de que se trata de un cuadrado (puesto que se trata de un rombo y un rectángulo).

Por lo general, este último hecho es el que interesa realmente al usuario: se trata del **objetivo del programa**.

No obstante, también podemos diseñar un sistema experto que permita saber si un hecho es verdadero o no. En este caso, se consultará si el hecho concreto se encuentra en la base de hechos una vez aplicadas las reglas.

En el ejemplo anterior de nuestro cuadrilátero, en lugar de preguntar cuál es el nombre del polígono (se trata de un cuadrado), podríamos haber preguntado si se trata de un rombo o si se trata de un triángulo rectángulo. Se obtendría una respuesta afirmativa en el primer caso (un cuadrado es un rombo particular) y negativa en el segundo caso (un cuadrado no es un triángulo).

3. Motor de inferencia

El **motor de inferencia** (o **sistema de inferencia**) es el núcleo del sistema experto.

El motor va a permitir seleccionar y aplicar las reglas. Esta tarea no es, necesariamente, sencilla, puesto que pueden existir miles de reglas. Además, no servirá de nada aplicar una regla que ya se ha utilizado antes o que no se corresponde con el problema que tratamos de resolver.

Por ejemplo, si creamos un sistema experto que permite reconocer la fauna y la flora de un bosque y queremos saber a qué familia pertenece un insecto, no servirá de nada intentar aplicar las reglas correspondientes a los árboles o arbustos.

Es también el motor de inferencia el que agregará los nuevos hechos a la base de hechos, donde accederá para verificar si un hecho ya es conocido. Agregar hechos que se sabe que son falsos resulta tan interesante como agregar hechos verdaderos. En efecto, saber que una forma no es un cuadrilátero permite eliminar varias reglas de golpe. En un sistema complejo, saber que el insecto buscado no posee alas es, también, bastante revelador y permite limitar los intentos para encontrar la especie buscada.

Una analogía sencilla es el juego "¿Quién es quién?". En este juego, debe encontrarse un personaje entre varios planteando preguntas cuyas respuestas pueden ser sí o no (por ejemplo, "¿lleva sombrero?"). Que la respuesta sea positiva o no aporta, en ambos casos, conocimiento. En efecto, saber que no posee sombrero permite eliminar aquellos personajes que sí lleven, igual que el hecho de saber que lleva gafas permite mantener únicamente aquellos que sí lleven.

Por último, el motor debe poder tomar una decisión importante: detenerse para presentar al usuario la respuesta a su pregunta. Debe saber, por tanto, cuándo se ha alcanzado un objetivo o cuándo no se alcanzará jamás.

La ventaja de los sistemas expertos sobre otras muchas técnicas de inteligencia artificial es su capacidad para explicar el razonamiento seguido: los motores implementan, a menudo, un mecanismo que permite encontrar todas las reglas utilizadas para alcanzar un hecho concreto. El usuario obtiene el razonamiento además del resultado, lo cual puede resultar importante en ciertos dominios.

Existen muchos motores de inferencia disponibles, y es posible codificarlos por partes, en cuyo caso no importa el lenguaje de programación. Se han creado, sin embargo, algunos lenguajes con el objetivo de implementar motores de inferencia. Se corresponden con la familia de los lenguajes de **programación lógica**, a los que pertenece Prolog. No obstante, también es posible crear un motor de inferencia en un lenguaje orientado a objetos como Java. Por último, existen ciertos puentes entre estos lenguajes que permiten utilizar un motor de inferencia Prolog dentro de un código implementado en otro lenguaje, por ejemplo.

4. Interfaz de usuario

El último elemento de un sistema experto es la **interfaz de usuario**. En efecto, es necesario que el usuario pueda, sencillamente, introducir los datos que posee, bien antes de comenzar el proceso o bien conforme los necesite el motor de inferencia.

Como con cualquier otra aplicación, si la interfaz no es agradable para el usuario o si resulta demasiado compleja, incluso poco intuitiva, el sistema será poco o nada utilizado.

Además, en un sistema experto, es primordial que las opciones disponibles para el usuario sean claras, para que pueda responder correctamente a las preguntas planteadas; en caso contrario el resultado devuelto sería falso.

En el caso del sistema experto correspondiente a los polígonos, existe poco margen de error, pues las reglas están basadas en el número de lados, de ángulos rectos, de lados paralelos y en el tamaño de los lados. Por lo tanto, una interfaz que solicite el orden del polígono sin precisar que se trata del número de lados resultaría poco adecuada.

En el caso del sistema de reconocimiento de insectos, si la aplicación pregunta si tiene fórceps o cornículos, existen pocas posibilidades de que un usuario no entomólogo sepa responder correctamente. Por el contrario, si se pregunta si el cuerpo termina con dos puntas largas (fórceps, como encontramos por ejemplo en las tijeretas) o dos pequeñas puntas (cornículos, presentes en los pulgones), habrá menos margen de error o de confusión. Una foto o un dibujo resultarían incluso más explícitos para el usuario.

Es importante, por lo tanto, trabajar el aspecto ergonómico del sistema experto con los futuros usuarios o representantes de los usuarios, con el objetivo de saber qué términos estarán mejor adaptados, así como la disposición de las pantallas, para limitar los errores.

Tipos de inferencia

Los motores de inferencia pueden encadenar las reglas de diversas maneras: es lo que se denomina el **método de razonamiento**. Los dos principales métodos de razonamiento son el razonamiento deductivo y el razonamiento inductivo, aunque existen motores que poseen un método de razonamiento mixto.

1. Razonamiento deductivo

a. Principio

Un motor de **razonamiento deductivo** también se denomina motor de inferencia **dirigido por los datos**.

En este tipo de motor se parte de los datos disponibles basándose en los hechos, y se comprueba para cada regla si se puede aplicar o no. Si se puede, se aplica y se agrega la conclusión a la base de hechos.

El motor explora, entonces, todas las posibilidades, hasta encontrar el hecho buscado o hasta que no puede aplicar nuevas reglas.

Este modo de razonamiento se propone por defecto en lenguajes del tipo CLIPS (*C Language Integrated Production System*), especializados en la construcción de sistemas expertos.

b. Aplicación a un ejemplo

En el caso de nuestro sistema experto relativo a los polígonos, supongamos que partimos de los hechos siguientes:

- El orden vale 3.
- Existe un ángulo recto.
- Dos lados son del mismo tamaño.

Empezaremos aplicando la siguiente regla, que agrega a la base de hechos que la forma es un triángulo:

```
SI (orden vale 3) ENTONCES es un triángulo
```

A continuación podemos deducir que se trata de un triángulo rectángulo gracias a la regla siguiente:

```
SI (triángulo Y 1 ángulo recto) ENTONCES es un triángulo rectángulo
```

Además, sabemos que es un triángulo isósceles:

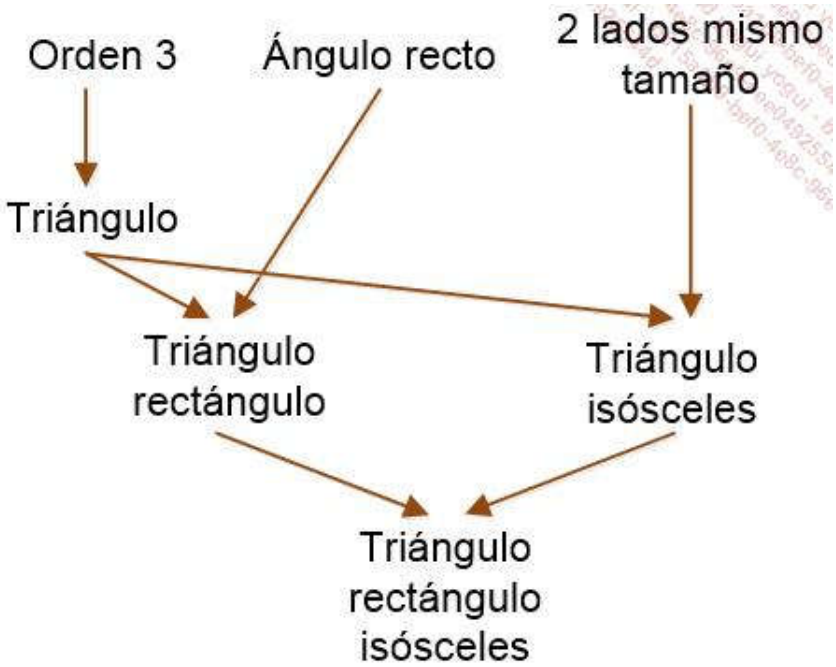
```
SI (triángulo Y 2 lados del mismo tamaño) ENTONCES es un triángulo isósceles
```

Por último, sabiendo que se trata de un triángulo rectángulo y de un triángulo isósceles, podemos aplicar:

```
SI (triángulo rectángulo Y triángulo isósceles) ENTONCES es un triángulo rectángulo isósceles
```

Se agrega, por tanto, el hecho de que se trata de un triángulo rectángulo isósceles. Como no existen más reglas que podamos aplicar, el motor de inferencia se detiene. Se informa al usuario de que su polígono es un triángulo rectángulo isósceles.

Podríamos resumir la lógica del motor con el siguiente esquema:



2. Razonamiento inductivo

a. Principio

Un motor de inferencia de **razonamiento inductivo** también se denomina **dirigido por objetivos**.

En este caso, se parte de los hechos que se desea obtener y se busca alguna regla que permita obtener dicho hecho. Se agregan todas las premisas de esta regla en los nuevos objetivos que hay que alcanzar.

Se vuelve a iterar, hasta que los nuevos objetivos que hay que alcanzar estén presentes en la base de hechos. Si un hecho está ausente en la base de hechos o se ha probado como falso, entonces se sabe que la regla no se puede aplicar. Estos motores poseen, por tanto, un mecanismo (el **backtracking**) que les permite pasar a una nueva regla, que será un nuevo medio de probar el hecho.

Si no existe ninguna regla que nos permita alcanzar el objetivo, entonces se considera como falso.

Este modo de razonamiento está presente por defecto en el lenguaje Prolog, dedicado a los sistemas expertos.

b. Aplicación a un ejemplo

Retomamos el ejemplo anterior, correspondiente a un polígono para el que:

- El orden vale 3.
- Existe un ángulo recto.
- Dos lados son del mismo tamaño.

Se pregunta a la aplicación si el triángulo es rectángulo isósceles. Es nuestro primer objetivo. El motor busca alguna regla que le permita obtener este hecho; existe solo una:

```
SI (triángulo rectángulo Y triángulo isósceles) ENTONCES es
un triángulo rectángulo isósceles
```

El motor agrega, ahora, los objetivos "triángulo rectángulo" y "triángulo isósceles" a su lista de objetivos. Empieza buscando alguna regla que le permita comprobar si se trata de un triángulo rectángulo. En este caso, tenemos una única regla:

```
SI (triángulo Y un ángulo recto) ENTONCES es un triángulo rectángulo
```

Se obtienen, ahora, dos nuevos objetivos: ¿se trata de un triángulo y tiene un ángulo recto? La presencia de un ángulo recto está en la base de hechos, de modo que este objetivo se ha alcanzado. Para saber si es un triángulo, seguiremos la siguiente regla:

```
SI (orden vale 3) ENTONCES es un triángulo
```

La base de hechos precisa que el orden vale 3, de modo que la regla se comprueba. El hecho triángulo puede agregarse a la base de hechos y permite alcanzar los objetivos correspondientes, en nuestro caso "triángulo rectángulo". Queda por comprobar "triángulo isósceles".

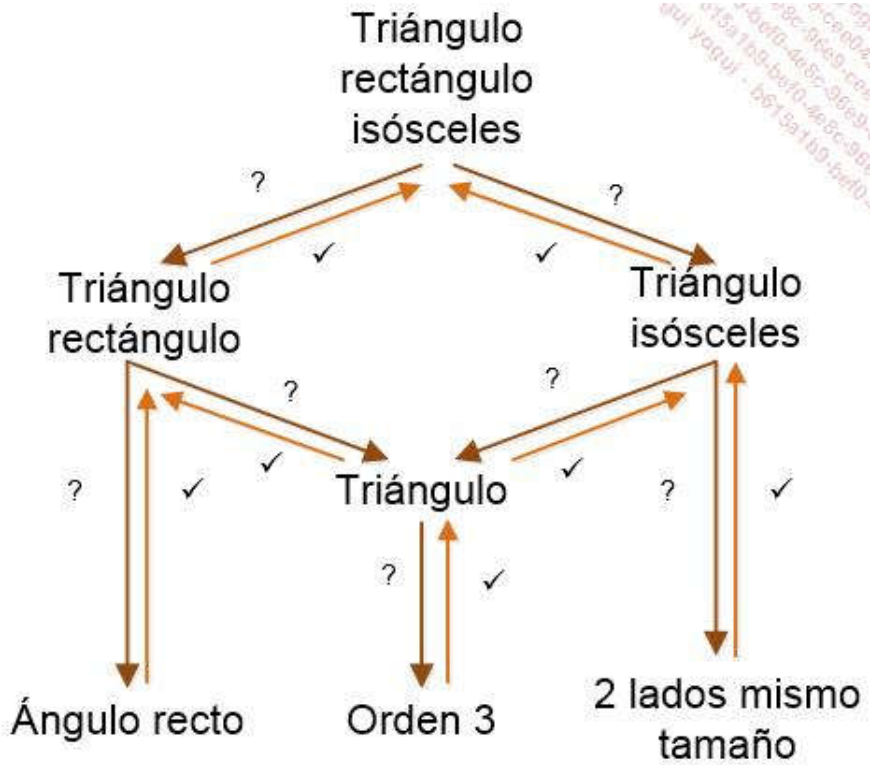
De la misma manera, se busca alguna regla que posea este objetivo:

```
SI (triángulo Y 2 lados del mismo tamaño) ENTONCES es un triángulo isósceles
```

El hecho de que la forma sea un triángulo está en la base de hechos (se ha obtenido justo antes) y la presencia de dos lados del mismo tamaño, también. Se agrega el hecho de que es un triángulo isósceles.

El programa termina volviendo a su objetivo inicial, a saber si se trata de un triángulo rectángulo isósceles. Como los hechos "triángulo rectángulo" y "triángulo isósceles" se han comprobado, podemos concluir que sí, que la forma es un triángulo rectángulo isósceles, e informárselo al usuario.

La lógica es la siguiente: se parte del objetivo que se ha de alcanzar y se intenta comprobar si las premisas son verdaderas.



3. Razonamiento mixto

Cada método de razonamiento tiene sus ventajas e inconvenientes:

- El razonamiento deductivo permite descubrir permanentemente nuevos hechos, pero corre el riesgo de aplicar y comprobar muchas reglas que no se corresponden con la información buscada por el usuario. Está mejor adaptado para la exploración.
- El razonamiento inductivo permite concentrarse en el objetivo preciso (o en varios objetivos), pero va a comprobar numerosas posibilidades que finalmente se demostrarán falsas. De este modo, intentará comprobar reglas que no se cumplirán. Su gestión es, también, más compleja (puesto que es necesario gestionar la lista de objetivos y permitir el backtracking).

Se ha propuesto una mezcla de ambos razonamientos: el **razonamiento mixto**. En este nuevo método de razonamiento se alternarán periodos de razonamiento deductivo (para deducir nuevos hechos basándose en los que acabamos de comprobar) y periodos de razonamiento inductivo (en los que buscaremos nuevos objetivos para comprobar).

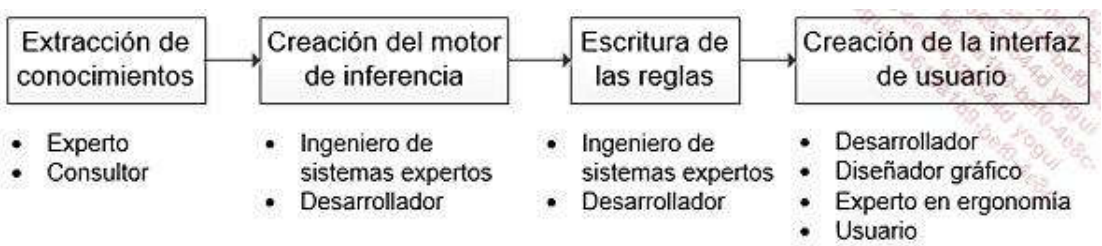
Se trata de un sabio equilibrio entre ambos métodos de búsqueda, en función de las reglas de búsqueda. Además, podemos alternar las fases de búsqueda en profundidad con fases de búsqueda en extensión según los objetivos. Esto supera, no obstante, el ámbito de este libro.

Aunque el razonamiento mixto se utiliza muy poco, puesto que añade complejidad al motor de inferencia, es mucho más eficaz.

Etapas de construcción de un sistema

Para crear íntegramente un sistema experto, es muy importante seguir distintas etapas que requerirán competencias y, por tanto, perfiles profesionales diferentes.

En términos generales, existen cuatro etapas que se presentan en la siguiente figura, junto a los principales roles necesarios en cada etapa:



1. Extracción del conocimiento

La primera etapa consiste en **extraer el conocimiento**. Para ello, es necesario encontrar a un experto al que se le preguntará para comprender las reglas subyacentes en las que se basa su trabajo. Esta fase puede parecer sencilla, pero es de hecho muy compleja. En efecto, un experto no reflexiona basándose en reglas, sino que se basa en ciertos automatismos que tendrá que llegar a explicitar.

Tomemos, como ejemplo, los insectos. Si nos centramos en insectos poco comunes o desconocidos, parece bastante sencillo determinar las reglas que permiten llegar al resultado deseado. Pero ¿qué reglas podemos aplicar para diferenciar una mosca de un mosquito, un caracol de una babosa, una hormiga de una cochinilla, o incluso una mariquita de un zapatero?

Planteando diversas cuestiones al experto podremos ayudarle a determinar las reglas que aplica, a menudo de manera inconsciente. El trabajo del interrogador es, de este modo, primordial, y está encargado de indicar las zonas de sombra, donde existen reglas que no son lo suficientemente específicas para poder discriminar entre dos resultados.

Esta etapa puede ser muy extensa, en particular cuando se abordan dominios muy vastos. Además, si el dominio de aplicación posee riesgos, es importante verificar las reglas por parte de varios expertos, que podrán completarlas o modificarlas si es necesario.

El éxito del sistema depende en gran medida de esta fase. No debe infravalorarse queriendo ir demasiado rápido.

2. Creación del motor de inferencia

Si el proyecto utiliza un motor de inferencia existente, esta fase consistirá en adquirirlo y configurarlo.

En caso contrario, será preciso **implementar un motor de inferencia**, con las distintas funcionalidades deseadas. En este momento se tendrá que crear la estructura de la base de hechos, definiendo las interacciones entre el motor y las bases (de reglas y de hechos).

El formalismo de las reglas quedará fijado, lo que podrá tener un impacto importante en las siguientes fases.

3. Escritura de las reglas

La siguiente fase consiste en **transformar las distintas reglas** obtenidas tras la extracción del conocimiento a un formato conveniente para el motor de inferencia.

Al final de esta etapa, la base de reglas debe ser completa. No debemos dudar en verificar varias veces la presencia de todas las reglas y su exactitud, pues un error a este nivel puede echar a perder todo el trabajo realizado con el experto o los expertos.

En esta etapa será necesario contar con un especialista del lenguaje del motor y del sistema experto.

4. Creación de la interfaz de usuario

La última etapa consiste en **crear la interfaz de usuario**. Hemos visto antes hasta qué punto debe trabajarse para permitir un uso sencillo y preciso del motor de inferencia y de las reglas.

En una primera versión, podemos imaginar una interfaz basada en entradas/salidas por una consola. Sin embargo, para una aplicación orientada al gran público, será conveniente disponer de una interfaz gráfica. Es importante contar con la intervención de la mayoría de los usuarios, o al menos con sus representantes, y especialistas en ergonomía y diseño para definir las distintas pantallas.

Existe, no obstante, un caso particular: si el sistema experto lo utiliza otro sistema informático (y no un ser humano), en su lugar será preciso crear los canales de comunicación entre los distintos programas (mediante API, archivos, flujos, sockets...).

Una vez creado el sistema experto, es posible utilizarlo.

Rendimiento y mejoras

Hay un aspecto que no se ha tenido en cuenta hasta ahora: el rendimiento. Se trata de un aspecto primordial para el éxito del sistema. Por ello, vamos a ver cuáles son los criterios de rendimiento y cómo construir un sistema que permita mejorarlo.

1. Criterios de rendimiento

El rendimiento de un sistema experto, sobre todo si está compuesto por muchas reglas, es primordial. El primer criterio de rendimiento es el **tiempo de respuesta**. En efecto, debe ser capaz de dar una respuesta al usuario en un espacio de tiempo aceptable.

Este tiempo depende del problema planteado. Por ejemplo, en nuestro sistema experto de geometría, el tiempo de respuesta será aceptable si se mantiene en el orden de un segundo. Visto el número de reglas, hay pocos riesgos de tener un tiempo superior.

En un sistema experto médico, o para ayudar a un mecánico, el tiempo no es la prioridad, siempre y cuando se mantenga en unos pocos segundos.

Sin embargo, si el sistema experto debe utilizarse en un entorno peligroso para tomar una decisión (por ejemplo, para detener o no una máquina) o debe comunicarse con otros sistemas, el tiempo de respuesta se convertirá en un criterio esencial para el éxito del proyecto.

Además del tiempo de respuesta, existe un segundo criterio de rendimiento: el **uso de la memoria**. En efecto, la base de hechos va a aumentar conforme se apliquen las reglas. En un motor de razonamiento inductivo, el número de objetivos que hay que alcanzar puede ocupar cada vez más espacio. Si el sistema debe instalarse en un dispositivo que posea poca memoria (como un robot), será necesario tener en cuenta este aspecto.

Por último, generalmente todos los medios implementados para optimizar el tiempo de respuesta tendrán un impacto negativo en la memoria, y viceversa. Es preciso, por tanto, encontrar un compromiso en función de las necesidades.

2. Mejorar el rendimiento mediante la escritura de reglas

La primera forma de mejorar el rendimiento es trabajar en la escritura de reglas. En efecto, a menudo es posible limitar su número.

En nuestro ejemplo con los triángulos, hemos definido el triángulo rectángulo isósceles como un triángulo rectángulo que es, a su vez, un triángulo isósceles, pero habríamos podido decir que un triángulo rectángulo e isósceles es un triángulo que posee un ángulo recto y dos lados del mismo tamaño. Es inútil implementar ambas reglas que, si bien diferentes, son redundantes.

Conviene saber también qué hechos se informarán o no por parte del usuario. De este modo, habríamos podido definir nuestros cuadriláteros no en función de sus lados paralelos y sus ángulos rectos, sino por sus propiedades relativas a las diagonales (por ejemplo, que se cruzan en el medio para un paralelogramo, que son del mismo tamaño para un rectángulo, o incluso que se cruzan en ángulo recto para un rombo). Sin embargo, si el usuario no dispone de esta información, estas reglas serán inútiles, mientras que un motor de razonamiento inductivo intentará probar estos objetivos intermedios.

El orden de las reglas es también importante. De hecho, la mayoría de los motores escogen la primera regla que se corresponde con lo que están buscando, es decir, la primera regla que cumple todas las premisas en un razonamiento deductivo o la primera que tiene como conclusión el objetivo en curso en un razonamiento inductivo.

Es interesante, por tanto, implementar las reglas de modo que se apliquen primero aquellas que tengan más probabilidades de cumplirse (o bien las más fáciles de comprobar o refutar).

Algunos motores utilizan criterios suplementarios para escoger las reglas, como el número de premisas o la "frescura" de los hechos utilizados (para utilizar la mayor cantidad de hechos recientes obtenidos). Es primordial, por lo tanto, conocer bien la manera en la que trabaja el motor de inferencia para optimizar la base de reglas.

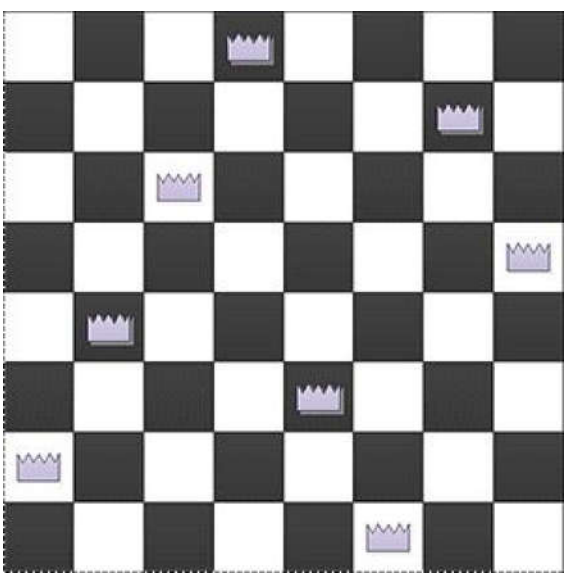
La última gran técnica para optimizar esta base es agregándole índices. Se basan en el mismo principio que los índices en las bases de datos: permiten encontrar más rápidamente las reglas utilizando un hecho determinado, bien como premisa (para motores de razonamiento deductivo) o como conclusión (en el caso de razonamiento inductivo).

3. Importancia de la representación del problema

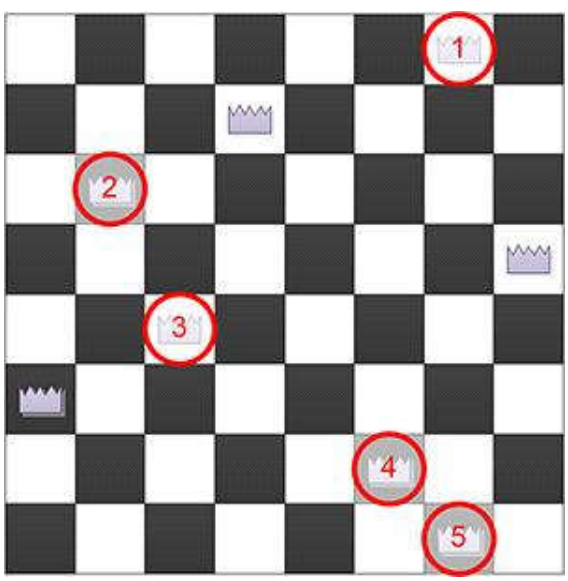
Un motor de inferencia, antes de llegar al resultado esperado, realizará muchos intentos. Es importante limitarlos para optimizar el rendimiento del sistema experto.

Para ello, nos interesaremos en un problema clásico: el problema de las "8 reinas". El objetivo es situar sobre un tablero de ajedrez (es decir, una rejilla de 8 * 8 casillas), ocho reinas, que no deben amenazarse entre sí, sin importar el color de las casillas. Dos reinas se amenazarán entre sí si están en la misma línea o la misma columna, o en la misma diagonal.

Las siguientes dos figuras muestran un caso de éxito y un caso incorrecto donde varias reinas se amenazan entre sí:



En esta posición, no existe ninguna amenaza entre las distintas reinas. Se trata, por lo tanto, de una posible solución.



En esta posición, por el contrario, 5 reinas se amenazan entre sí. Las reinas 1 y 3 están situadas en la misma diagonal ascendente, mientras que las reinas 1 y 5 están en la misma columna, 2, 4 y 5 están en la misma diagonal descendente.

En total, existen 92 posiciones posibles que responden al problema de las 8 reinas.

Si definimos nuestro problema para que nos dé las posiciones posibles de las 8 reinas con la forma (x,y), veremos que existen 64 posiciones posibles por reina (visto que hay 64 casillas). ¡Esto nos lleva a tener que comprobar 648 posibilidades (es decir, más de 280.000 billones)! Un algoritmo así tardará mucho tiempo en ejecutarse, incluso

aunque se comprueben varios miles de posiciones por segundo.

Podemos observar el hecho de que cada reina debe estar situada sobre una casilla diferente de las anteriores. De este modo, para la primera reina, tenemos 64 posibilidades. Cuando buscamos la posición para la segunda reina, ya tenemos una casilla ocupada, de modo que quedan solamente 63 posibilidades. La tercera reina podrá situarse en las 62 casillas restantes, y así sucesivamente. En lugar de 64^8 posiciones para probar, tenemos "solamente" 64 * 63 * 62 * 61 * 60 * 59 * 58 * 57. Este cálculo vale más de 170.000 billones, pero ya representa una mejora apreciable (en torno a un 37 %).

En matemáticas, hablamos de variaciones. Siguen la notación A_{64}^{8} y su valor es $\frac{64!}{(64-8)!}$

Podríamos plantear una reflexión para comprender que las reinas deben estar situadas en columnas diferentes (en caso contrario se amenazarán entre sí, como ocurre con las reinas 1 y 5 en la figura anterior). Buscaremos, por tanto, para cada reina sobre qué fila (entre las 8 posibles) está situada. Queda por comprobar "solamente" 88 posibilidades, algo menos de 17 millones de posibilidades: aquí, la primera reina dispone de 8 casillas posibles en la primera columna; la segunda, 8 casillas en la segunda columna...

Por último, teniendo en cuenta el hecho de que las filas deben ser también diferentes, vemos que si la primera reina se sitúa en la fila 1, la segunda reina no dispone más que de 7 filas posibles para situarse. Obtenemos, entonces, 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 posibilidades, que se escribe 8! (factorial de 8) y vale 40.320. En matemáticas, hablamos de permutaciones.

Cambiando simplemente la representación del problema, gracias a las restricciones conocidas, hemos podido pasar de más de 200.000 billones de posibilidades a aproximadamente 40.000. Si el tiempo necesario para comprobar todas las posibilidades en el primer caso hacía que el problema fuera imposible de resolver, vemos que sería realizable en un tiempo aceptable en el último caso.

La elección en la representación del problema no es, en absoluto, anodina, y es importante reflexionar bien antes de embarcarse en la creación de las reglas. El rendimiento puede verse fuertemente impactado.

Más adelante en este capítulo se propone una implementación para el sistema experto sobre los polígonos y para las 8 reinas en Prolog. Será esta última versión, que utiliza permutaciones, la que se usará para resolver este problema.

Agregar incertidumbre y probabilidades

Los sistemas expertos vistos hasta el momento se basaban en reglas seguras, y los hechos eran necesariamente verdaderos o falsos. Sin embargo, en la vida real, las cosas son a menudo más complejas. Conviene pensar, por lo tanto, en gestionar la incertidumbre.

1. Incorporar incertidumbre

En un sistema experto destinado a identificar los animales en función de sus características físicas, puede resultar complicado estimar exactamente el número de dedos en las patas del animal o el color de su vientre. Sobre todo si se trata de un depredador o de un animal venenoso, puede resultar difícil examinarlo bajo todos los ángulos para responder a las preguntas del sistema.

En este caso, puede resultar interesante agregar incertidumbre sobre los hechos: el usuario podrá decir que le parece que el animal tiene el vientre blanco, pero que no está del todo seguro.

Además, en un sistema médico experto, parecería peligroso decir que si los síntomas de una enfermedad son dolor en todo el cuerpo, fiebre y un gran cansancio, entonces se trata necesariamente de una gripe. En efecto, existen enfermedades mucho más raras y más peligrosas que podrían esconderse detrás de estos síntomas.

Esta vez, son las propias reglas las que resultan inciertas: hay una gran posibilidad de que se trate de una gripe, pero no es la única explicación posible.

Estos dos tipos de incertidumbre (sobre los hechos y sobre las reglas) pueden gestionarse en un sistema experto para hacerlo más eficaz.

2. Hechos inciertos

Para los hechos, es posible agregarles una probabilidad. Indica hasta qué punto el usuario está seguro de sí mismo.

De este modo, un hecho con una seguridad del 80 % indica que el usuario tiene una pequeña duda acerca del hecho. A la inversa, un hecho seguro al 100 % indica que es absolutamente cierto.

Es bastante fácil agregar estas probabilidades. Sin embargo, durante la aplicación de las reglas, habrá que cambiar el funcionamiento del motor de inferencias. Este calculará en primer lugar la probabilidad de las premisas de las reglas. Se tratará del valor mínimo de los distintos hechos.

De este modo, si tenemos una regla del tipo "Si A y B entonces C", y A es verdadero al 75 % y B al 85 %, consideraremos que el conjunto de las premisas es verdadero al 75 %.

Una regla cuya probabilidad sea inferior al 50 % no se aplicará, generalmente.

El hecho inferido tomará también como valor de certidumbre el correspondiente a la regla. En nuestro caso, agregaremos el hecho C con un valor del 75 % a la base de hechos.

De este modo, existirá una propagación de las distintas probabilidades.

3. Reglas inciertas

Como con los hechos, es posible agregar probabilidades a las reglas. Podríamos decir también que un diagnóstico es verdadero al 75 %, es decir, 3 veces de cada 4.

Un hecho inferido a partir de esta regla sería verdadero al 75 %, los hechos inferidos reciben como probabilidad la de la regla que los ha creado.

Evidentemente, es posible acumular las probabilidades sobre los hechos y las reglas. La probabilidad de un hecho inferido es la probabilidad de las premisas multiplicada por la probabilidad de la regla. De este modo, premisas verdaderas al 80 % en una regla verdadera al 75 % producen un hecho que será verdadero al 75*80/100 = 60 %.

Si un mismo hecho inferido se obtiene de distintas maneras (por ejemplo, aplicando varias reglas), hay que combinar las probabilidades obtenidas. El cálculo resulta, aquí, más complejo, puesto que debe tener en cuenta las probabilidades ya obtenidas.

En efecto, si una primera regla nos indica que el hecho es verdadero al 80 % y otro que el hecho lo es al 50 %, no podemos concluir simplemente que es verdadero al 80% tomando el valor máximo. Diremos que de entre los 20 % que no son seguros de entre la primera regla, la segunda satisface el 50 %, es decir un 10 % del total. El hecho inferido tendrá, entonces, una probabilidad del 90 % (80+10).

La fórmula que permite calcular esta probabilidad total en función de un hecho verdadero con probabilidad P_a y una nueva regla que lo produce con una probabilidad P_b es:

$$P_{total} = P_a + (1 - P_a) * P_b$$

Podemos destacar que el orden de aplicación de las reglas no es importante; el resultado obtenido es siempre el mismo.

Si fuera necesario, es posible integrar estas probabilidades a todos los niveles, para mejorar los sistemas expertos producidos.

Dominios de aplicación

El primer sistema experto apareció en 1965. Llamado Dendral, permitía encontrar componentes de un determinado material a partir de información acerca de su resonancia magnética y el espectrograma de masa de este.

Después, los sistemas expertos se han desarrollado en numerosos dominios y están presentes hasta en algunos dispositivos de uso cotidiano.

1. Ayuda al diagnóstico

El primer gran dominio de aplicación de los sistemas expertos es la **ayuda al diagnóstico**. En efecto, gracias a su base de reglas, permiten comprobar y descartar distintas posibilidades hasta encontrar una o varias probables.

Los encontramos, de este modo, en aplicaciones médicas para asistir al diagnóstico de ciertas enfermedades. Podemos citar MYCIN, que permite determinar qué bacteria se encuentra en el cuerpo de un paciente y qué tratamiento (tipo de antibiótico y posología) administrarle para que mejore, o CADUCEUS, una extensión de MYCIN, que permite diagnosticar más de 1000 enfermedades de la sangre. Existen otras aplicaciones que permiten asistir al diagnóstico de enfermedades a partir de radiografías o de imágenes médicas. Más recientemente, las encontramos en otras aplicaciones médicas: determinación de la edad de los huesos a partir de un radio (Seok, 2016) o incluso clasificación de las medidas de glucemia en el marco de una diabetes gestacional (Caballero-Ruiz, 2016).

Resultan muy prácticas para ayudar en la búsqueda de fallos mecánicos (como ocurre en los coches) o en electrónica (para reparar electrodomésticos), acotando las piezas potencialmente defectuosas.

Los encontramos también en nuestros ordenadores cuando se detecta algún fallo o error y el sistema nos plantea diversas preguntas antes de ofrecernos un procedimiento para intentar resolver el problema (el asistente de Microsoft Windows es un buen ejemplo).

Por último, serán cada vez más utilizados en los distintos objetos conectados que encontraremos; por ejemplo, en las pulseras inteligentes que nos permiten seguir nuestro estado de salud en tiempo real.

2. Evaluación de riesgos

El segundo gran dominio es la **evaluación de riesgos**. En efecto, partiendo de las diversas reglas del experto, la aplicación permitirá evaluar los riesgos para hacerles frente o intentar evitarlos.

Existen también sistemas expertos capaces de determinar los riesgos sobre las construcciones, para poder reaccionar rápidamente (reforzando la estructura). Los tres más importantes son Mistral y Damsafe para las presas y Kaleidos para los monumentos históricos.

En el dominio médico, existen también sistemas expertos que permiten determinar las poblaciones en riesgo, por ejemplo para determinar la probabilidad de un nacimiento prematuro durante un embarazo.

En el dominio financiero, permiten determinar los riesgos para los créditos o la seguridad vinculada a un préstamo. Se utilizan, por último, en la detección de fraude monitorizando las transacciones que parecen anormales (por ejemplo, para determinar si se ha hecho un uso indebido de una tarjeta de crédito robada).

La estimación de riesgos no se limita a estos dominios, y puede encontrarse por todas partes. Así, "Rice-crop doctor" es un sistema experto utilizado en agricultura que permite indicar riesgos de enfermedades en el arroz.

3. Planificación y logística

Los sistemas expertos que permiten comprobar varias posibilidades siguiendo reglas están muy bien adaptados a la creación de **planificaciones** que deban respetar distintas restricciones impuestas. Este problema, sobre todo cuando es de medio o gran tamaño, es imposible de resolver por un solo ser humano en un tiempo razonable.

Los encontramos también en todos los dominios que requieren cierta planificación, bien en las escuelas para organizar los horarios, en los aeropuertos para organizar los vuelos o en los hospitales para planificar el uso de las salas de operaciones.

En logística, periten también optimizar la organización de los almacenes o las rutas de reparto.

4. Transferencia de competencias y conocimiento

Los sistemas expertos manipulan conocimiento, y es interesante utilizarlos para **transferir competencias**.

Es posible usarlos, de este modo, en educación y en formación: permiten indicar al aprendiz las etapas que le van a ayudar a determinar un hecho o las reglas que se aplican en un dominio. Permiten guiarle en su razonamiento y enseñarle ciertos "reflejos", adquiridos por los expertos gracias a años de experiencia.

La identificación de objetos, de plantas, de animales o de piedras es más fácil gracias a un sistema experto: formulando preguntas, será capaz de indicar lo que intentamos identificar. La importancia está, en este caso, en el orden de las preguntas. Esta sucesión de preguntas se denomina "clave de determinación". Existen muchas, para todos los dominios, siempre basadas en un conjunto de reglas.

Por último, no solo pueden utilizar el conocimiento de un experto, sino que pueden también manipular este conocimiento para deducir nuevos hechos, desconocidos hasta el momento. La demostración automática consiste, por tanto, en dotar al sistema de hechos (matemáticos, por ejemplo) y de reglas que indiquen cómo combinarlos. Se deja, a continuación, que el sistema encuentre nuevas demostraciones para teoremas conocidos, o que busque nuevos teoremas.

5. Otras aplicaciones

Las aplicaciones de los sistemas expertos no se limitan a los grandes dominios anteriores. También es posible encontrarlos en dominios mucho más específicos.

Las herramientas actuales de procesamiento de textos poseen, todas ellas, correctores ortográficos y gramaticales. Si bien la ortografía se basa únicamente en el reconocimiento o no de las palabras escritas, la gramática es mucho más compleja y requiere un sistema experto que conozca las diversas reglas (por ejemplo, la concordancia entre el nombre y el adjetivo). Los hechos son las palabras utilizadas, con su género y su número, así como las estructuras de las frases. Estos sistemas son cada vez más precisos.

Siempre en el ámbito del lenguaje, se utilizan cada vez más los sistemas expertos en la comprensión de textos escritos o hablados, para pre-tratar las frases antes que cualquier otra etapa. Es el caso de Watson (IBM), que ganó el Jeopardy en 2011 gracias a su comprensión del inglés escrito. En primer lugar trata una primera frase gracias a un sistema experto, antes de utilizar otros algoritmos. Watson utiliza Prolog para su sistema experto y continua siendo mejorado.

Podemos encontrar también sistemas expertos como ayuda a la configuración de máquinas o de sistemas en función de hechos relativos a su uso, al sistema subyacente o al servicio para el que se instala. Otros permiten controlar en tiempo real sistemas, como ocurre en la NASA con ciertas operaciones realizadas sobre las naves espaciales.

Encontramos sistemas expertos en aplicaciones de creación de piezas mecánicas. Estos sistemas permiten respetar

las reglas de diseño, difíciles de tener en cuenta por parte de un operador humano, debido a su complejidad o a la gran cantidad de ellas.

Por último, hace algunos años que podemos utilizar Prolog para diseñar sitios web sin tener que pasar por el conjunto más habitual "LAMP" (Linux - Apache - MySQL - PHP). En efecto, hay diferentes frameworks para simplificar la escritura de sitios completos y dinámicos, basados en reglas y de manera más concisa que un lenguaje imperativo (como PHP). También podemos citar ClioPatria, Prosper o SWI-Prolog que ofrecen diferentes librerías.

Los sistemas expertos pueden, por tanto, encontrarse en todos los dominios donde se requiera actualmente un experto humano, lo que supone una gran capacidad de adaptación y numerosas aplicaciones.

Creación de un sistema experto en Java

Codificar un motor de inferencia genérico y completo en Java es una tarea compleja de realizar. Además, existen motores disponibles (gratuitos o no) que fácilmente pueden adaptarse a nuestro problema.

Vamos a interesarnos, sin embargo, en la creación del sistema experto que permite determinar el nombre de un polígono partiendo de sus características.

Este sistema experto será, no obstante, fácil de adaptar a otros problemas similares. Además, el código Java será portable sobre todos los equipos que posean una máquina virtual (JVM). Las salidas se harán por consola.

1. Definición de requisitos

Este sistema debe ser capaz de adaptarse a muchos problemas equivalentes, de tipo identificación, a partir de información introducida por el usuario.

Aquí, el sistema parte de los datos proporcionados por el usuario para intentar determinar la forma que desea reconocer. No tenemos un objetivo preciso. Conviene, por tanto, utilizar un motor de razonamiento deductivo para este problema, que además es más sencillo de implementar.

Además, partiendo de nuestras reglas, tendremos dos tipos de hechos:

- Hechos cuyo valor es un número entero, como el orden del polígono o el número de lados del mismo tamaño.
- Hechos cuyo valor es un valor booleano, como la presencia o no de un ángulo recto, o el hecho de ser un triángulo.

Nuestro sistema experto deberá, por lo tanto, tener en cuenta estos dos tipos de hechos.

El usuario debe, también, poder utilizar fácilmente el sistema experto. Las reglas tendrán que escribirse en un lenguaje cercano al lenguaje natural, y la interfaz de usuario tendrá que gestionarse por separado del motor para hacerla más genérica (aquí realizaremos únicamente una interfaz en modo consola).

Por último, también será interesante conocer el último nombre encontrado para la forma y los nombres intermedios que se han utilizado (por ejemplo, se ha utilizado "Triángulo", a continuación "Triángulo rectángulo" y "Triángulo isósceles" y por último se ha terminado con un "Triángulo rectángulo isósceles"). Trataremos de implementar, entonces, una manera de conocer el orden de los hechos.

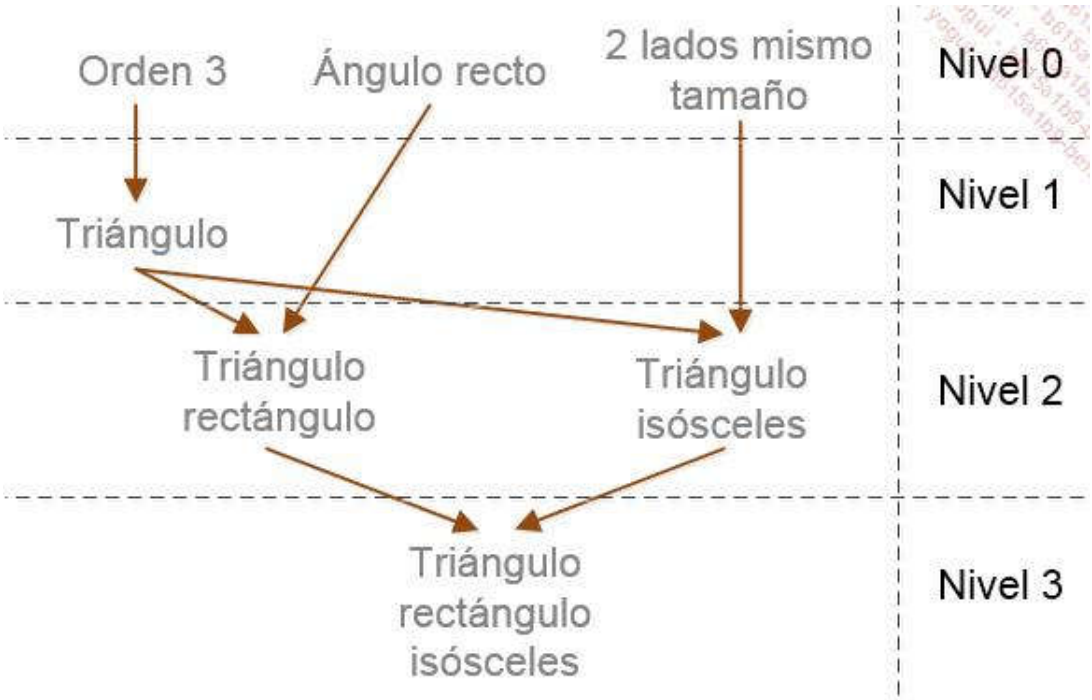
2. Implementación de los hechos

Los hechos son los primeros que se codifican. Como tenemos dos tipos de hechos, se codifica una interfaz genérica, que se implementa a continuación en dos tipos particulares.

Un hecho puede poseer:

- Un nombre, que es una cadena de caracteres.
- Un valor, que es de tipo `Object` a nivel de la interfaz (en las clases concretas, se tratará de un valor entero o un valor booleano, pero podríamos imaginar otros tipos de hechos).
- Un nivel, que se corresponde con el lugar dentro del árbol de decisiones: el nivel será 0 para los hechos definidos por el usuario, y se aumentará en 1 el nivel para los hechos inferidos. Este nivel será modificable por el motor.
- Una pregunta que se planteará al usuario, para los hechos que pueden solicitarse (los hechos que solo puedan inferirse no tendrán pregunta).

En el ejemplo del triángulo rectángulo isósceles, la siguiente figura muestra los niveles asociados a cada conclusión obtenida. Este nivel se incrementa en 1 respecto a los hechos utilizados como premisas. Esto nos permite saber cuál es el hecho de más alto nivel (que probablemente será el más importante para el usuario).



Se crea una interfaz **IHecho** que define varios métodos que permiten leer los atributos, y un método para modificar el nivel del hecho:

```
public interface IHecho {
    String Nombre();
    Object Valor();
    int Nivel();
    String Pregunta();

    void setNivel(int l); // Permite modificar el nivel de un hecho
}
```

Dos clases implementarán esta interfaz: `HechoEntero`, que será un hecho con valor entero, y `HechoBooleano`, un hecho con valor booleano. Estas clases poseerán atributos protegidos, accesibles mediante los métodos de la interfaz.

Para el hecho entero, se añade un constructor que inicializa los diferentes valores. El método `toString` para la representación creará una cadena del tipo "Orden=3 (0)", lo que significa que el hecho "Orden" vale 3, y que es un hecho definido por el usuario (nivel 0).

He aquí el código de la clase **HechoEntero**:

```
class HechoEntero implements IHecho {

    protected String nombre;
```

```
    @Override
    public String Nombre() {
        return nombre;
    }

    protected int valor;
    @Override
    public Object Valor() {
        return valor;
    }

    protected int nivel;
    @Override
    public int Nivel() {
        return nivel;
    }

    public void setNivel(int l) {
        nivel = l;
    }

    protected String pregunta = "";
    @Override
    public String Pregunta() {
        return pregunta;
    }


    public HechoEntero(String _nombre, int _valor, String _pregunta,
int _nivel) {
        nombre = _nombre;
        valor = _valor;
        pregunta = _pregunta;
        nivel = _nivel;
    }

    @Override
    public String toString() {
        return nombre + "=" + valor +" (" + nivel + ")";
    }
}
```

Ocurre de manera similar para la clase **HechoBooleano** que representa un hecho booleano. Para representarlo, si el hecho es verdadero lo representaremos únicamente con la forma Hecho(nivel) indicando que el hecho se ha obtenido en el nivel definido, y !Hecho(nivel) si el hecho es falso.

He aquí su código:

```
class HechoBooleano implements IHecho {

    protected String nombre;
    @Override
    public String Nombre() {
```

```
        return nombre;
    }

    protected boolean valor;
    @Override
    public Object Valor() {
        return valor;
    }

    protected int nivel;
    @Override
    public int Nivel() {
        return nivel;
    }

    @Override
    public void setNivel(int n) {
        nivel = n;
    }

    protected String pregunta = null;
    @Override
    public String Pregunta() {
        return pregunta;
    }

    public HechoBooleano(String _nombre, boolean _valor, String
_pregunta, int _nivel) {
        nombre = _nombre;
        valor = _valor;
        pregunta = _pregunta;
        nivel = _nivel;
    }

    @Override
    public String toString()
    {
        String res = "";
        if (!valor)
        {
           res += "!";
        }
        res += nombre + " (" + nivel + ")";
        return res;
    }
}
```

3. Base de hechos

Una vez definidos los hechos es posible implementar la base de hechos, llamada `BaseDeHechos`. Esta, vacía al principio, se completará poco a poco por el motor. Contiene una lista de hechos, se inicializa en el constructor y está accesible en lectura mediante un accesor:

```
import java.util.ArrayList;

class BaseDeHechos {
    protected ArrayList<IHecho> hechos;
    public ArrayList<IHecho> getHechos()
    {
        return hechos;
    }

    public BaseDeHechos()
    {
         hechos = new ArrayList<IHecho>();
    }
}
```

Se implementan otros dos métodos: uno que permite agregar un hecho a la base y otro para vaciarla completamente (para poder tratar un nuevo caso, por ejemplo):

```
   public void Vaciar() {
        hechos.clear();
   }

   public void AgregarHecho(IHecho hecho) {
        hechos.add(hecho);
   }
```

Además, hacen falta otros dos métodos más específicos:

- Un método `Buscar` que permite buscar un hecho en la base. Recibe como parámetro un nombre y devuelve el hecho si lo encuentra (o `null` en caso contrario).
- Un método `RecuperaValorHecho` que permite encontrar el valor de un hecho en la base, siempre a partir de su nombre, que se pasa como parámetro. Si el hecho no existe, este método devuelve `null`. Para ser genérico, el valor devuelto es de tipo `Object`.

He aquí el código:

```
    public IHecho Buscar(String nombre) {
        for(IHecho hecho : hechos) {
            if (hecho.Nombre().equals(nombre)) {
                return hecho;
            }
        }
        return null;
    }

    public Object RecuperarValorHecho(String nombre) {
        for(IHecho hecho : hechos) {
            if (hecho.Nombre().equals(nombre)) {
                return hecho.Valor();
            }
```

```
        }
        return null;
    }
```

La base de hechos está, ahora, completa.

4. Reglas y base de reglas

Tras los hechos, es posible codificar las reglas. Estas contienen tres atributos, con los getters/setters correspondientes:

- Un nombre, que será una cadena de caracteres.
- Una lista de hechos que forman las premisas de la regla (la parte izquierda).
- Un hecho que es la conclusión de la regla (la parte derecha).

El código base (que contiene los tres atributos y los getters/setters) de la clase **`Regla`** es el siguiente:

```
import java.util.ArrayList;
import java.util.StringJoiner;

public class Regla {
   protected ArrayList<IHecho> premisas;
   public ArrayList<IHecho> getPremisas() {
       return premisas;
   }

   public void setPremisas(ArrayList<IHecho> _premisas) {
       premisas = _premisas;
   }

   protected IHecho conclusion;
   public IHecho getConclusion() {
       return conclusion;
   }
   public void setConclusion(IHecho _conclusion) {
       conclusion = _conclusion;
   }

   protected String nombre;
   public String getNombre() {
       return nombre;
   }
   public void setNombre(String _nombre) {
       nombre = _nombre;
   }

   public Regla(String _nombre, ArrayList<IHecho> _premisas,
IHecho _conclusion) {
       nombre = _nombre;
       premisas = _premisas;
       conclusion =  conclusion;
```

```
    }
}
```

Por motivos de legibilidad, se agrega un método `toString`. Describe la regla de la siguiente manera:

```
Nombre: IF (premisa1 AND premisa2 AND ...) THEN conclusión
```

El código de este método utiliza la clase `StringJoiner` de Java, que permite concatenar varias cadenas con un separador (AND en nuestro caso).

```
    @Override
    public String toString() {
        String cadena = nombre + " : IF (";

        StringJoiner sj = new StringJoiner(" AND ");
        for(IHecho hecho : premisas) {
            sj.add(hecho.toString());
        }

        cadena += sj.toString() + ") THEN " + conclusion.toString();
        return cadena;
    }
```

La base de reglas se denomina **`BaseDeReglas`**. Contiene una lista de reglas, accesible en lectura o en escritura a través de métodos, y un constructor que inicializa la lista. Preste atención: cuando se crea una nueva base a partir de una lista de reglas, hay que tener la precaución de copiarlas una a una, de cara a evitar que la eliminación de una regla en la base provoque la eliminación del contenido de la base original.

```
import java.util.ArrayList;

class BaseDeReglas {
    protected ArrayList<Regla> reglas;
    public ArrayList<Regla> getReglas() {
        return reglas;
    }
    public void setReglas(ArrayList<Regla> _reglas) {
        // Se copian las reglas y se agregan
        for (Regla r : _reglas) {
            Regla copia = new Regla(r.nombre, r.premisas, r.conclusion);
            reglas.add(copia);
        }
    }

    public BaseDeReglas() {
        reglas = new ArrayList();
    }
}
```

Existen varios métodos para agregar una regla, eliminarla o vaciar toda la base de reglas:

```
    public void ClearBase()
    {
        reglas.clear();
    }

    public void AgregarRegla(Regla r)
    {
        reglas.add(r);
    }

    public void Eliminar(Regla r)
    {
        reglas.remove(r);
    }
```

Se han creado las dos bases (hechos y reglas), así como las estructuras contenidas en ellas (los hechos y las reglas).

5. Interfaz

El último elemento relacionado con nuestro motor en un sistema experto es la interfaz de usuario (o IHM). Es necesario definir en primer lugar una interfaz que indique los métodos que tendrán que implementar todas las IHM.

Necesitamos dos métodos que permitan solicitar al usuario información acerca de un hecho, entero (`PedirValorEntero`) o booleano (`PedirValorBooleano`). Además, se necesitan dos métodos para mostrar los hechos (`MostrarHechos`) y las reglas (`MostrarReglas`).

He aquí el código de la interfaz **`IHM`**:

```
import java.util.ArrayList;

public interface IHM {
    int PedirValorEntero(String pregunta);
    boolean PedirValorBooleano(String pregunta);
    void MostrarHechos(ArrayList<IHecho> hechos);
    void MostrarReglas(ArrayList<Regla> reglas);
}
```

Estos métodos son, voluntariamente, muy genéricos: en función del programa deseado, las entradas podrán realizarse por línea de comandos, mediante un formulario web, en un campo de texto, mediante un cursor... Del mismo modo, la representación de las reglas y de los hechos es libre (texto, tabla, lista, gráfico...).

En nuestro caso, el programa principal **`Aplicacion`** implementa estos métodos utilizando únicamente la consola para las entradas/salidas. Para la lectura de los valores, se realiza una conversión para los valores enteros y una lectura del tipo sí/no para los valores booleanos (no se realiza una gestión de errores).

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.util.ArrayList;
import java.util.Collections;

public class Aplicacion implements IHM {
    public static void main(String[] args) {
        // A completar más tarde
    }

    // Solicita un valor entero al usuario
    @Override
    public int PedirValorEntero(String pregunta) {
        System.out.println(pregunta);
        try {
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            return Integer.decode(br.readLine());
        }
        catch (Exception e) {
            return 0;
        }
    }

    // Solicita un valor booleano, con si (verdadero) o no.
    // Los errores se ignoran (devuelve falso)
    @Override
    public boolean PedirValorBooleano(String pregunta) {
        try {
            System.out.println(pregunta+ " (si, no)");
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            String res = br.readLine();
            return (res.equals("si")};
        }
        catch (IOException e) {
            return false;
        }
    }
}
```

Para la representación, se trata de simples llamadas a los métodos `toString`(). Sin embargo, para la visualización de los hechos, debemos ordenar los hechos por orden decreciente de nivel (gracias al método `sort`) y quitar los de nivel 0 (los hechos introducidos por el usuario). De este modo, los hechos obtenidos al final y de más alto nivel se mostrarán primero: "Triángulo rectángulo" se mostrará, así, antes que "Triángulo".

```
    @Override
    public void MostrarHechos(ArrayList<IHecho> hechos) {
        String res = "Solucion(es) encontrada(s) : \n";
        Collections.sort(hechos, (IHecho f1, IHecho f2) -> {
            return Integer.compare(f1.Nivel(), f2.Nivel()*(-1);
        });
        for(IHecho h: hechos) {
            if (h.Nivel() != 0) {
                res += h.toString() + "\n";
```

```
            }
        }
        System.out.println(res);
    }

    @Override
    public void MostrarReglas(ArrayList<Regla> reglas) {
        String res = "";
        for(Regla r: reglas) {
            res += r. toString() + "\n";
        }
        System.out.println(res);
    }

}
```

Esta clase se completará más adelante.

6. Motor de inferencia

A continuación debemos implementar la pieza central del sistema: un pequeño motor de inferencia de razonamiento deductivo, fácilmente adaptable a otros problemas de identificación (de plantas, de insectos, de animales, de rocas...).

Esta clase, llamada **`MotorInferencia`**, contiene en primer lugar tres atributos: una base de hechos, una base de reglas y una interfaz. El constructor inicializará las dos bases y recuperará la IHM que se le pase como parámetro. Se agrega un cuarto parámetro (`nivelMaxRegla`), aunque se utilizará posteriormente.

El código base de esta clase es el siguiente:

```
import java.util.ArrayList;

public class MotorInferencia {
    private BaseDeHechos bdh;
    private BaseDeReglas bdr;
    private IHM ihm;

    private int nivelMaxRegla;

    public MotorInferencia(IHM _ihm) {
        ihm = _ihm;
        bdh = new BaseDeHechos();
        bdr = new BaseDeReglas();
    }
}
```

Se completa la clase con dos métodos que permiten pedir (mediante la IHM) los valores booleanos o enteros de los hechos. No son más que redirecciones a los métodos de la interfaz.

```
    int PedirValorEntero(String pregunta) {
```

```
        return ihm.PedirValorEntero(pregunta);
    }

    boolean PedirValorBooleano(String pregunta) {
        return ihm.PedirValorBooleano(pregunta);
    }
```

Antes de continuar, debemos implementar un método que permita preguntar al usuario el valor de un hecho y crearlo (para agregarlo, posteriormente, a la base de hechos). Como el hecho puede ser de distintos tipos (`HechoEntero` o `HechoBooleano`) y de cara a ser lo más genérico posible, se utilizará una clase estática como fábrica de hechos. Esta clase se encargará de crear un hecho del tipo adecuado y devolverá un `IHecho`.

El código de esta clase **`HechoFactory`** es el siguiente (se completará más adelante):

```
class HechoFactory {
    static IHecho Hecho(IHecho h, MotorInferencia m) {
        try {
            IHecho nuevoHecho;
            Class clase = h.getClass();
            if
(clase.equals(Class.forName("sistemaexperto.HechoEntero"))) {
               nuevoHecho = CrearHechoEntero (f, m);
            }
            else {
               nuevoHecho = CrearHechoBooleano(f,m)
            }
            return nuevoHecho;
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(HechoFactory.class.getName()).
log(Level.SEVERE, null, ex);
            return null;
        }
    }
   static IHecho CrearHechoEntero (IHecho f, MotorInferencias m) {
        int valor = m.SolicitarValorEntero(f.Pregunta ());
        return new HechoEntero(f.Nombre(), valor, null, 0);
    }
    static IHecho CrearHechoBooleano(IHecho f, MotorInferencias m) {
        boolean valorB = m.SolicitarValorBooleano(f.Pregunta ());
        return new HechoBooleano(f.Nombre(), valorB, null, 0);
    }
}
```

Volvemos, ahora, a la clase `MotorInferencia`. Vamos a crear un método `EsAplicable`, que indica si una regla puede aplicarse (es decir, si se cumplen todas las premisas). Este método debe recorrer los hechos indicados como premisa y verificar si existen en la base de hechos.

Pueden darse varios casos:

- El hecho no está presente en la base de hechos: o bien posee una pregunta, y en este caso hay que consultar el valor al usuario y agregarlo a la base de hechos, o bien no posee ninguna pregunta (en cuyo caso es un hecho únicamente inferible), y la regla no podrá aplicarse.

- El hecho está presente en la base de hechos: o bien el valor se corresponde, en cuyo caso pasamos al hecho siguiente, o bien el valor no se corresponde, en cuyo caso la regla no se aplicará.

Además, durante el recorrido de los hechos, es preciso buscar cuál es el nivel máximo de las premisas. En efecto, si la regla se aplica, el hecho inferido tendrá como nivel el de las premisas + 1. De este modo, una regla que tenga premisas de nivel 0 y 3 creará un hecho de nivel 4. Se devolverá este último valor. Como es obligatoriamente positivo, devolveremos un **-1** en el caso de que la regla no pueda aplicarse.

```
    int EsAplicable(Regla _r) {
        int nivelMax = -1;
        for (IHecho h : _r.getPremisas()) {
            IHecho hechoEncontrado = bdh.Buscar(h.Nombre());
            if (hechoEncontrado == null) {
                if (h.Pregunta() != null) {
                    hechoEncontrado = HechoFactory.Hecho(h, this);
                    bdh.AgregarHecho(hechoEncontrado);
                }
                else {
                    return -1;
                }
            }

            if (!hechoEncontrado.Valor().equals(h.Valor())) {
                return -1;
                nivelMax = Math.max(nivelMax, hechoEncontrado.
Nivel());
            }
        }
        return nivelMax;
    }
```

El siguiente método, `EncontrarRegla`, permitirá buscar, entre todas las reglas de la base, la primera que pueda aplicarse. Se basa por tanto en `EsAplicable`.

Si puede aplicarse alguna regla, esta se devuelve, y el atributo `nivelMaxRegla` del motor se modifica para contener el nivel de la regla devuelta. En caso de que no pueda aplicarse ninguna regla, se devuelve `null`.

```
    Regla EncontrarRegla(BaseDeReglas bdrLocal) {
        for(Regla r : bdrLocal.getReglas()) {
            int nivel = EsAplicable(r);
            if (nivel != -1) {
                nivelMaxRegla = nivel;
                return r;
            }
        }
        return null;
    }
```

El método principal del motor, `Resolver()`, permite resolver completamente un problema. El código de este método es bastante sencillo:

- Se realiza una copia local de todas las reglas existentes y se inicializa la base de hechos (vacía).
- Mientras pueda aplicarse alguna regla:
  - Se aplica, y el hecho inferido se agrega a la base de hechos (incrementando su nivel).
  - Se elimina (para no volver a ejecutarla más adelante sobre el mismo problema).
- Cuando no quedan más reglas aplicables, se muestra el resultado.

He aquí el código de este método:

```
    public void Resolver() {
       BaseDeReglas bdrLocal = new BaseDeReglas();
       bdrLocal.setReglas(bdr.getReglas());

       bdh.Vaciar();

       Regla r = EncontrarRegla(bdrLocal);
       while(r!=null) {
           IHecho nuevoHecho = r.conclusion;
           nuevoHecho.setNivel(nivelMaxRegla + 1);
           bdh.AgregarHecho(nuevoHecho);
           bdrLocal.Eliminar(r);
           r = EncontrarRegla(bdrLocal);
       }

       ihm.MostrarHechos(bdh.getHechos());
   }
```

Llegados a este punto, la aplicación de las reglas está gestionada completamente. No queda más que construir algún medio para agregar las reglas. Como para solicitar al usuario un hecho, vamos a tener que proveer un medio para leer la regla escrita, extraer los hechos y crear un hecho de la clase correspondiente en función de si se trata de un hecho entero o booleano.

Para simplificar el trabajo, agregaremos un nuevo método en **`HechoFactory`**, que permitirá crear un hecho a partir de una cadena. Los hechos se expresarán con la forma "Nombre=Valor (pregunta)" para un hecho entero y "Nombre(pregunta)"/"!Nombre(pregunta)" para un hecho booleano.

He aquí el código de este método `Hecho`, que devuelve un `IHecho` en función de la cadena recibida. El código consiste en dividir la cadena mediante los separadores ("=", "(",")", "!"), eliminar los espacios al principio y al final de la cadena (método `trim()`) o `replaceFirst()` si la cadena empieza por una palabra clave) y crear el hecho correspondiente con el valor correcto. Completaremos también la pregunta si esta se ha proporcionado (en cuyo caso no será un hecho solo inferido).

```
   static IHecho Hecho(String hechoStr) {
       hechoStr = hechoStr.trim();
       if (hechoStr.contains("=")) {
           // Existe el símbolo "=", se trata de un hecho entero
           hechoStr = hechoStr.replaceFirst("^" + "\\(", "");
           String[] nombreValorPregunta = hechoStr.split("[=()]");
           if (nombreValorPregunta.length >= 2) {
               String pregunta = null;
```

```
                if (nombreValorPregunta.length == 3) {
                    pregunta = nombreValorPregunta[2].trim();
                }
                return new HechoEntero(nombreValorPregunta[0].trim(),
Integer.parseInt(nombreValorPregunta[1].trim()), pregunta, 0);
            }
        }
        else {
            // Es un hecho booleano nombre[(pregunta)] o
!nombre[(pregunta)]
            boolean valor = true;
            if (hechoStr.startsWith("!")) {
                valor = false;
                hechoStr = hechoStr.substring(1).trim();
            }
            // Split, tras eliminar el primer delimitador
si es necesario : "("
            hechoStr = hechoStr.replaceFirst("^" + "\\(", "");
            String[] nombrePregunta= hechoStr.split("[()]");
            String pregunta = null;
            if (nombrePregunta.length == 2) {
                pregunta = nombrePregunta[1].trim();
            }
            return new HechoBooleano(nombrePregunta[0].trim(),
valor, pregunta, 0);
        }
        return null;
    }
```

La clase **`MotorInferencia`** puede, a continuación, terminarse con un método que permita agregar una regla a partir de una cadena de caracteres (lo cual resultará más sencillo para el usuario). Este método divide, en primer lugar, la cadena a partir del símbolo ":" para separar el nombre de la regla. A continuación, separando la cadena en las palabras clave IF y THEN, podemos separar las premisas de la conclusión. En el caso de las premisas, las separaremos identificando la presencia de 'AND'.

Las reglas proporcionadas mediante la implementación de este método no podrán contener las palabras IF, THEN ni los siguientes símbolos: "=", ":", "(", ")". En efecto, sirven de separadores.

He aquí el código de este método `AgregarRegla`:

```
    public void AgregarRegla(String str) {
     // Separación nombre:regla
     String[] nombreRegla = str.split(":");
     if (nombreRegla.length == 2) {
          String nombre = nombreRegla[0].trim();
          // Separación premisas THEN conclusión
          String regla = nombreRegla[1].trim();
          regla = regla.replaceFirst("^" + "IF", "");
          String[] premisasConclusion = regla.split("THEN");
          if (premisasConclusion.length == 2) {
             // Lectura de las premisas
             ArrayList<IHecho> premisas = new ArrayList();
```

```
            String[] premisasStr =
premisasConclusion[0].split(" AND ");
            for(String cadena : premisasStr) {
            IHecho premisa = HechoFactory.Hecho(cadena.trim());
                   premisas.add(premisa);
              }

            // Lectura de la conclusión
            String conclusionStr = premisasConclusion[1].trim();
            IHecho conclusion = HechoFactory.Hecho(conclusionStr);

            // Creación de la regla e incorporación a la base
            bdr.AgregarRegla(new Regla(nombre, premisas,
conclusion));
           }
        }
    }
```

7. Escritura de reglas y uso

El sistema experto está completo. Es genérico, puede aplicarse a cualquier problema.

La clase **`Aplicacion`** se completa mediante el método `main` para poder resolver el problema del nombre de los polígonos.

El `main` será sencillo, no hará más que invocar al método `Run` descrito a continuación:

```
   public static void main(String[] args) {
         Aplicacion app = new Aplicacion();
         app.Run();
   }
```

El método `Run` tendrá que:

- Crear un nuevo motor.
- Agregar las reglas, mediante su versión textual. Aquí, solo se agregarán las once reglas correspondientes a los triángulos y cuadriláteros, pero es fácil completarlas con nuevas reglas. Son también fáciles de leer por el usuario o por su diseñador.
- Ejecutar la resolución del problema.

He aquí su código:

```
    public void Run() {
        // Creación del motor
        System.out.println("** Creación del motor **");
        MotorInferencia m = new MotorInferencia(this);

        // Agregar las reglas
        System.out.println("** Agregar las reglas **");
```

```
        m.AgregarRegla("R1 : IF (Orden=3(¿Cuál es el orden?)) THEN
Triángulo");
        m.AgregarRegla("R2 : IF (Triángulo AND Ángulo Recto(¿La figura
tiene al menos un ángulo recto?)) THEN Triángulo Rectángulo");
        m.AgregarRegla("R3 : IF (Triángulo AND Lados Iguales=2(¿Cuántos
lados iguales tiene la figura?)) THEN Triángulo Isósceles");
        m.AgregarRegla("R4 : IF (Triángulo rectángulo AND Triángulo
Isósceles) THEN Triángulo Rectángulo Isósceles");
        m.AgregarRegla("R5 : IF (Triángulo AND Lados Iguales=3(¿Cuántos
lados iguales tiene la figura?)) THEN Triángulo Equilátero");
        m.AgregarRegla("R6 : IF (Orden=4(¿Cuál es el orden?)) THEN
Cuadrilátero");
        m.AgregarRegla("R7 : IF (Cuadrilátero AND Lados
Paralelos=2(¿Cuántos lados paralelos entre sí - 0,
2 o 4?)) THEN Trapecio");
vm.AgregarRegla("R8 : IF (Cuadrilátero AND Lados
Paralelos=4(¿Cuántos lados paralelos entre sí - 0,
2 o 4?)) THEN Paralelogramo");
        m.AgregarRegla("R9 : IF (Paralelogramo AND Ángulo Recto(¿La
figura tiene al menos un ángulo recto?)) THEN Rectángulo");
        m.AgregarRegla("R10 : IF (Paralelogramo AND Lados
Iguales=4(¿Cuántos lados iguales tiene la figura?)) THEN Rombo");
        m.AgregarRegla("R11 : IF (Rectángulo AND Rombo THEN Cuadrado");

        // Resolución
        while (true) {
            System.out.println("\n** Resolución **");
            m.Resolver();
        }
    }
```

He aquí el tipo de salida que podemos obtener (aquí se determina un triángulo rectángulo isósceles y, a continuación, un rectángulo):

```
** Creación del motor **
** Agregar las reglas **

** Resolución **
¿Cuál es el orden?
3
¿La figura tiene al menos un ángulo recto? (sí, no)
sí
¿Cuántos lados iguales tiene la figura?
2
Solución(es) encontrada(s):
Triángulo Rectángulo Isósceles (3)
Triángulo Rectángulo (2)
Triángulo Isósceles (2)
Triángulo (1)

** Resolución **
¿Cuál es el orden?
4
```

```
¿Cuántos lados paralelos entre sí - 0, 2 o 4?
4
¿La figura tiene al menos un ángulo recto? (sí, no)
sí
¿Cuántos lados iguales tiene la figura?
2
Solución(es) encontrada(s):
Rectángulo (3)
Paralelogramo (2)
Cuadrilátero (1)
```

Nuestro sistema está ahora completamente implementado.

Uso de Prolog

Es posible codificar un sistema experto en programación orientada a objetos, pero no es el paradigma de programación mejor adaptado. Los lenguajes de **programación lógica** están construidos para ejecutar este tipo de tarea. La escritura de código será más sencilla, pues toda la lógica del motor ya está implementada en el núcleo del lenguaje.

1. Presentación del lenguaje

Prolog, de PROgrammation LOGique, es uno de los primeros lenguajes de este paradigma, creado en 1972 por dos franceses: Alain Colmerauer y Philippe Roussel.

No es, no obstante, el único lenguaje de programación lógica. Podemos citar también Oz o CLIPS. Estos lenguajes son próximos a los de la programación funcional (LISP, Haskell...), que pueden utilizarse en sistemas expertos.

Prolog contiene un **motor de inferencia de razonamiento inductivo**, con backtracking. Se le define un objetivo (que debe ser un hecho) que intentará resolver. Si el objetivo contiene alguna variable, buscará todos los valores posibles para dicha variable. Si el objetivo contiene un atributo, confirmará o descartará el hecho.

Prolog funciona sobre la base de **predicados**. Cada predicado puede ser un hecho comprobado, o bien un hecho inferido gracias a las reglas. Por ejemplo, orden(3) es un predicado con un parámetro que es un hecho definido por el usuario. Por el contrario, nombre (triángulo) será un hecho inferido por la regla "SI orden(3) ENTONCES nombre (triángulo)".

El lenguaje contiene también predicados predefinidos que permiten, por ejemplo, escribir y leer en la consola de Prolog, cargar un archivo o manipular listas.

En lo sucesivo, utilizaremos el SWI-Prolog, que puede descargar gratuitamente del sitio web http://www.swi-prolog.org/ y que está disponible para Windows, Mac OS X o Linux.

2. Sintaxis del lenguaje

Aquí se detalla únicamente la sintaxis necesaria para la comprensión de los ejemplos. No dude en consultar la documentación de su implementación de Prolog para saber más. Además, según la función utilizada, la sintaxis puede variar ligeramente. En el anexo hallará una explicación del uso de SWI-Prolog para Windows.

a. Generalidades

En Prolog, deben diferenciarse dos partes:

- El archivo que contiene las reglas y los hechos (es decir, el conjunto de predicados definidos para el problema).
- La consola que sirve únicamente para interactuar con el usuario.

En el archivo cargado, podemos encontrar en primer lugar **comentarios**. Estos pueden ser de dos tipos:

```
% Comentario en una única línea
```

```
/* Comentario que puede
```

```
ocupar varias líneas */
```

b. Predicados

Encontramos, a continuación, los **predicados**, que deben empezar necesariamente por una letra minúscula. Terminan con un punto, similar al punto y coma que anuncia el final de una instrucción en un lenguaje orientado a objetos. Los predicados pueden recibir parámetros, que se indican entre paréntesis. Si estos parámetros son **variables**, entonces su nombre debe comenzar por una letra mayúscula; en caso contrario empiezan por minúscula (existe un caso particular: "_", que representa una variable anónima, cuyo valor no tiene importancia).

He aquí algunos ejemplos de predicados:

```
comer(gato, ratón).
comer(ratón, queso).

piel(gato).
piel(ratón).
```

También podemos leer que el gato se come al ratón, y que el ratón come queso. Además, el gato y el ratón poseen, ambos, piel.

c. Plantear preguntas

Este archivo tan sencillo ya es ejecutable. Basta con cargarlo en la consola. Es posible, a continuación, plantear preguntas. Cuando Prolog devuelve un resultado, si este no termina con un punto, significa que existen (potencialmente) otras respuestas. Pulsando la tecla ";" pueden obtenerse las siguientes, hasta que falle (`false.`) o aparezca un punto.

El primer tipo de pregunta permite, simplemente, conocer si un hecho es verdadero o falso. También podemos preguntar si el gato come queso (es falso) o si el ratón come queso (es verdadero). La presencia de un punto al final de la respuesta indica que Prolog no podrá proporcionar otras respuestas:

```
?- comer(gato, queso).
false.

?- comer(ratón, queso).
true.
```


Las líneas que empiezan con "?-" se escriben siempre en la consola de Prolog. "?-" es el prompt.

Es posible, también, preguntar quién come queso. Esta vez, se utiliza una variable, cuyo nombre debe empezar por mayúscula. Por convención, se utiliza X, pero no es obligatorio (podríamos llamarla ComedoresDeQueso si así lo deseáramos):

```
?- comer(X, queso).
X = ratón.
```

Prolog realiza lo que denominamos una **unificación**: busca los posibles valores para la variable X. Aquí se obtiene un único resultado.

Además, es posible preguntar quién come qué/quién, y quién tiene piel. Esta vez, se van a pedir los siguientes resultados con ";":

```
?- comer(X, Y).
X = gato,
Y = ratón ;
X = ratón,
Y = queso.

?- piel(X).
X = gato ;
X = ratón.
```

En ambos casos, vemos que se obtienen dos resultados: las parejas (gato, ratón) y (ratón, queso) en el primer caso, y gato, así como ratón, en el segundo.

d. Escritura de las reglas

Aunque nuestro archivo cargado contiene únicamente cuatro líneas, que son hechos, este ya es un programa funcional. Sin embargo, sobre todo en el caso de un sistema experto, es conveniente poder agregar nuevas reglas.

El formato de las **reglas** es el siguiente:

```
conclusión :-
    premisa1,
    premisa2,
    % otras premisas
    premisaN.
```

En este caso, las variables pueden unificarse. Tomemos como ejemplo una regla que diga que los enemigos de nuestros enemigos son nuestros amigos (utilizando la relación comer), que se escribe de la siguiente manera:

```
amigos(X, Y) :-
      comer(X, Z),
      comer(Z, Y).
```

Una vez agregada esta regla al archivo y cargado, es posible preguntar quién es amigo de quién, en la consola:

```
?- amigos(X, Y).
X = gato,
Y = queso ;
false.
```

Existe, potencialmente, una única pareja de amigos, el gato y el queso (puesto que el gato se come al ratón, que come queso).

Es posible tener más reglas para el mismo predicado, siempre que respeten todas ellas la misma firma (y, por lo tanto, el mismo número de parámetros) y que sean diferentes.

e. Otros predicados útiles

Es posible, también, indicar que una **regla fracasa** si se completan ciertas condiciones con la palabra clave "`fail`".

En nuestro caso, cuando una regla falle, si el predicado posee otras, el motor va a comprobarlas también gracias al mecanismo de backtracking. Existe, sin embargo, un medio de impedir este backtracking, para evitar comprobar nuevas reglas cuando se sabe que la regla no se cumplirá jamás: el **operador "cut"**, que se representa mediante un signo de exclamación.

Existen también formas de utilizar una minibase de datos incluida en el lenguaje que permite recordar hechos que acaban de deducirse, para no tener que volver a encontrarlos si existen otras reglas que los necesitan. A la inversa, es posible eliminar los hechos registrados. Estos predicados son las series `assert` (`assert`, `assertz`, `asserta`) y `retract` (`retract` y `retractall`). Se utilizarán en el ejemplo. Las entradas/salidas con el usuario utilizan los predicados de Prolog `read` y `write`.

Por último, Prolog contiene varios predicados que permiten manipular listas, crearlas, recorrerlas...

3. Codificación del problema de las formas geométricas

Se crea un nuevo proyecto para los nombres de los polígonos. Como con la versión en Java, este ejemplo estará limitado a los triángulos y cuadriláteros, pero es sencillo completar las reglas.

Existen varios predicados que se corresponden con información del tipo: `ladosIguales(X)`, que permite indicar el número de lados iguales en tamaño; `anguloRecto(X)`, que vale sí o no e indica la presencia de al menos un ángulo recto; `ladosParalelos(X)`, que indica el número de lados paralelos entre sí (0, 2 o 4), y `orden(X)`, que indica el número de lados.

El nombre de la forma será del tipo `nombre(X)`.

Es posible, a continuación, escribir las distintas reglas siguiendo la sintaxis de Prolog:

```
%********** Reglas **********
% Triángulos
nombre(triangulo) :-
   orden(3).

nombre(trianguloIsosceles) :-
   nombre(triangulo),
   ladosIguales(2).

nombre(trianguloRectangulo) :-
   nombre(triangulo),
   anguloRecto(si).

nombre(trianguloRectanguloIsosceles) :-
   nombre(trianguloIsosceles),
```

```
   nombre(trianguloRectangulo).

nombre(trianguloEquilatero) :-
   nombre(triangulo),
   ladosIguales(3).

% Cuadriláteros
nombre(cuadrilatero) :-
   orden(4).

nombre(trapecio) :-
   nombre(cuadrilatero),
   ladosParalelos(2).

nombre(paralelogramo) :-
   nombre(cuadrilatero),
   ladosParalelos(4).

nombre(rectangulo) :-
   nombre(paralelogramo),
   anguloRecto(si).

nombre(rombo) :-
   nombre(paralelogramo),
   ladosIguales(4).

nombre(cuadrado) :-
   nombre(rombo),
   nombre(rectangulo).
```

Podemos encontrar las mismas reglas que en el código en Java; sin embargo, la lectura de estas por parte de alguien que no sea experto puede resultar algo más compleja debido a la sintaxis.

La gestión de los hechos será también más complicada. Se plantean dos casos:

- El hecho está presente en memoria: o bien se valida el valor correcto, o bien tiene un valor incorrecto y se detiene aquí diciendo que la regla ha fracasado (y no se buscan otros medios para resolverla).
- El hecho no está presente en memoria: se le pregunta al usuario, se lee su respuesta, se almacena en memoria y se consulta si el valor respondido se corresponde con el esperado.

Se crea un predicado `memory`. Este recibe dos parámetros: el nombre del atributo y su valor. Como en un principio no hay nada en la memoria, y agregaremos los hechos conforme se avance, este predicado debe indicarse como dinámico en el archivo de reglas:

```
:- dynamic memory/2.
```

Para tratar los valores de los hechos, se crea un predicado `ask`, que recibe tres parámetros:

- El hecho que se busca.
- La pregunta que se desea plantear al usuario.

- La respuesta obtenida.

Existen tres reglas asociadas (que deben mantenerse en este orden). La primera se corresponde con el caso en que el hecho ya esté presente en memoria, con el valor correcto. La pregunta que se ha de plantear ya no es importante, viendo que la respuesta ya se conoce, y será una variable anónima de la regla. En este caso, la regla funciona:

```
ask(Pred, _, X) :-
   memory(Pred, X).
```

Para la segunda regla, se sabe que se evaluará únicamente si la primera fracasa. Se busca, por tanto, un valor (anónimo) en memoria. Si se encuentra alguno, significa que el hecho ya está definido, pero con un valor diferente al esperado. La regla va a fracasar, y se agrega el `cut` para estar seguros de no evaluar otra regla.

```
ask(Pred, _, _) :-
   memory(Pred, _),
   !,
   fail.
```

Por último, la tercera regla es relativa al caso en que el hecho no se ha encontrado en memoria hasta el momento (por parte de las dos primeras reglas). La pregunta se plantea con `write`, se recoge la respuesta con `read` y se almacena el hecho en memoria con `asserta` (que agrega al principio). Por último, se consulta si el valor obtenido es el valor esperado. Si sí, la regla tiene éxito; en caso contrario fracasa, y como no existen otras reglas el predicado solicitado se considera como falso.

```
ask(Pred, Question, X) :-
   write(Question),
   read(Y),
   asserta(memory(Pred, Y)),
   X == Y.
```

Los cuatro hechos que pueden solicitarse al usuario basándose en `ask` se agregan a las reglas:

```
ladosIguales(X) :- ask(ladosIguales, '¿Cuántos lados iguales tiene
la figura? ', X).

anguloRecto(X) :- ask(anguloRecto, '¿La figura posee
ángulos rectos (sí, no)? ', X).

ladosParalelos(X) :- ask(ladosParalelos, '¿Cuántos lados paralelos
tiene la figura (0, 2 o 4)? ', X).

orden(X) :- ask(orden, '¿Cuántos lados? ', X).
```

Finalmente, incorporamos una última regla que permite agregar un caso. Esta regla borrará, en primer lugar, todos los hechos registrados en memoria y, a continuación, preguntará gracias al predicado Prolog "`findAll`" todos los valores de X que pueden unificarse con `nombre(X)` (lo que equivale a preguntar todos los nombres de la forma). El resultado se almacena en la variable R que se muestra a continuación.

En Prolog, esta regla se escribe de la siguiente manera:

```
solve :-
   retractall(memory(_,_)),
   findall(X, nombre(X), R),
   write(R).
```

Hemos completado el archivo de reglas, así como el programa. No contiene más que unas cincuenta líneas de código, a diferencia del programa escrito en Java. Para utilizarlo, basta con invocar desde la consola el predicado `solve`.

He aquí algunos ejemplos de diálogo obtenidos (para un triángulo rectángulo isósceles y a continuación para un rectángulo):

```
?- solve.
¿Cuántos lados? 3.
¿Cuántos lados iguales tiene la figura? 2.
¿La figura posee ángulos rectos (sí, no) ? Sí.
[triangulo,trianguloIsosceles,trianguloRectangulo,
trianguloRectanguloIsosceles] true.

?- solve.
¿Cuántos lados? 4.
¿Cuántos lados paralelos posee la figura (0, 2 o 4)? 4.
¿La figura posee ángulos rectos (sí, no)? Sí.
¿Cuántos lados iguales tiene la figura? 2.
[cuadrilatero,paralelogramo,rectangulo]
true.
```


Preste atención: para validar una respuesta solicitada por `read`, debe terminar la línea con un punto.

4. Codificación del problema de las ocho reinas

a. Interés del razonamiento inductivo

Prolog utiliza un razonamiento inductivo, es decir, parte del objetivo que se ha de alcanzar y busca todas las reglas que le permiten llegar a él. En el ejemplo anterior, deberíamos haberle indicado que buscábamos asociar un nombre a la forma. Un razonamiento deductivo habría resultado más sencillo de implementar (pero no está integrado en Prolog).

El razonamiento inductivo es útil, sobre todo, cuando debe realizarse mucho backtracking, para comprobar otras soluciones cuando alguna fracasa. El problema de las 8 reinas es un caso típico. En efecto, buscaremos la manera de colocar nuestras reinas, e intentaremos otras posibilidades hasta obtener una que funcione. A continuación tendremos que recordar todas las opciones probadas para no volver a intentarlas.

Escribir este programa en Java es factible. No obstante, resulta poco legible y se extiende mucho más allá del diseño de un sistema experto, las reglas desaparecen para dar paso a bucles que permitan comprobar las

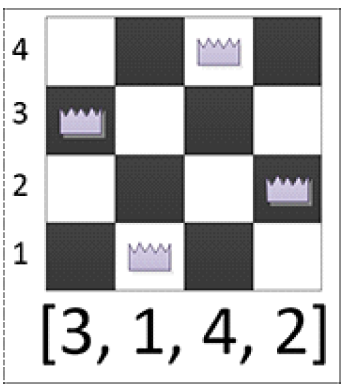
distintas posiciones.

b. Estudio del problema

El código será una versión genérica que permita resolver el problema de las N reinas (se trata, por lo tanto, de situar N reinas sobre un tablero de NxN casillas). Sin embargo, esta solución no está del todo optimizada para conservar la legibilidad, y el tiempo de cálculo a partir de N=15 empieza a resultar importante en máquinas corrientes (los códigos Prolog más optimizados permiten superar N=20, pero ninguno es capaz de responder en un tiempo aceptable a valores de N>30).

Antes de pasar al código, es importante comprender la lógica de este problema. Hemos visto en este capítulo que resulta importante conocer bien la representación de la solución. En este caso, se trata de una lista de N valores, todos diferentes. Esto permite asegurar que cada reina está situada sobre una columna diferente (definida por la posición en la lista) y una fila diferente (de modo que todos los números serán diferentes).

De este modo, sobre un tablero de 4x4, se podría obtener la solución [3, 1, 4, 2] correspondiente a la siguiente organización:



c. Reglas que se han de aplicar

Se debe asegurar que los números de la lista solución son permutaciones de la lista [0,..., N]. Aquí, [3, 1, 4, 2] es, efectivamente, una permutación de [1, 2, 3, 4]. Prolog nos permite construir fácilmente la lista de valores enteros de 1 a N gracias al predicado `numlist`, y realizar permutaciones mediante el predicado `permutation`.

La representación ha eliminado los problemas de las filas y las columnas, de modo que solo queda por verificar que dos reinas no entren en conflicto sobre la misma diagonal. Para cada reina, será preciso verificar que no está situada en la misma diagonal que todas las reinas siguientes.

Se produce, por lo tanto, un doble bucle: se deben comprobar todas las reinas de la solución, y compararlas con todas las reinas restantes. Son necesarios dos predicados diferentes.

El recorrido de las listas es únicamente recursivo en Prolog: será preciso determinar la condición de parada, pues el caso general debe invocar el predicado en curso. Además, Prolog permite recorrer una lista separando únicamente el elemento situado en la cabeza de la lista. De este modo, [C | L] representa una lista que empieza por el elemento C, seguido de la lista L.

Las condiciones de parada se producirán cuando la lista de los siguientes elementos esté vacía: una reina no puede entrar en conflicto con una lista vacía de otras reinas.

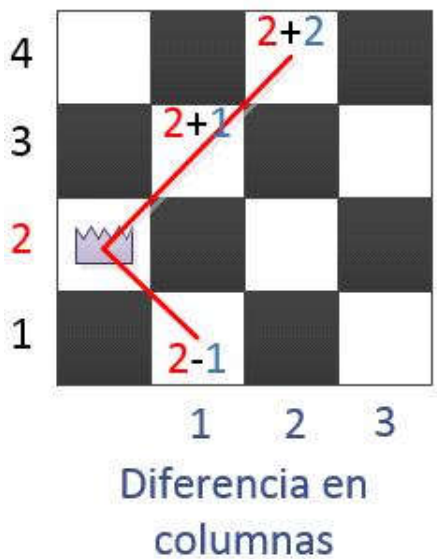
d. Reglas de conflicto entre reinas

Una vez realizado el estudio del problema, es posible pasar a la escritura del problema en Prolog.

El primer predicado debe indicar si la reina en curso está en conflicto con la lista de reinas proporcionada, y que empieza en Col columnas más adelante. De este modo, en el ejemplo descrito para las 4 reinas [3, 1, 4, 2], se comprueba si la reina situada en 3 está en conflicto con la reina situada en 1 a 1 columna de distancia, a continuación con la reina situada en 4 a 2 columnas de distancia, y por último con la reina situada en 2 a 3 columnas de distancia.

Este número de columnas es importante, En efecto, dos reinas estarán sobre una misma diagonal si el número de la reina en curso más la diferencia de columnas se corresponde con la posición de la reina comprobada para una diagonal ascendente (es preciso restar el número de columnas para comprobar la diagonal descendente).

El siguiente esquema muestra un ejemplo de diagonal para una reina situada en segunda fila: vemos cómo debe agregarse a la fila en curso la diferencia de columnas para producirse el caso de que se encuentren en la misma diagonal ascendente (o restarse para la diagonal descendente).



De este modo, en [3, 1, 4, 2], no existe ninguna reina en conflicto con la reina en 3. En efecto, 3+1 y 3-1 son diferentes a 1 (la posición de la reina a 1 columna), 3+2 y 3-2 son diferentes de 4 (la posición de la reina a 2 columnas) y 3+3 y 3-3 son diferentes de 2 (la posición de la reina a 3 columnas).

El caso de parada es, simplemente, la lista vacía, indicando que no existe ningún conflicto y que la posición funciona.

Tenemos, por lo tanto, las dos reglas siguientes para el predicado `diagReina` que recibe como parámetro la reina en curso, seguido de la lista de las demás reinas, y por último la distancia en columnas (el símbolo =\= indica la diferencia):

```
diagReina(_, [], _) :-
    true.

diagReina(Reina, [C|L], Col) :-
    (Reina + Col) =\= C,
    (Reina - Col) =\= C,
    diagReina(Reina, L, Col+1).
```

Ahora se sabe si alguna reina concreta entra en conflicto con las demás. Es necesario realizar un bucle para recorrer todas las reinas. Se trabaja de manera recursiva: una reina no está en conflicto si no quedan más reinas detrás, y si una reina determinada no está en conflicto entonces se comprueba la lista sin esta reina.

He aquí el predicado `diagsOK` que recibe como parámetro la lista actual:

```
diagsOK( [ _ | [] ] ) :-
    true.

diagsOK([Cabeza | Lista]) :-
    diagReina(Cabeza, Lista, 1),
    diagsOK(Lista).
```

e. Objetivo del programa

Una vez comprobadas las diagonales, tan solo queda escribir el objetivo denominado "`reinas`" y que recibe como parámetro el número de reinas que se han de ubicar. Este debe crear una lista de 1 a N, que nos servirá de base, y a continuación buscar todas las permutaciones de esta lista que respetan las restricciones impuestas sobre las diagonales:

```
reinas(N, Res) :-
   numlist(1,N,Base),
   permutation(Res,Base),
   diagsOK(Res).
```

Esta vez, pedimos mostrar únicamente la primera solución encontrada. Podrán obtenerse más soluciones presionando ";" cada vez que Prolog proponga una solución. En efecto, si bien solo hay dos soluciones posibles para el problema de las 4 reinas, existen 92 para las 8 reinas y el número aumenta de forma exponencial (si bien no existe una fórmula exacta que permita calcularlo).

f. Ejemplos de uso

He aquí algunas llamadas posibles por consola, pidiendo todas las soluciones para N = 4, a continuación para N = 6 y por último únicamente la primera solución para N = 8:

```
?- reinas(4, Res).
Res = [3, 1, 4, 2] ;
Res = [2, 4, 1, 3] ;
false.

?- reinas(6, Res).
Res = [4, 1, 5, 2, 6, 3] ;
Res = [5, 3, 1, 6, 4, 2] ;
Res = [2, 4, 6, 1, 3, 5] ;
Res = [3, 6, 2, 5, 1, 4] ;
false.

?- reinas(8, Res).
Res = [1, 7, 5, 8, 2, 4, 6, 3] .
```

Este programa, no optimizado pero que utiliza el backtracking nativo de Prolog, no ocupa más que 15 líneas de código, definiendo incluso las reglas que permiten deducir si dos reinas no están en conflicto. El interés de los lenguajes de programación lógica en casos como este resulta evidente.

Las versiones más rápidas están codificadas, sin embargo, en C (en varios cientos de líneas). Prolog permite simplificar la escritura del programa, pero no siempre es el más eficaz en términos de tiempo de respuesta. Existen, no obstante, soluciones en Prolog menos legibles pero mucho más rápidas que la propuesta aquí.

Resumen

Un sistema experto permite partir de hechos y aplicar reglas para obtener nuevos hechos, llamados hechos inferidos. Reemplazan o complementan el conocimiento del experto, según el caso.

Están compuestos por una base de reglas que contiene todas las reglas conocidas por el experto y que permite llegar a conclusiones. Se componen también de una base de hechos que representan todo lo que se conoce por parte del usuario, así como los hechos inferidos hasta el momento. Una interfaz de usuario permitirá comunicarse de forma clara con las distintas personas que utilicen el sistema.

El núcleo del sistema experto es su motor de inferencia. Es el que va a escoger y aplicar las reglas, y ejecutar las interacciones con el usuario. Puede ser de razonamiento deductivo si parte de los hechos para obtener otros nuevos o de razonamiento inductivo si parte de un objetivo y busca cómo alcanzarlo.

La creación de este motor no es una tarea sencilla, pero es posible implementarlo en todos los lenguajes. Los motores de razonamiento deductivo son, sin embargo, mucho más fáciles de codificar, por ejemplo en Java. Existen lenguajes particulares, propios de la programación funcional o lógica, como Prolog, que permiten simplificar la implementación de sistemas expertos. En efecto, el motor es una parte integrante del lenguaje.

Por último, es posible agregar la gestión de incertidumbre, tanto a nivel de los hechos introducidos por el usuario como en las reglas, para dominios más difíciles de modelar o en aquellos casos en los que el usuario no puede responder con total certeza a las cuestiones planteadas por el sistema.

Los sistemas expertos, por su facilidad de implementación, su potencia y su facilidad de uso, se encuentran actualmente en numerosos dominios, ya sea en el diagnóstico, la estimación de riesgos, la planificación y la logística o la transferencia de conocimientos y de competencias.

Presentación del capítulo

La lógica difusa es una técnica de inteligencia artificial determinista que permite tomar decisiones. Permite, también, obtener un comportamiento coherente y reproducible en función de las reglas que se le proveen. La inteligencia de esta técnica se encuentra en su capacidad de gestionar la imprecisión y presentar un comportamiento más flexible que un sistema informático tradicional.

Este capítulo empieza definiendo la noción de imprecisión, para no confundirla con incertidumbre. A continuación, se abordan los distintos conceptos: los conjuntos difusos, las funciones de pertenencia y los distintos operadores en lógica difusa.

La siguiente sección aborda las reglas difusas y los pasos para aplicar estas reglas a un caso concreto y obtener un resultado utilizable (se denominan, respectivamente, fuzzificación y defuzzificación).

El capítulo continúa con la presentación de diversos dominios de aplicación de la lógica difusa, que encontramos actualmente desde en nuestras lavadoras hasta en nuestros coches, pasando por los motores.

Para terminar, la última sección consiste en mostrar cómo puede implementarse un motor de lógica difusa genérico y evolutivo en Java. Se detallan las diferentes clases y puede descargarse el código completo. Se provee, también, un ejemplo de uso de este motor. Este capítulo finaliza con un resumen: en él se recuerdan los principales resultados.

Incertidumbre e imprecisión

Es muy importante distinguir dos conceptos: incertidumbre e imprecisión. En efecto, cada uno va a estar asociado a una técnica de inteligencia artificial diferente.

1. Incertidumbre y probabilidad

La **incertidumbre** es, evidentemente, lo contrario de la certidumbre. Por ejemplo, la regla "Si va a llover, entonces cogeré el paraguas" es segura (al 100 %): mojarse no resulta agradable. Por el contrario, el enunciado "Mañana debería llover" es incierto: el parte meteorológico ha podido anunciar lluvias, pero nada obliga a que tenga que llover necesariamente. Podríamos decir que la probabilidad de que llueva es del 80 %, por ejemplo. Cualquier enunciado cuya probabilidad sea diferente del 100 % es incierto.

2. Imprecisión y subjetividad

Por el contrario, la **imprecisión** se manifiesta cuando falta... ¡precisión! En los hechos, esto se traduce en enunciados que resulta difícil evaluar: parecen subjetivos. Por ejemplo, en la frase "Si hace mucho calor, entonces no cogeré chaqueta", la imprecisión se sitúa en la expresión "mucho calor".

Es cierto que si hace "mucho calor" no cogeré la chaqueta, en cuyo caso no hay incertidumbre. Pero ¿hace mucho calor a 35º? Probablemente sí. ¿Y a 30º? De algún modo, podríamos afirmar que hace "mucho calor" a partir de 30º. Pero en este caso, ¿qué ocurre si la temperatura es 29,5º?

Nos damos cuenta de que para un ser humano "mucho calor" es una noción muy difusa: depende de la persona, del lugar, del contexto... de modo que "mucho calor" para un gallego no tendrá, probablemente, el mismo significado que para un andaluz. Además, no utilizamos un termómetro para realizar la evaluación, sino que nos basamos en la propia percepción. Se es muy impreciso. La mayoría de nuestras decisiones sufren esta imprecisión. Es así, por ejemplo, cuando decidimos (o no) cruzar una carretera a través de los huecos libres (estimando la velocidad de los demás coches y su distancia) o nuestra manera de vestirnos (en función del tiempo que hace).

3. Necesidad de tratar la imprecisión

Podemos imaginar una persiana gestionada por un sistema informático clásico. Si le definimos la regla "si la temperatura es superior o igual a 25º, entonces baja la persiana, en caso contrario súbela", corremos el riesgo de obtener una persiana que no pare de subir y bajar si tenemos 25º pero un cielo ligeramente nublado.

En efecto, cada vez que pase una nube, la temperatura descenderá a 24.9º y la persiana se levantará. Una vez que pase la nube, la temperatura volverá a subir y la persiana descenderá. El motor estará permanentemente trabajando. En realidad, nos gustaría decir "Si hace calor fuera, entonces baja la persiana, en caso contrario súbela", pero un ordenador no comprende el término "calor".

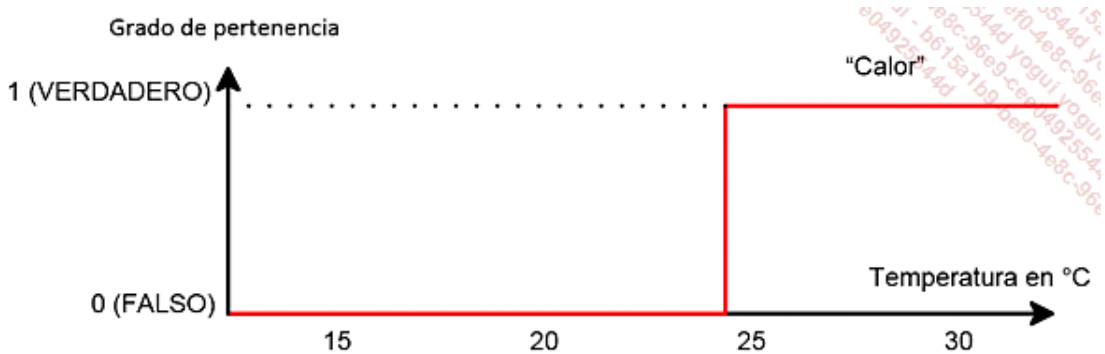
Es precisamente para gestionar este tipo de imprecisión para lo que aparece la **lógica difusa** en 1965. La formalizó Lotfi Zadeh como una extensión de la **lógica booleana** (la lógica "clásica" en la que algo no puede ser más que verdadero o falso). Nos permite definir un enunciado, verdadero, pero cuyos datos (sobre los que se basa) son subjetivos o, en cualquier caso, imprecisos.

Conjuntos difusos y grados de pertenencia

Si retomamos el ejemplo de nuestra persiana eléctrica y la temperatura, nos gustaría definir el término "calor". Vamos a definir, por tanto, para qué temperaturas hace calor o no.

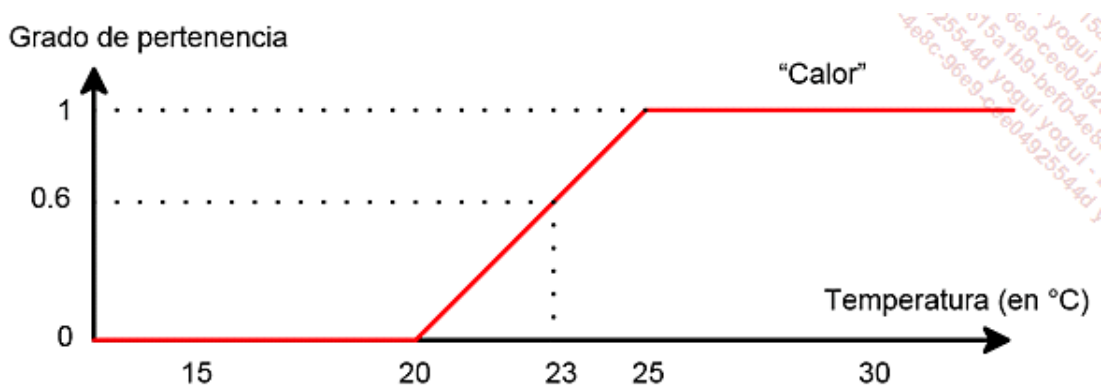
1. Lógica booleana y lógica difusa

En lógica booleana (la lógica clásica), un valor puede ser solamente verdadero o falso. Se debe definir un valor preciso que sirve de transición. Para definir "caliente" en nuestra persiana, este valor es 25º. Por encima de esta temperatura, "calor" es verdadero, y por debajo, "calor" es falso. No existen valores intermedios.



En lógica difusa, utilizaremos un **conjunto difuso**. Se diferencia de un conjunto booleano por la presencia de una "**fase de transición**", durante la cual la variable se sitúa entre los valores verdadero y falso. Para temperaturas entre 20º y 25º, hará más o menos calor.

Por debajo de los 20º, diremos que no hace calor, y por encima de los 25º, diremos que hace calor en un 100 %. Pero, a 23º, no hace calor más que en un 60 % (es decir, un 0.6).



2. Funciones de pertenencia

Vemos, así, como en la lógica booleana se trabaja únicamente con los términos "verdadero" o "falso". Se pasa de falso (0 %) a verdadero (100 %) a 25º. En lógica difusa, se agregan etapas intermedias: entre 20º y 25º se pasa, progresivamente, de falso a verdadero. De esta manera, es posible leer en el gráfico que el **grado de pertenencia** (o valor de verdad) de "calor" para una temperatura de 23º es de 0.6, es decir, un 60 %. De forma similar, a 24º, hace calor en un 80 %.

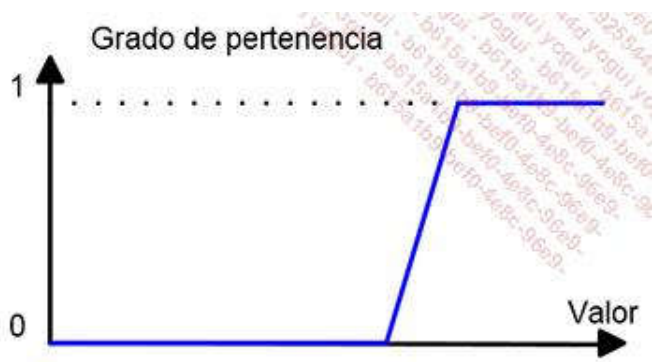
La curva que indica los grados de pertenencia se denomina **función de pertenencia**. Permite definir un **conjunto difuso**, que posee límites que no son claros, sino progresivos, como un fundido. Estos conjuntos difusos pueden

utilizar distintas funciones de pertenencia que asocian todas un grado de pertenencia para los diferentes valores posibles. Existen, no obstante, cinco funciones clásicas.

1. La función triangular: hay un único valor que posee un grado de pertenencia de 1, y se definen dos fases de transición lineales (antes y después de dicho valor).

2. La función trapezoidal: se observa una meseta en la que todos los valores son verdaderos, con dos fases de transición lineales antes y después de dicha meseta.

3. Las funciones 1/2 trapecio (derecha o izquierda): sirven para representar umbrales. Todos los valores situados antes o después de un valor determinado son verdaderos al 100 %. Una fase de transición lineal separa esta meseta de los valores completamente falsos.



Función 1/2 trapecio derecha

Función 1/2 trapecio izquierda

4. La función gaussiana (más conocida como "campana"): parte del principio de la función triangular, eliminando los ángulos, lo que permite trabajar con transiciones todavía más suaves.

5. La función sigmoide: parte de la función 1/2 trapecio, reemplazando los ángulos y las transiciones lineales por una curva más suave.

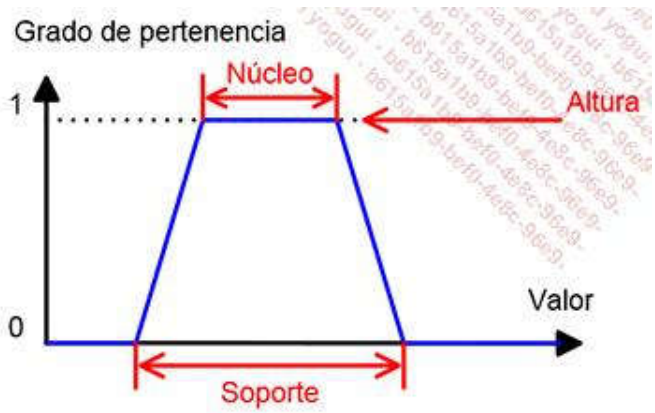
Sea cual sea la función escogida, queda una constante: cada función de pertenencia debe asociarse con un grado comprendido entre 0 y 1 para cada valor potencial del dominio. Matemáticamente, las funciones son continuas en todo el conjunto de la definición.

La forma de las funciones de pertenencia es, sin embargo, libre. Es perfectamente posible utilizar funciones diferentes a las expuestas, pero en la práctica resulta raro hacerlo. En efecto, estas funciones son muy conocidas, así como sus propiedades matemáticas, lo cual simplifica los cálculos.

3. Características de una función de pertenencia

Las distintas funciones de pertenencia se caracterizan por:

- La **altura**: es el grado máximo de pertenencia que se puede obtener. En la gran mayoría de los casos, se considera que las funciones tienen una altura igual a 1. Estas funciones se llaman normalizadas.
- El **soporte**: es el conjunto de valores para los que el grado de pertenencia es mayor que 0. Esto representa, en consecuencia, todos aquellos valores para los que el término es más o menos verdadero (y, por tanto, no es falso).
- El **núcleo**: es el conjunto de valores para los que el grado de pertenencia vale 1. Está, por tanto, incluido en el soporte y se corresponde únicamente con aquellos valores verdaderos al 100 %.



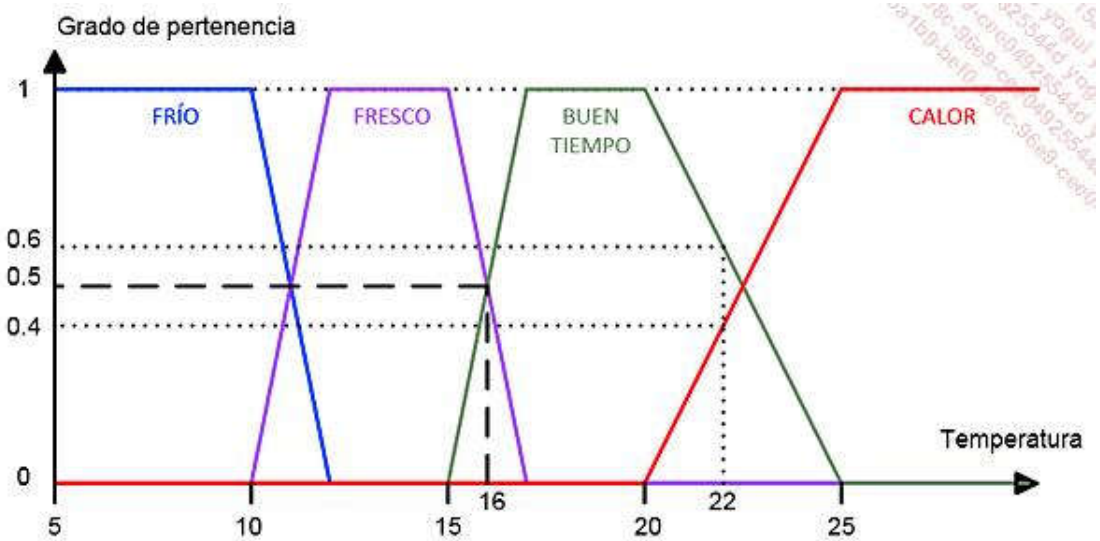
Para el término "calor" definido antes, tenemos una altura de 1 (la función está normalizada), el núcleo es el conjunto [25º; +∞] (todas las temperaturas superiores o iguales a 25º se consideran totalmente verdaderas), y el soporte es el conjunto [20º; +∞] (todas las temperaturas superiores o iguales a 20º son al menos parcialmente verdaderas).

4. Valores y variables lingüísticas

"Calor" se denomina **valor lingüístico**. Se trata, por lo tanto, de un valor que representa un término del lenguaje corriente.

Podríamos imaginar definir varios valores lingüísticos, por ejemplo: "frío", "fresco", "buen tiempo" y "calor". Este conjunto de valores va a definir la "temperatura", que se denominará **variable lingüística**.

Para nuestra variable lingüística "temperatura", podemos obtener el esquema global siguiente, que representa los distintos valores lingüísticos.



En la figura, podemos ver que se han definido cuatro valores lingüísticos: "frío", "fresco", "buen tiempo", "calor".

Es importante comprender cómo se pueden leer los grados de pertenencia de un valor numérico, por ejemplo 16º. En el esquema, vemos con guiones que, para 16º, se cruzan dos curvas: la que representa el valor "fresco" y la que representa el valor "buen tiempo". A 16º, hace fresco y buen tiempo, pero no hace ni frío ni calor. La lectura del gráfico indica que "fresco" y "buen tiempo" son, cada uno, verdaderos al 50 %.

Del mismo modo, para 22º (con un trazo punteado) vemos que no hace ni "frío" ni "fresco", sino que hace "buen tiempo" en un 60 % (o 0.6) y "calor" al 40 %.

En nuestro ejemplo, los valores lingüísticos se definen de manera que la suma de los grados de pertenencia es, siempre, 1 para un valor determinado. Esto no es obligatorio, pero es una buena práctica cuando se define una variable lingüística.

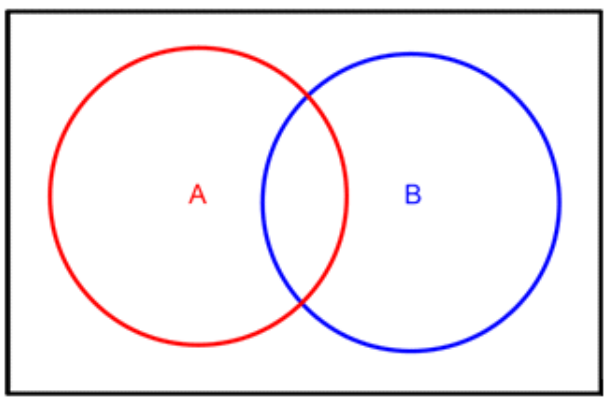
Operadores sobre los conjuntos difusos

En lógica clásica, existen tres operadores de composición elementales: la unión (O), la intersección (Y) y la negación (NO). Estos operadores son también necesarios en lógica difusa, en particular para poder componer valores lingüísticos (por ejemplo, "fresco O bueno") y escribir reglas.

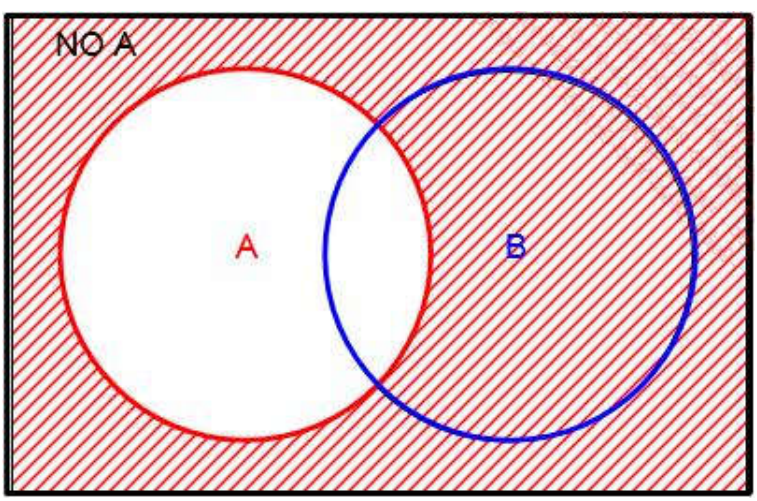
1. Operadores booleanos

En lógica booleana, estos tres operadores pueden representarse gracias a **diagramas de Venn**. En ellos, los conjuntos se representan mediante círculos.

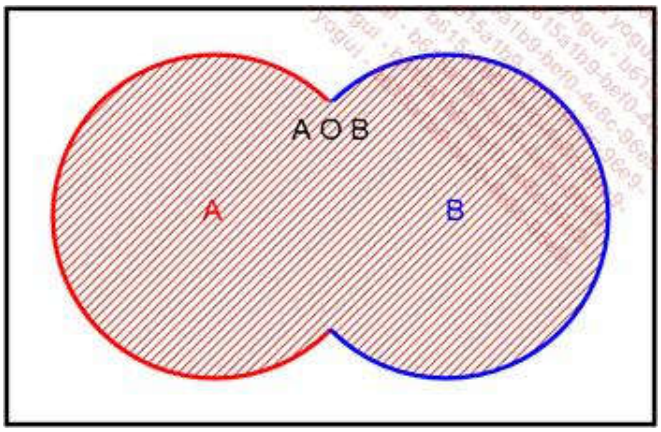
Aquí vemos dos conjuntos, A y B, que se superponen en parte:



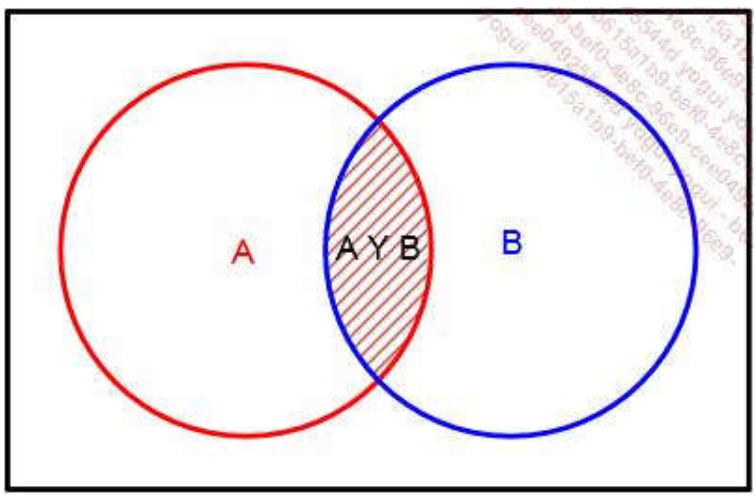
La negación del conjunto A se denomina **NO A** y se escribe $\overline{A}$. Representa el conjunto de valores que no pertenecen a A. Se representa mediante la zona sombreada:



La unión de A y B se denomina **A O B** y se escribe A $\cup$ B. Representa el conjunto de valores que pertenecen a uno u otro conjunto. Aquí se representa mediante la zona sombreada:



Por último, la intersección se lee **A Y B** y se escribe A ∩ B; representa el conjunto de valores que pertenecen a ambos conjuntos al mismo tiempo. Se representa aquí mediante la zona sombreada común a ambos conjuntos:



2. Operadores difusos

En lógica difusa, no hay una transición clara entre lo que es falso y lo que es verdadero. Los operadores clásicos no pueden, por lo tanto, aplicarse y los diagramas de Venn no están bien adaptados.

Es preciso trabajar con las funciones de pertenencia de los conjuntos difusos. En lo sucesivo, esta función se denomina μ_A para el conjunto difuso A. El grado de pertenencia de un valor numérico particular x se escribe entonces $\mu_A(x)$.

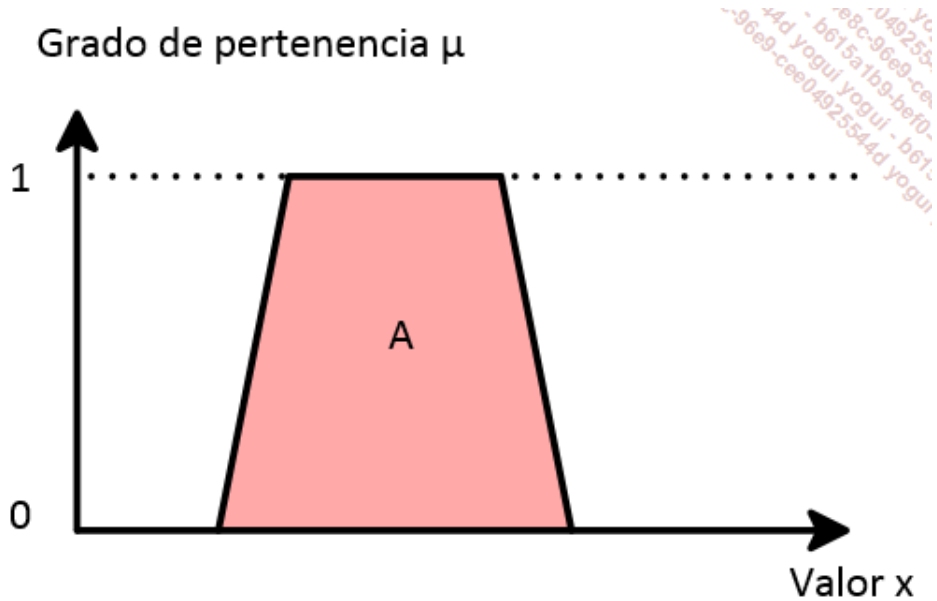
a. Negación

Para la **negación difusa**, NO A se define como el conjunto difuso que tiene como función de pertenencia una función definida por:

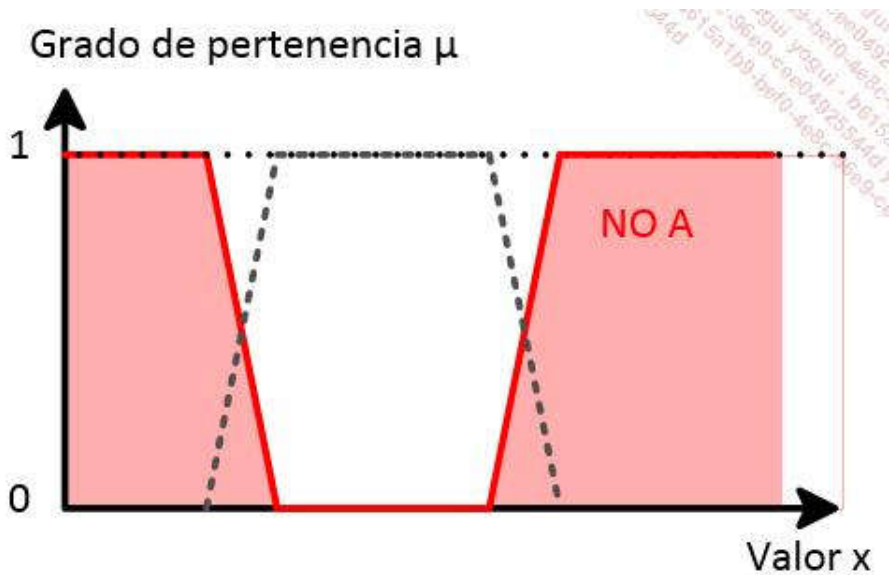
$$\mu_{\bar{A}}(\mathrm{x}) = 1 - \mu_A(x)$$

Esto significa que si 22º se considera como "calor" al 0.4 (es decir, a un 40 %), entonces se considera como "NO calor" a 1-0.4=0.6 (es decir, un 60 %).

Es posible representar gráficamente esta negación. En primer lugar, definamos un conjunto difuso A cuya función de pertenencia sea la siguiente:



El conjunto difuso $\overline{A}$ tiene como función de pertenencia la siguiente función (se recuerda la de A con guiones):

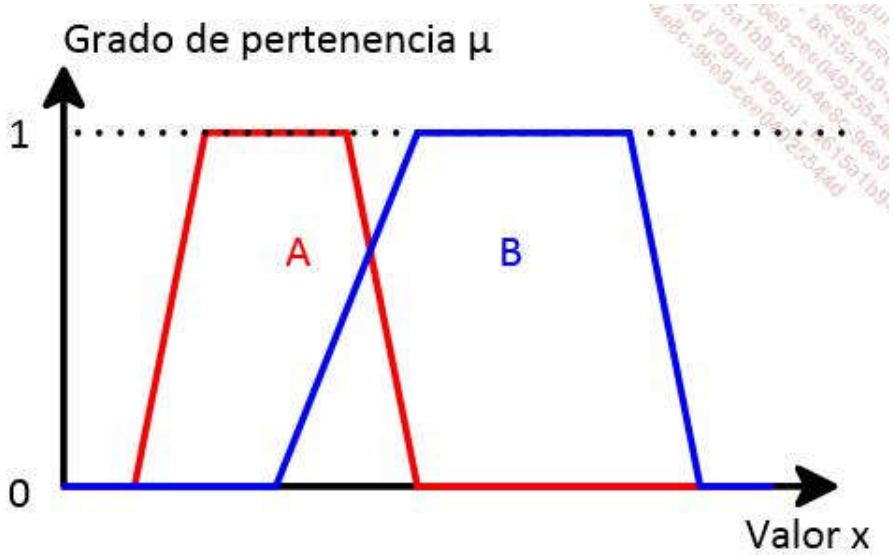


Destacamos que las mesetas están invertidas: una meseta a 1 se encuentra a 0 y viceversa. Además, las transiciones se han invertido.

Geométricamente, podemos percibir que ambas curvas son simétricas una de la otra respecto a un eje de ecuación: grado = 0.5.

b. Unión e intersección

Para estudiar ambos operadores, se definen dos conjuntos difusos A y B, así como sus funciones de pertenencia:

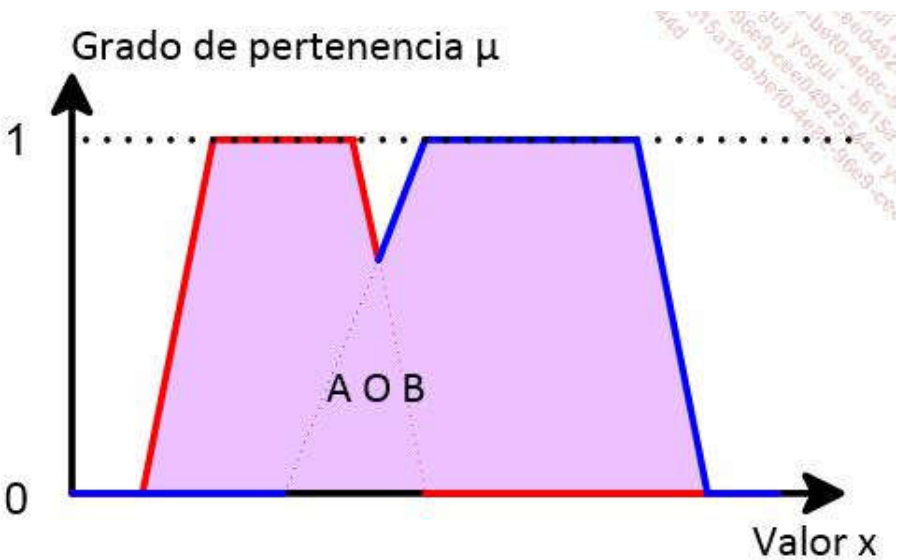


Existen varias formas de calcular la unión o la intersección de estos dos conjuntos. La más habitual (y la más sencilla de implementar) consiste en utilizar los operadores definidos por Zadeh.

La unión A ∪ B se define por la función de pertenencia:

$$\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$$

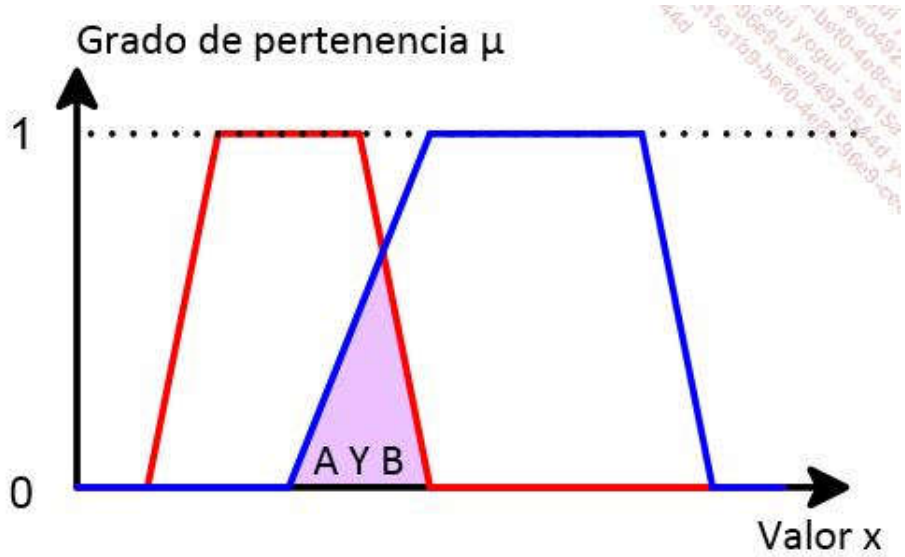
Esto equivale a mantener el nivel de ambas curvas. En efecto, para cada valor, se conserva el valor máximo entre las dos funciones de pertenencia. Se obtiene, por tanto, el siguiente esquema:



Para la intersección A ∩ B, la función de pertenencia se define por:

$$\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$$

Esta vez solo se conserva el nivel común a ambos conjuntos, puesto que para cada valor se conserva la altura mínima para obtener el grado de pertenencia. Esto da como resultado:



Estos operadores son los más próximos a los de la lógica booleana; equivalen a aplicar los operadores clásicos, no sobre diagramas de Venn, sino sobre curvas de funciones de pertenencia.

Otros autores han propuesto operadores de unión y de intersección diferentes. Podemos citar los operadores de Łukasiewicz, que son la variante más utilizada. Su formulación es, sin embargo, más compleja y no se abordará aquí. Tenga en mente que los operadores de Zadeh no son la única opción posible.

Gracias a estos operadores sobre conjuntos, podemos escribir reglas difusas y, más adelante, evaluarlas para tomar decisiones.

Creación de reglas

1. Reglas en lógica booleana

En un sistema clásico, como el control de la persiana del principio de este capítulo, una regla se expresa de la siguiente manera:

```
SI (condición precisa) ENTONCES acción
```

Por ejemplo:

```
SI (temperatura ≥ 25°) ENTONCES bajar la persiana
```

Podemos diseñar reglas más complejas. Por ejemplo, podríamos tener en cuenta la claridad exterior, que se mide en lux (puesto que si hace sol conviene protegerse). Va desde 0 (noche negra sin estrellas ni luna) hasta más de 100 000 (claridad directa del sol). Un cielo nublado de día se corresponde con una claridad entre 200 y 25 000 lux aproximadamente (en función de la densidad de las nubes).

En nuestra aplicación de control de la persiana, podríamos crear la siguiente regla:

```
SI (temperatura ≥ 25° Y claridad ≥ 30 000 lux) ENTONCES bajar la persiana
```

Esto plantea, sin embargo, problemas cuando la temperatura medida o la claridad están cerca de los valores límites.

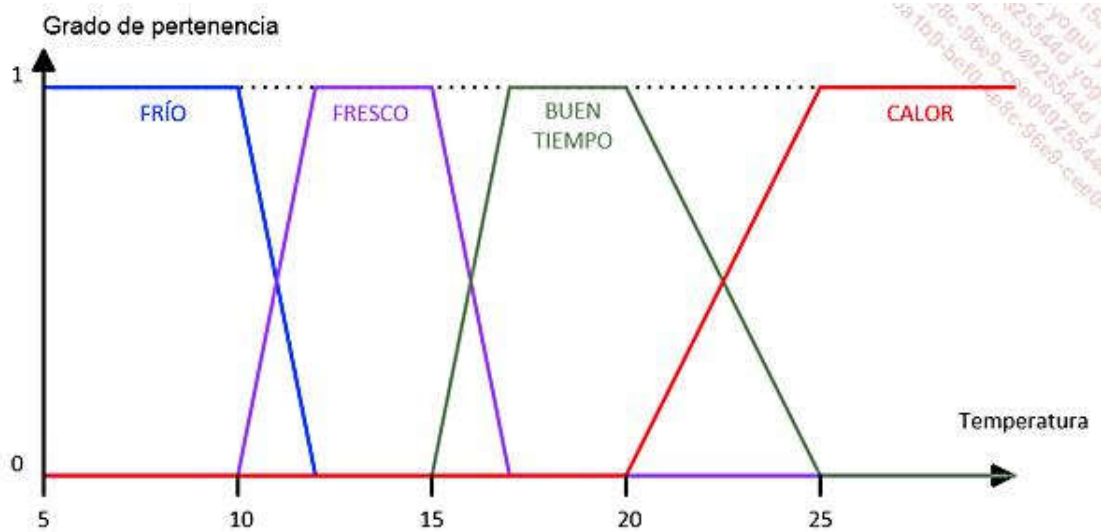
2. Reglas difusas

En un sistema difuso, las reglas utilizan valores difusos en lugar de valores numéricos. La notación de las expresiones utilizadas en las reglas sigue la forma:

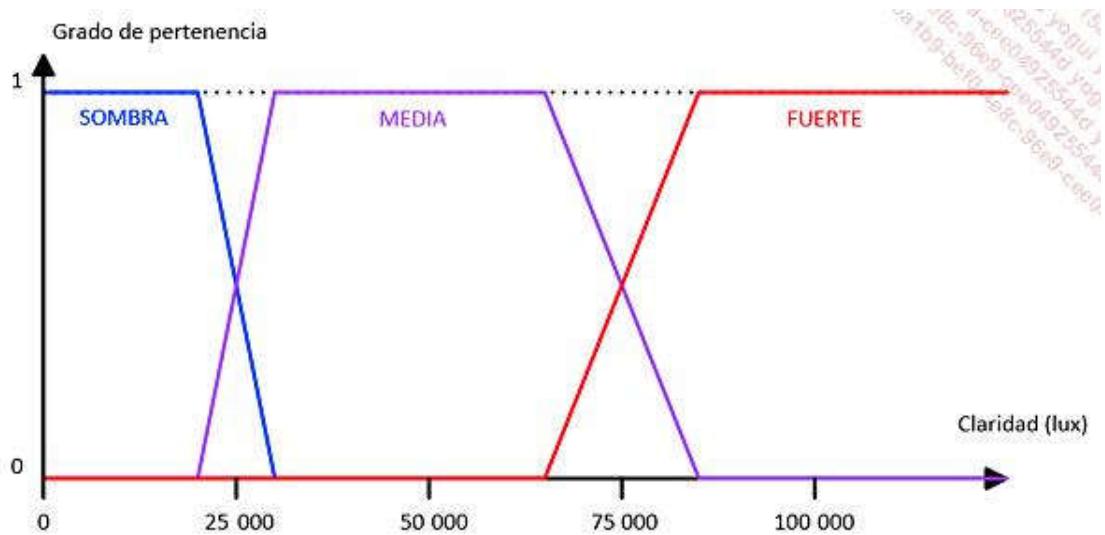
```
"Variable lingüística" ES "valor lingüístico"
```

Vamos a definir tres variables lingüísticas: la temperatura, la claridad y la altura de la persiana.

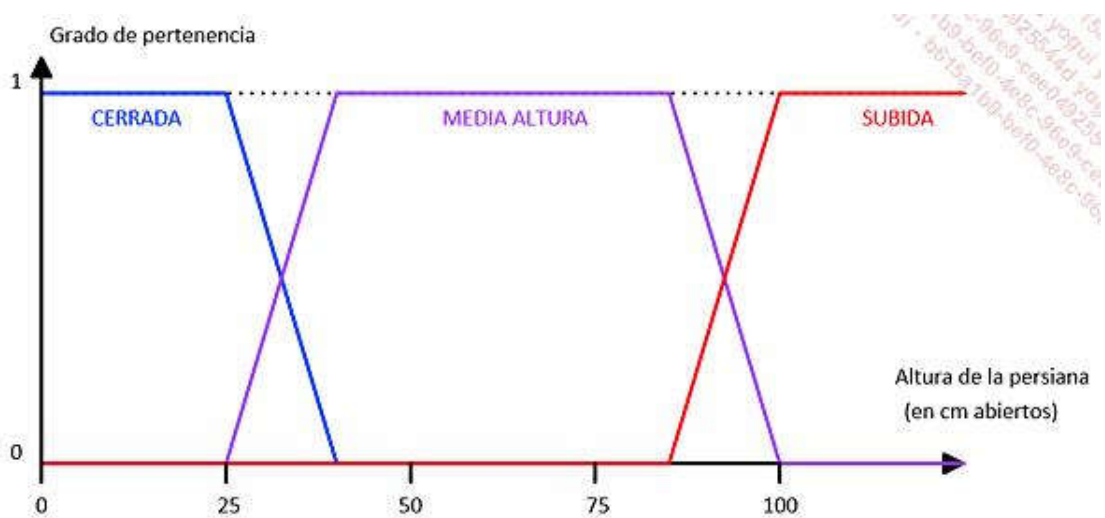
Empezamos con la temperatura. Para ello, recuperamos simplemente el esquema anterior, con temperaturas expresadas en ºC.



La claridad representa la potencia del sol en la ventana. Esta se expresa en lux.



Por último, la altura de la persiana se mide en cm abiertos para una ventana clásica de 115 cm. Para 0, la persiana está completamente bajada, y para 115 está completamente subida.



La temperatura y la claridad son variables lingüísticas de entrada: se miden directamente y son el origen de nuestra decisión. Por el contrario, la altura de la persiana es una variable de salida: es la decisión que hay que tomar.

He aquí un ejemplo de regla que podríamos definir:

```
SI (temperatura ES calor Y claridad ES fuerte) ENTONCES altura de la persiana ES cerrada
```

Evidentemente, un sistema difuso puede poseer varias reglas. Lo más sencillo para representarlas cuando se reciben dos variables lingüísticas como entrada es una tabla de doble entrada, que indica para cada combinación de valores el valor de la variable de salida (en nuestro caso, la altura de la persiana).

La siguiente tabla indica, por lo tanto, las 12 reglas que se han numerado, por simplicidad, de R1 a R12. Cada una de ellas se corresponde con un caso de temperatura y de claridad, e indica la decisión que hay que tomar. Por ejemplo, si hace frío con una fuerte claridad, entonces se subirá la persiana para intentar aprovechar al máximo los rayos del sol (regla R3).

Claridad → **Temperatura ↓**	**Sombra**	**Media**	**Fuerte**
Frío	R1. Subida	R2. Subida	R3. Subida
Fresco	R4. Subida	R5. Subida	R6. Media altura
Buen tiempo	R7. Subida	R8. Media altura	R9. Bajada
Calor	R10. Subida	R11. Media-altura	R12. Bajada

Nuestras reglas van a intentar maximizar el confort del usuario, sabiendo que, si hace demasiado calor y el sol es fuerte, es mejor estar a la sombra, mientras que si hace frío, conviene aprovechar el sol si está presente.

Fuzzificación y defuzzificación

1. Valor de verdad

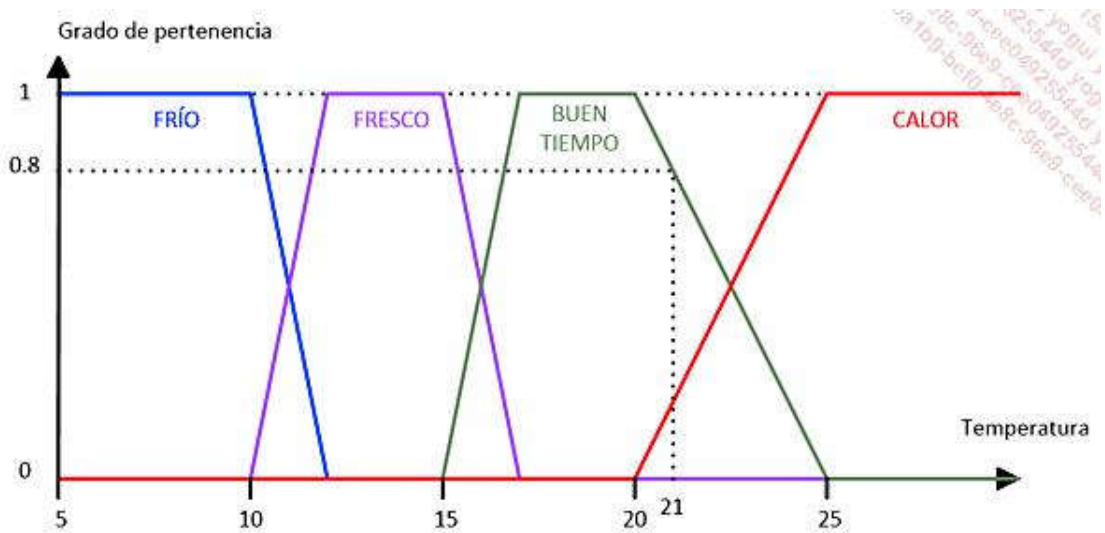
Las distintas reglas poseen, todas, una implicación (la cláusula ENTONCES). Será preciso expresar hasta qué punto la regla debe aplicarse, en función de los valores numéricos medidos: es la etapa de **fuzzificación.**

Nos centraremos en la regla R8:

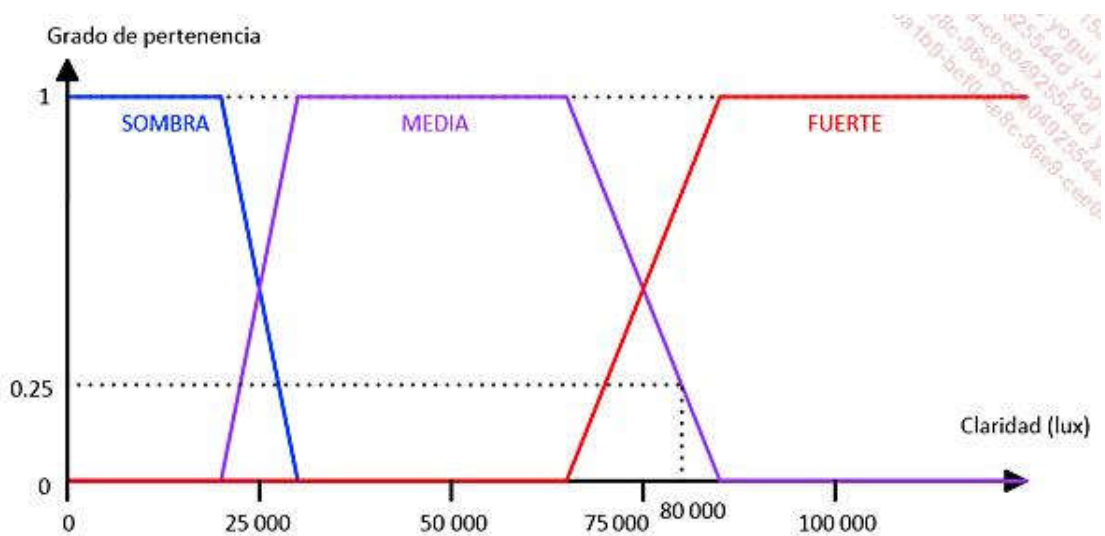
```
SI temperatura ES buen tiempo Y claridad ES media ENTONCES persiana ES
a media altura
```

Queremos saber hasta qué punto se aplica esta regla para una temperatura de 21 ºC y una claridad de 80 000 lux.

Buscaremos, en primer lugar, hasta qué punto hace "BUEN TIEMPO". La siguiente figura nos indica que a 21 ºC hace buen tiempo al 80 %.



Buscaremos, a continuación, hasta qué punto la claridad es MEDIA para 80 000 lux. Podemos comprobar que está al 25 %:



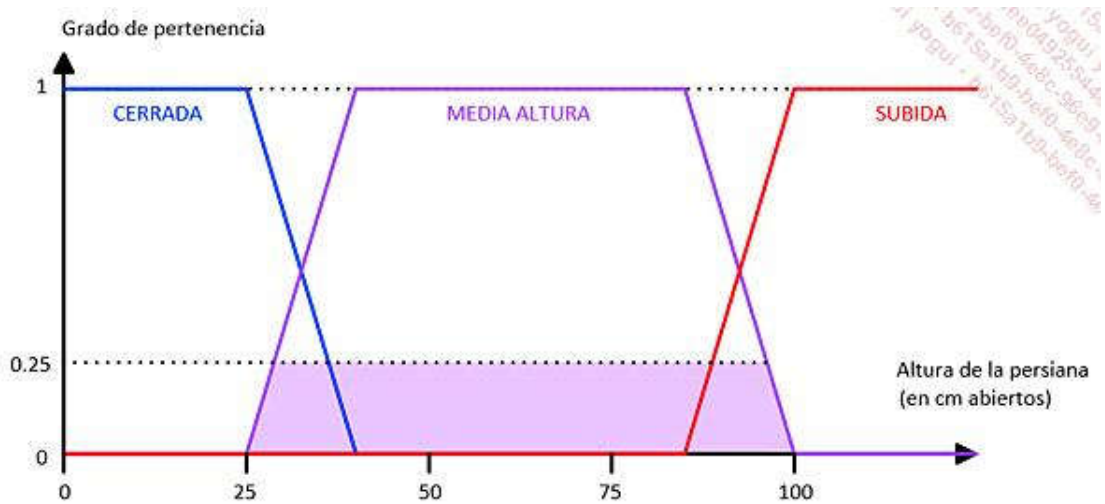
La regla contiene, por lo tanto, una parte verdadera al 80 % y una parte verdadera al 25 %. Diremos que la regla entera es verdadera al 25 %, el valor mínimo (operador Y). Es el término menos verdadero el que determina el valor de la verdad de una regla completa.

2. Fuzzificación y aplicación de las reglas

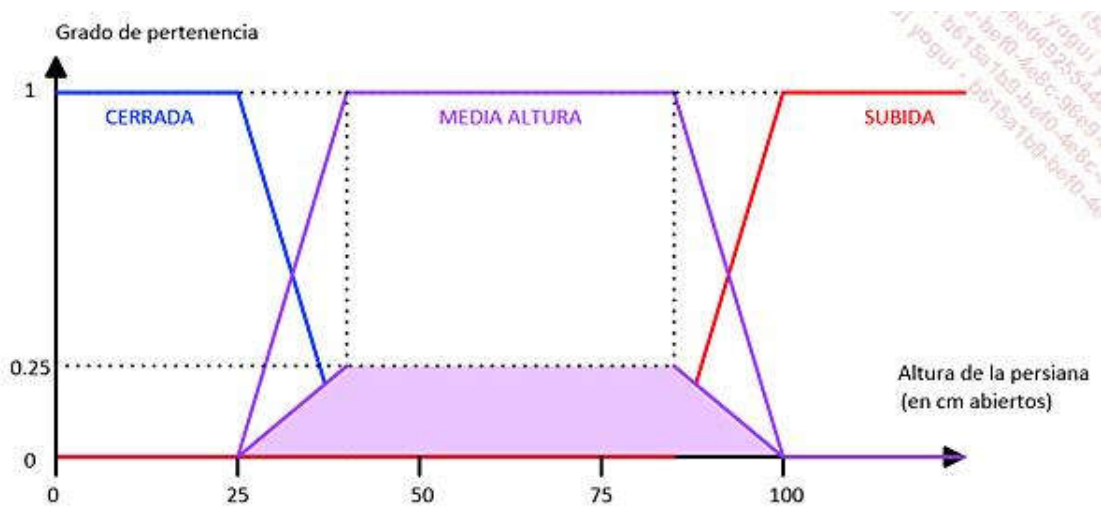
Buscaremos ahora saber cuál será el resultado de esta regla. Nos dice que la persiana debe estar a media altura. Sabemos que la regla se aplica en un 25 %.

Tenemos muchas opciones para el operador de implicación a fin de determinar el conjunto difuso resultante. Nos interesaremos en dos de ellos: la implicación de Mamdani y la de Larsen. Lo que cambia entre ambas es la forma del conjunto difuso obtenido.

Para el operador de implicación de Mamdani, el conjunto difuso se trunca al valor de verdad de la regla. Para nuestra regla R8, que se aplica en un 25 %, obtendríamos la siguiente salida.



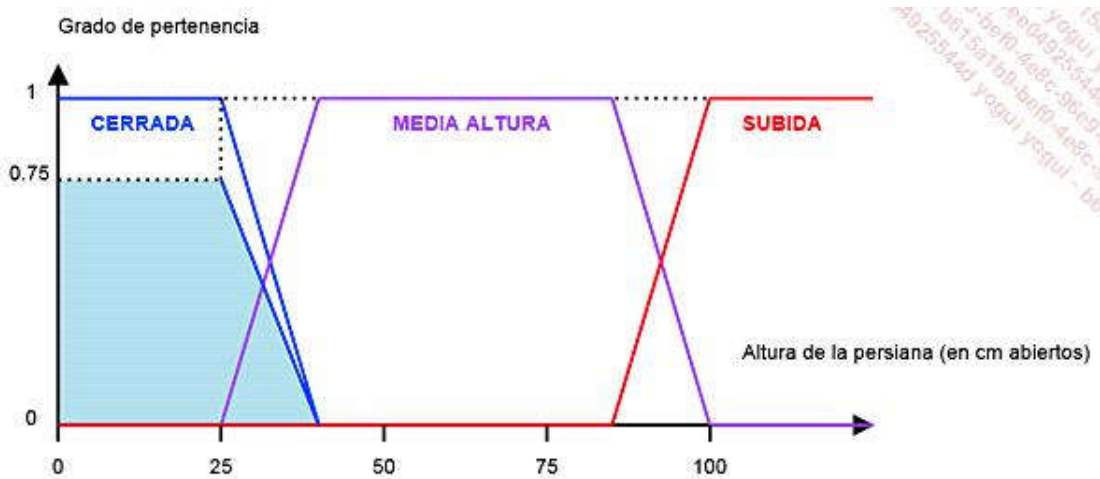
Para el operador de implicación de Larsen, la forma global de la función de pertenencia se reduce, a fin de limitar el grado de verdad de la regla. Esto equivale a multiplicar todos los valores de la función por el grado correspondiente. En nuestro caso, multiplicaremos la función por 0.25, conservando así la forma trapezoidal, con la misma meseta.



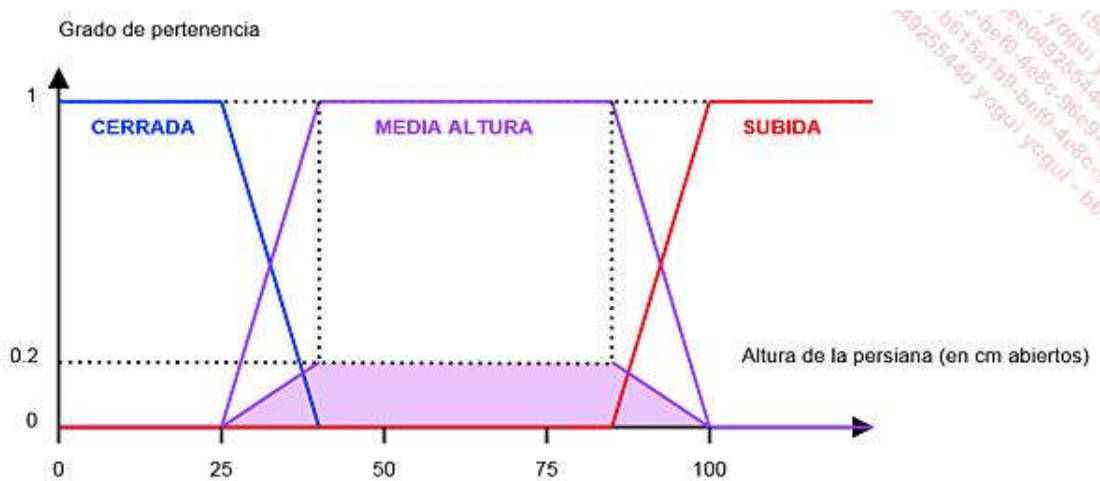
En lo sucesivo, utilizaremos la implicación de Larsen. No existe, sin embargo, una solución mejor que otra en términos absolutos, sino que dependerá de cada tipo de problema y de las preferencias de cada uno. El operador de Larsen se mantiene en este caso, puesto que es más rápido de calcular cuando se trabaja con conjuntos difusos, dado que no se trata más que de una multiplicación de los grados de pertenencia, mientras que para Mamdani es preciso calcular las nuevas coordenadas de los puntos límites de la meseta.

Basándonos en una temperatura de 21 ºC y una claridad de 80 000 lux, en realidad tenemos cuatro reglas que pueden aplicarse: R8, R9, R11 y R12. Ya hemos visto qué resultado nos daba el operador de implicación como conjunto difuso para la regla R8.

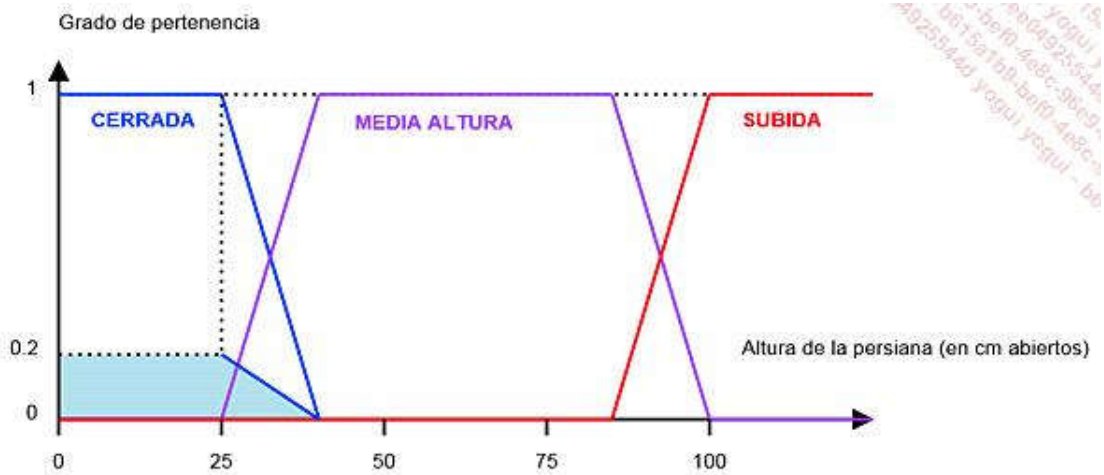
A continuación se debe seguir el mismo razonamiento para R9: si la temperatura es buena (lo cual es verdadero al 80 % a 21 ºC) y la claridad fuerte (verdadero al 75 %), entonces la persiana debe bajarse. Se obtiene el siguiente conjunto difuso, de altura 0.75:



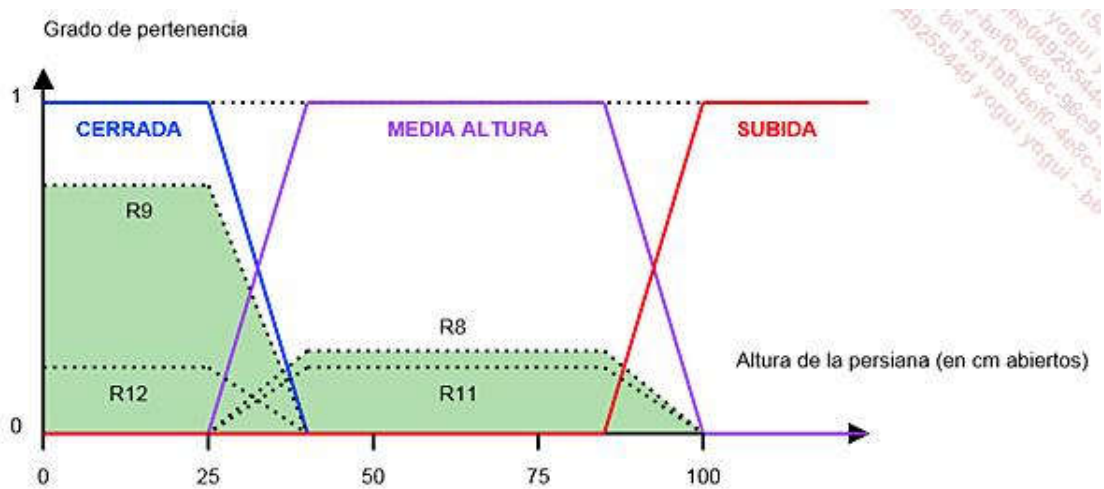
La regla R11 nos dice que, si la temperatura es calor (verdadero al 20 %) y la claridad media (verdadero al 25 %), entonces la persiana debe estar a media altura (en nuestro caso a un 20 %).



Por último, para R12, buscamos una temperatura calurosa (verdadero al 20 % a 21 ºC) y una claridad fuerte (verdadero al 75 %). La regla que pide una persiana bajada se aplica, por tanto, al 20 %.



Estos conjuntos difusos se componen entre sí mediante el operador de unión para obtener la salida difusa de nuestro sistema. Obtenemos, por tanto, el conjunto final siguiente (recuerde que la unión consiste en considerar la altura máxima obtenida).



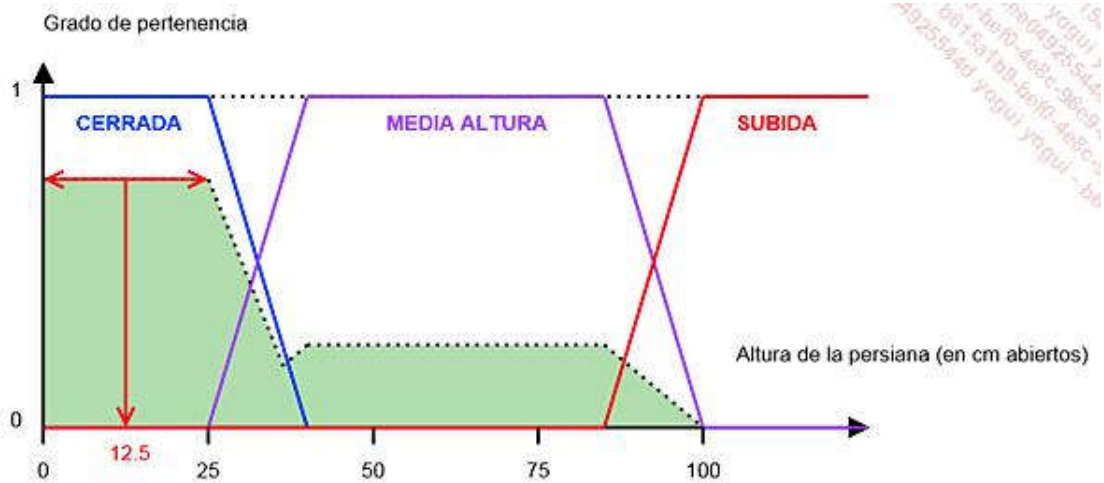
3. Defuzzificación

Una vez calculado el conjunto difuso resultante, se debe tomar una decisión que es un valor numérico único y no un conjunto difuso: es la etapa de **defuzzificación.** En efecto, los controles (motores, válvulas, controladores, frenos...) piden una orden que puedan ejecutar.

En el caso de la persiana, es preciso saber si subirla o bajarla. En efecto, el motor de la persiana necesita un valor único que le indique la altura que debe aplicar.

Incluso en este caso, existen varias soluciones. Vamos a ver dos: la defuzzificación por cálculo de la media y por baricentro.

La defuzzificación por cálculo de la media, que es la más sencilla, consiste en calcular la media de la meseta más elevada. Aquí, tenemos una meseta de 0 a 25 cm gracias a la regla R9. La media es, por tanto, de 12.5 cm: la persiana no dejará más que 12.5 cm abiertos (estará, por lo tanto, casi cerrada, dejando pasar un pequeño hilo de luz). Esto se corresponde con lo que queremos: cuando hace bastante calor dentro y el sol es fuerte, conviene bajar la persiana.

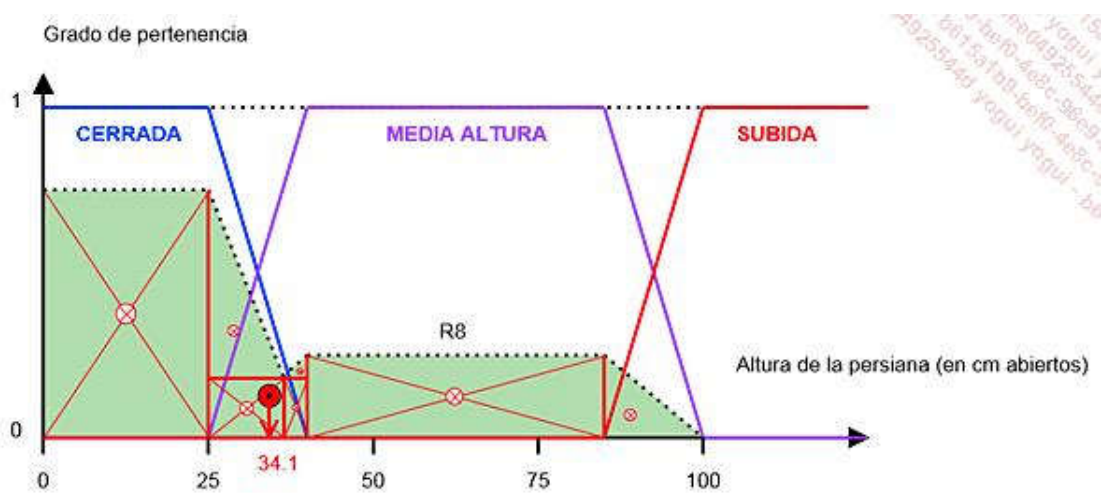


La defuzzificación por baricentro, más compleja, consiste en buscar el baricentro (también llamado centroide o, de manera abusiva, centro de gravedad) de la forma obtenida. Esto permite tener en cuenta el conjunto de las reglas, y no solamente la regla mayoritaria, como ocurría en el caso anterior (solamente la regla R9 participaba de la decisión final).

Si se recorta la forma obtenida en una cartulina, el baricentro se corresponde con el punto que permite mantener la forma en equilibrio sobre la punta de un lápiz. Si bien comprender el sentido del baricentro es algo muy sencillo, resulta sin embargo mucho más difícil de calcular. Es preciso, de hecho, calcular la media ponderada de las diferentes formas (triángulos o rectángulos) que componen la forma global, e incluso realizar integrales si la forma no es un polígono.

En nuestro caso, podemos descomponer la forma en formas más sencillas. Para cada subforma es posible, a continuación, encontrar el baricentro: está situado en la mitad para los rectángulos (donde se cruzan las diagonales) y a 1/3 en los triángulos rectángulos.

Empezaremos dividiendo el conjunto obtenido en pequeñas formas más sencillas (rectángulos o triángulos rectángulos). Asociaremos, a continuación, a cada forma su baricentro (puntos blancos). Cada uno se ponderará según el área de la forma (gráficamente, los puntos son más gruesos en las formas más grandes). Por último, realizaremos la media ponderada para obtener el punto rojo, que es el baricentro global de nuestra figura:



Se obtiene una altura para la persiana de 34.1 cm, que se corresponde con una persiana prácticamente cerrada, pero como no hace demasiado calor, se puede aprovechar un poco de claridad.

La segunda solución es, por tanto, más precisa, pero es más compleja de calcular, sobre todo si las funciones de

pertenencia no son lineales a trozos. Con la capacidad actual de los ordenadores, es cada vez más fácil y rápido calcularla, y no presenta ningún problema de rendimiento.

Existen otros métodos para realizar la defuzzificación, menos utilizados, que no abordaremos. Tienen en común, sin embargo, la capacidad de determinar un valor único a partir de un conjunto difuso.

Dominios de aplicación

Zadeh expuso las bases teóricas de la lógica difusa en 1965. Los países occidentales no se interesaron realmente por esta técnica en sus inicios. Por el contrario, Japón comprendió rápidamente su interés, seguido algunos años más tarde por el resto del mundo.

1. Primer uso

En 1987, el primer tren controlador por un sistema basado en reglas difusas apareció en Sendai, una ciudad a menos de 400 km al norte de Tokio. Los ingenieros quisieron maximizar el confort de los viajeros minimizando el consumo de energía, de modo que el vehículo debía realizar numerosos cambios de velocidad. La velocidad del tren es, por tanto, resultado de la lógica difusa.

Se ha demostrado que el consumo de energía disminuyó en un 10 % respecto a un conductor humano, y que los pasajeros elogiaban la suavidad en la conducción, principalmente en los arranques y las paradas.

Está todavía en circulación y ha supuesto el primer gran éxito comercial de la lógica difusa.

2. En los productos electrónicos

Muchos otros constructores comprendieron que un controlador difuso podía mejorar el funcionamiento de las máquinas que lo contenían.

Es así como actualmente estamos envueltos de lógica difusa, sin ni siquiera saberlo: encontramos ejemplos en las lavadoras de la marca LG (para seleccionar el tiempo y la potencia ideal en función del contenido), las secadoras de la marca Whirlpool o los cocedores de arroz de la marca Panasonic.

Actualmente, cada vez vemos más termostatos inteligentes (conectados o no), que permiten conservar una temperatura agradable. La mayoría de ellos utilizan un controlador difuso para realizar esta tarea. También se aplica a los sistemas de calefacción/climatización, más complejos porque cubren potencialmente toda la casa.

Otro uso al que todos estamos acostumbrados, pero al que no prestamos atención, es la luminosidad automática de nuestro teléfono y otras pantallas. En efecto, también aquí en función de la luminosidad exterior, la posición del objeto, de las acciones que se realicen u otra información, la luminosidad de la pantalla se adapta.

3. En el mundo del automóvil

La lógica difusa no solo está presente en nuestros electrodomésticos, sino que la encontramos también en nuestros coches. De hecho, actualmente la gran mayoría de los conductores utilizan un controlador difuso para el ABS (Antiblockiersystem, o sistema de antibloqueo de ruedas), con Nissan y Mitsubishi a la cabeza. Este sistema permite asegurar que la frenada es eficaz, en función del estado o del tipo de firme.

Encontramos reglas difusas, también, en los coches de inyección electrónica para decidir la cantidad de carburante que deben utilizar, en las cajas de cambios automáticas para seleccionar la marcha, en ciertas transmisiones, en los sistemas ESP (que permiten evitar o limitar los derrapes y las pérdidas de control) o incluso en los limitadores de velocidad.

4. Otros dominios

La lógica difusa se utiliza con frecuencia en los **sistemas industriales**, para decidir la apertura de válvulas, el control

de una producción, la potencia de los compresores, el funcionamiento de los convertidores, la carga y las pruebas sobre las baterías... Estos controladores difusos permiten ahorrar energía o tiempo de vida del mecanismo, gracias a un funcionamiento más suave, sin golpes bruscos.

La **robótica** es otro gran dominio de esta técnica, para permitir a los robots tener un comportamiento mejor adaptado y, sobre todo, más fácil de comprender para los humanos que interactúan con ellos, lo cual resulta primordial para los robots de compañía.

Los programas informáticos como los **videojuegos** utilizan a menudo la lógica difusa. Este es el caso si el comportamiento y el desplazamiento de los enemigos/amigos está gobernado por reglas difusas, que permiten una mejor inmersión del jugador en el juego. *Halo*, *Thief - Deadly shadows*, *Unreal*, *Battle Cruiser: 3000AD*, *Civilization*: *Call to power*, *Close Combat* o incluso *The Sims* son, de este modo, grandes juegos que utilizan esta técnica.

Por último, la lógica difusa se utiliza cada vez más en aplicaciones de **procesamiento de imagen**, permitiendo mejorar los algoritmos existentes para ayudar a clasificar los colores, reconocer formas o extraer información de la imagen, como el avance de una enfermedad sobre la hoja de un árbol o en agricultura razonada.

La lógica difusa es, por tanto, una técnica muy simple en su principio, fácil de implementar, y de la que existen numerosas derivaciones, con aplicaciones muy abundantes.

Implementación de un motor de lógica difusa

Esta sección describe cómo codificar un motor de lógica difusa, utilizando las premisas expuestas hasta el momento. El siguiente código está escrito en Java, pero podría adaptarse fácilmente a cualquier otro lenguaje orientado a objetos. Utiliza Java 10, aunque no necesita librerías externas para una mejor portabilidad.

Cuando se requieran conocimientos matemáticos, se explicarán las fórmulas utilizadas.

1. El núcleo del código: los conjuntos difusos

a. Punto2D: un punto de una función de pertenencia

Vamos a comenzar creando las clases básicas. Para ello, necesitaremos una clase **`Punto2D`** que nos permita definir las coordenadas de un punto representativo de las funciones de pertenencia. El eje de abscisas (x) representa el valor numérico y el eje de ordenadas (y) el valor de pertenencia correspondiente, entre 0 y 1.

La base de esta clase es la siguiente:

```
public class Punto2D {
  // Coordenadas
  public double x;
  public double y;

  // Constructor
  public Punto2D(double _x, double _y) {
    x = _x;
    y = _y;
  }
}
```

Más adelante, habrá que comparar dos puntos para conocer su orden. En lugar de comparar nosotros mismos las coordenadas x de los puntos, vamos a implementar la interfaz `Comparable` en esta clase. Modificaremos el encabezado de la siguiente manera:

```
public class Punto2D implements Comparable
```

A continuación, es preciso agregar el método `compareTo`, que permite saber si el punto que se pasa como parámetro es más pequeño, igual o más grande en relación con el objeto en curso (el método debe devolver, respectivamente, un número positivo, cero o negativo). Nos contentaremos con calcular la diferencia de las abscisas:

```
  @Override
  public int compareTo(Object t) {
    return (int) (x - ((Punto2D) t).x);
  }
```

Por último, el método `toString()` permite facilitar la visualización:

```
  @Override
  public String toString() {
    return "(" + x + ";" + y + ")";
  }
```

b. ConjuntoDifuso: un conjunto difuso

La principal clase de nuestro programa (**`ConjuntoDifuso`**), tanto en líneas de código como en importancia, es el conjunto difuso. Está formado por una lista de puntos que estarán ordenados según el eje x y por valores particulares: el mínimo y el máximo que podrán tomar los valores numéricos.

Esta clase contiene, entonces, tres atributos:

- `puntos`: la lista de puntos que componen la función de pertenencia.
- `min`: el valor mínimo posible.
- `max`: el valor máximo.

Además, se agrega un constructor que permite inicializar la lista, y dos métodos que permiten agregar puntos a la lista (que está ordenada): el primero recibe como parámetro un `Punto2D`, y el segundo, dos coordenadas.

El código base de esta clase es, entonces:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.StringJoiner;

public class ConjuntoDifuso {
    protected ArrayList<Punto2D> puntos;
    protected double min;
    protected double max;

    // Constructor
    public ConjuntoDifuso(double _min, double _max) {
        puntos = new ArrayList();
        min = _min;
        max = _max;
    }

    // Agregar un punto
    public void Agregar(Punto2D pt) {
        puntos.add(pt);
        Collections.sort(puntos);
    }
    public void Agregar(double x, double y) {
        Punto2D pt = new Punto2D(x,y);
        Agregar(pt);
    }
}
```

Esta clase posee, también, un método `toString()`, que muestra el intervalo de valores y, a continuación, los distintos puntos registrados en la lista:

```
@Override
public String toString() {
    StringJoiner sj = new StringJoiner(" ");
    sj.add("[" + min + "-" + max + "]:");
    for (Punto2D pt : puntos) {
         sj.add(pt.toString());
    }
    return sj.toString();
}
```

c. Operadores de comparación y de multiplicación

El operador de comparación **`equals`** permite saber si dos conjuntos difusos son iguales o no. Para ello, nos basaremos en el método `toString()`: dos conjuntos difusos serán idénticos si producen la misma cadena (dado que la cadena tiene en cuenta todos los atributos). Esto evita tener que comparar los puntos uno a uno.

```
// Operador de comparación (se comparan las cadenas resultantes)
@Override
public boolean equals(Object pt2) {
    return toString().equals(((Punto2D)pt2).toString());
}
```

Agregamos, por último, la multiplicación por un número `MultiplicarPor()`. Basta con multiplicar todas las ordenadas de los puntos por el valor pasado como parámetro y agregarlos en un nuevo conjunto difuso que será devuelto. Esto resultará muy útil en la aplicación de reglas difusas.

```
// Operador de multiplicación
public ConjuntoDifuso MultiplicarPor(double valor) {
    ConjuntoDifuso conjunto = new ConjuntoDifuso(min, max);
    for(Punto2D pt : puntos) {
        conjunto.Agregar(pt.x, pt.y * valor);
    }
    return conjunto;
}
```

d. Operadores sobre conjuntos

El primer operador sobre conjuntos y el más sencillo de codificar es el operador **NOT**. Crearemos un nuevo conjunto difuso sobre el mismo intervalo y agregaremos, para cada punto del conjunto de partida, un punto de altura 1-y.

```
// Operador NOT (negación)
public ConjuntoDifuso No() {
    ConjuntoDifuso conjunto = new ConjuntoDifuso(min, max);
    for (Punto2D pt : puntos) {
```

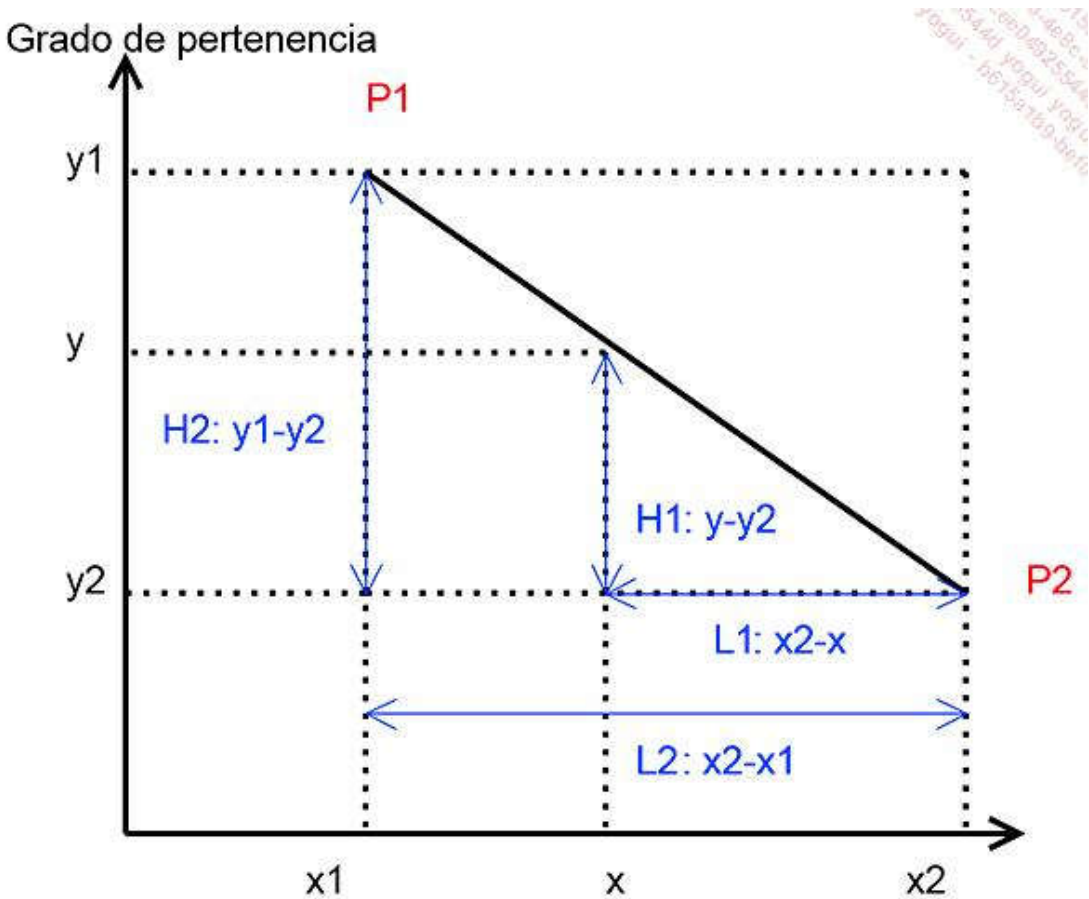
```
            conjunto.Agregar(pt.x, 1 - pt.y);
        }
        return conjunto;
    }
```

El resto de los métodos son algo más complejos de codificar. En primer lugar, hace falta un método que devuelva el grado de pertenencia a partir de un valor numérico. Es necesario para realizar la fuzzificación (que consiste en transformar los valores numéricos medidos en grados de pertenencia para los distintos valores lingüísticos), pero también va a servirnos para la unión y la intersección de conjuntos difusos.

Para determinar el grado de pertenencia de un valor numérico, pueden darse tres casos:

1. El valor está fuera del intervalo para este conjunto difuso: el grado de pertenencia es nulo.
2. Se ha definido un punto para este valor en el conjunto difuso: en este caso, basta con devolver el grado registrado.
3. No se ha definido ningún punto para este valor: será preciso interpolar el grado de pertenencia. Para ello, nos hace falta el punto inmediatamente anterior y el punto inmediatamente posterior.

En el siguiente esquema se han situado los puntos anterior (P1) y posterior (P2). Se definen, respectivamente, mediante las coordenadas (x1, y1) y (x2, y2). Buscaremos el grado y del punto de valor x.



Utilizaremos el teorema de Thales. Como las alturas H1 y H2 son paralelas, entonces sabemos que:

$$\frac{L1}{L2} = \frac{H1}{H2}$$

Reemplazando cada distancia por su expresión literal, tenemos:

$$\frac{x2 - x}{x2 - x1} = \frac{y - y2}{y1 - y2}$$

Es decir:

$$(x2 - x) * (y1 - y2) = (y - y2) * (x2 - x1)$$

$$\frac{(y1 - y2) * (x2 - x)}{x2 - x1} = y - y2$$

Podemos deducir que:

$$y = \frac{(y1 - y2) * (x2 - x)}{x2 - x1} + y2$$

Y obtenemos el siguiente código:

```
    // Cálculo del grado de pertenencia de un punto
    public double ValorDePertenencia(double valor) {
        // Caso 1: al exterior del intervalo del conjunto difuso
        if (valor < min || valor > max || puntos.size() < 2) {
            return 0;
        }

        Punto2D ptAntes = puntos.get(0);
        Punto2D ptDespues = puntos.get(1);
        int index = 0;
        while(valor >= ptDespues.x) {
            index++;
            ptAntes = ptDespues;
            ptDespues = puntos.get(index);
        }

        if (ptAntes.x == valor) {
            // Tenemos un punto en este valor
            return ptAntes.y;
        }
        else {
            // Se aplica la interpolación
            return ((ptAntes.y - ptDespues.y) * (ptDespues.x -
valor) / (ptDespues.x - ptAntes.x) + ptDespues.y);
        }
```

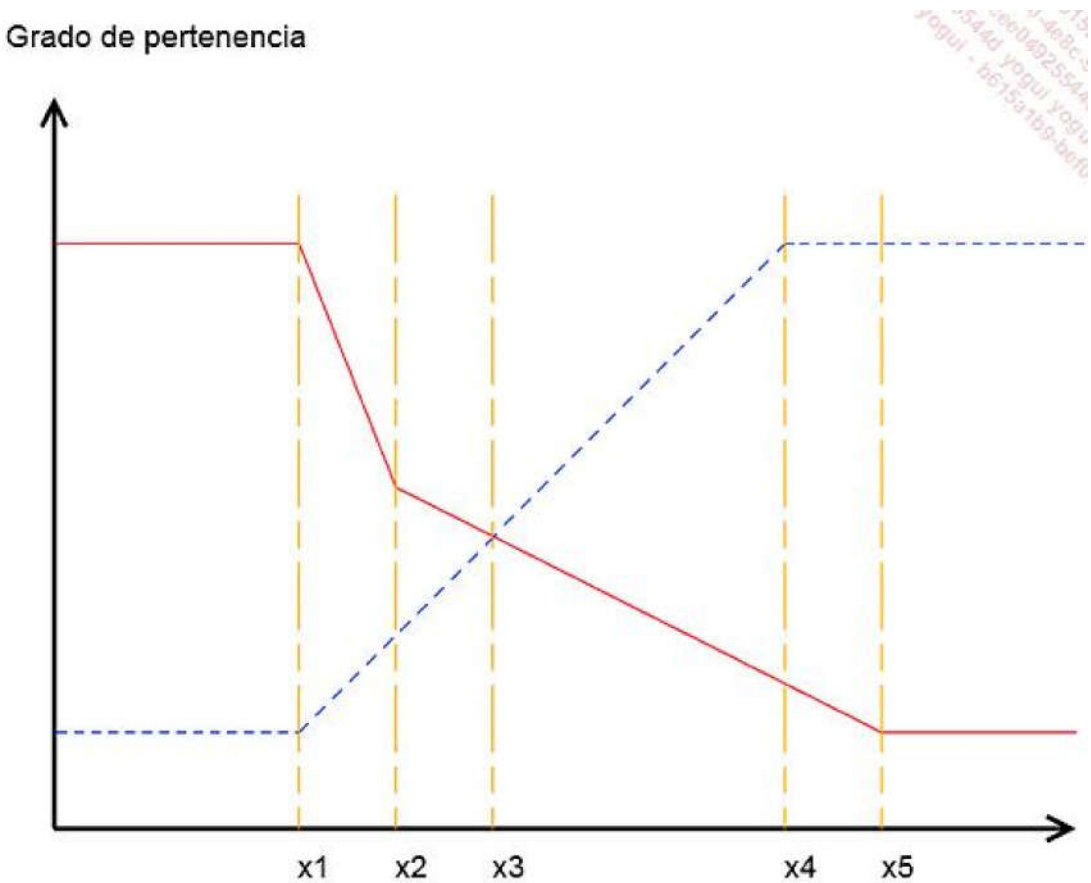
```
    }
```

Los dos métodos siguientes permiten calcular la intersección y la unión de conjuntos difusos, es decir, los operadores & (Y) y | (O). En ambos casos, se tratará de operadores de Zadeh. Recordemos que se trata de utilizar el valor mínimo para la intersección y el valor máximo para la unión.

La dificultad se plantea sobre todo en los recorridos de ambos conjuntos difusos. En efecto, no poseerán, necesariamente, puntos en las mismas abscisas. Será necesario recorrer los puntos de ambas colecciones en paralelo. Aun así, pueden darse varios casos:

1. Los dos conjuntos poseen un punto en la misma abscisa: basta con calcular el grado de pertenencia correcto (`min` o `max` según el método).
2. Un solo conjunto posee un punto en una abscisa determinada: es preciso calcular (gracias al método anterior) el grado de pertenencia para el segundo conjunto y guardar el valor correcto.
3. Ninguno de los conjuntos posee el punto, pero ambas curvas se cruzan, en cuyo caso debemos crear un punto en la intersección. La dificultad estriba, sobre todo, en detectar y calcular estos puntos de intersección.

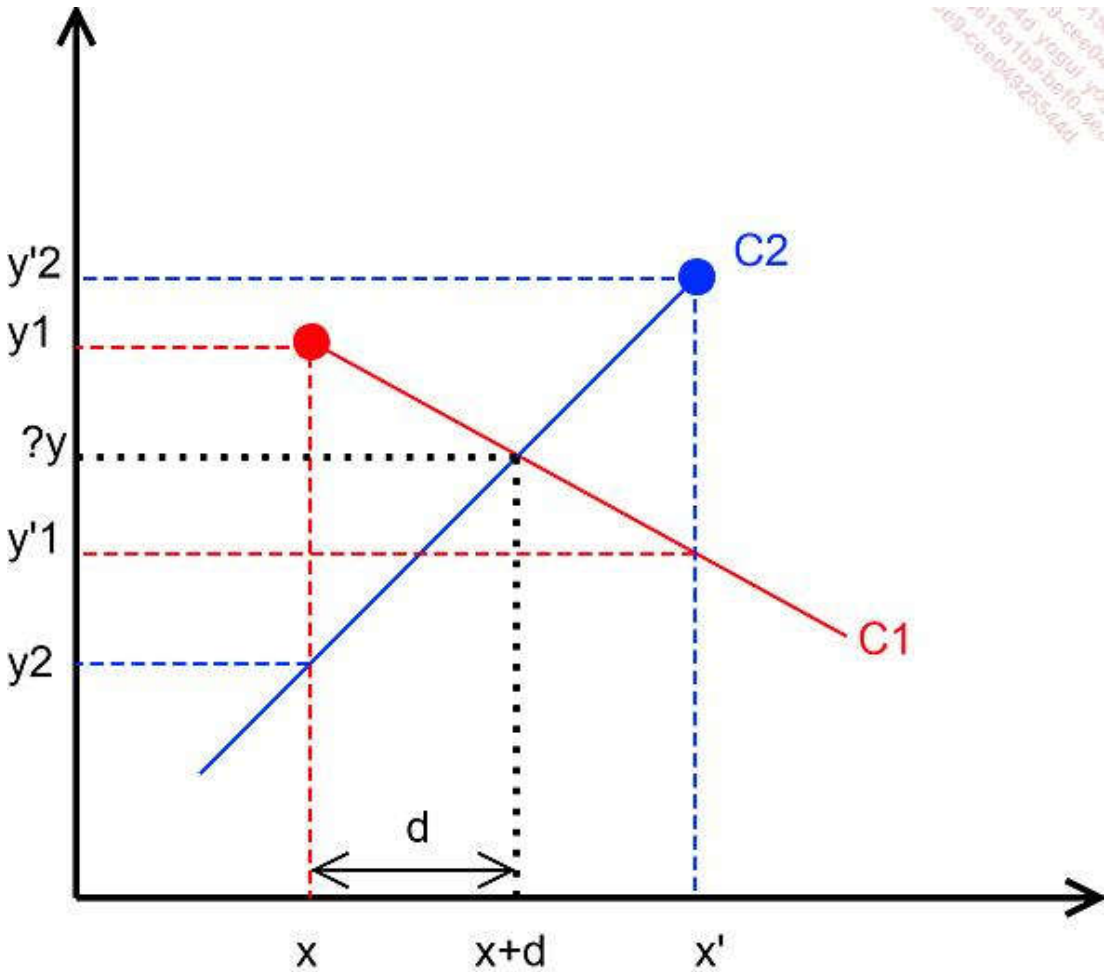
Los distintos casos se ilustran en la siguiente figura:



En x1, ambos conjuntos poseen un valor; basta con calcular el valor correcto (min o max según el operador). En x2, el conjunto de trazo continuo posee un valor, pero no así en el conjunto punteado. Calcularemos su grado de pertenencia y aplicaremos el operador correspondiente. En x3, ambas curvas se invierten, de modo que ninguna posee un punto en este lugar: es preciso detectar que las funciones se cruzan para poder calcular las coordenadas del punto de intersección. Por último, en x4 y x5, solo hay un conjunto que tenga valores: se calculará el valor para el segundo conjunto.

Es necesario, por lo tanto, además de calcular los grados de pertenencia de los conjuntos para cada punto del otro conjunto, saber qué curva se encuentra detrás del otro conjunto para ser capaces de detectar las inversiones. Es preciso calcular los puntos de intersección (que forman parte del conjunto resultante).

Sin embargo, este cálculo no es para nada sencillo. De hecho, no conocemos ni la abscisa ni la ordenada de este punto de intersección, y los puntos anterior y posterior en cada conjunto difuso no están, necesariamente, alineados. Encontramos el siguiente caso:



El punto de la curva C1, que es el último observado, tiene como coordenadas (x, y1). Para la segunda curva (C2), tenemos las coordenadas (x, y2). El punto situado después de la intersección tiene como abscisa x', lo que produce el punto (x', y'1) para la curva C1 y (x', y'2) para la curva C2.

La intersección se produce en x+d. Sabemos que en la intersección los valores en y son iguales para ambas curvas. Llamemos p1 y p2 a las pendientes de estos segmentos. Tenemos:

$$y = y2 + p2 * d = y1 + p1 * d$$

Deducimos que:

$$y2 - y1 = p1 * d - p2 * d$$

$$y2 - y1 = d(p1 - p2)$$

$$d = \frac{y2 - y1}{p1 - p2}$$

La intersección tiene lugar en x+d. Las pendientes son fáciles de calcular y valen:

$$p1 = \frac{(y'1 - y1)}{(x' - x)} \text{ y } p2 = \frac{(y'2 - y2)}{(x' - x)}$$

Podemos escribir una función genérica `Fusionar` que permita fusionar dos conjuntos gracias a un método `Optimo` cuyo nombre se pasa como parámetro. Este método recibe dos valores y devuelve el valor que hay que guardar:

```
    // Método min o max
    private static double Optimo(double valor1, double valor2,
String metodo) {
        if (metodo.equals("Min")) {
            return Math.min(valor1, valor2);
        }
        else {
            return Math.max(valor1, valor2);
        }
    }

    // Método genérico
    private static ConjuntoDifuso Fusionar(ConjuntoDifuso c1,
ConjuntoDifuso c2, String metodo) {
        // Aquí el código
    }
```

Para utilizarla, basta con indicar cuál es el operador matemático deseado: "`Min`" para la intersección y "`Max`" para la unión:

```
    // Operador Y
    public ConjuntoDifuso Y(ConjuntoDifuso c2) {
        return Fusionar(this, c2, "Min");
    }

    // Operador O
    public ConjuntoDifuso O(ConjuntoDifuso c2) {
        return Fusionar(this, c2, "Max");
    }
```

El código del método `Fusionar` es el siguiente:

```
    private static ConjuntoDifuso Fusionar(ConjuntoDifuso c1,
ConjuntoDifuso c2, String metodo) {
       // Creación del resultado
```

```
        ConjuntoDifuso resultado = new ConjuntoDifuso(Math.min(c1.min,
c2.min), Math.max(c1.max, c2.max));

       // Se van a recorrer las listas mediante los iteradores
       Iterator<Punto2D> iterador1 = c1.puntos.iterator();
       Punto2D ptConjunto1 = iterador1.next();
       Punto2D antiguoPtConjunto1 = ptConjunto1;
       Iterator<Punto2D> iterador2 = c2.puntos.iterator();
       Punto2D ptConjunto2 = iterador2.next();

       // Se calcula la posición relativa de las dos curvas
       int antiguaPosicionRelativa;
       int nuevaPosicionRelativa = (int)
Math.signum(ptConjunto1.y - ptConjunto2.y);

       boolean lista1terminada = false;
       boolean lista2terminada = false;
       // Bucle sobre todos los puntos de ambas colecciones
       while(!lista1terminada && !lista2terminada) {
            // Se recuperan las abscisas de los puntos en curso
            double x1 = ptConjunto1.x;
            double x2 = ptConjunto2.x;

            // Cálculo de las posiciones relativas
            antiguaPosicionRelativa = nuevaPosicionRelativa;
            nuevaPosicionRelativa = (int)
Math.signum(ptConjunto1.y - ptConjunto2.y);

            // ¿Las curvas están invertidas?
            // Si no, ¿se tiene un solo punto
            // a tener en cuenta?
            if (antiguaPosicionRelativa !=
nuevaPosicionRelativa && antiguaPosicionRelativa != 0 &&
nuevaPosicionRelativa !=0) {
                // Se debe calcular el punto de intersección
                double x = (x1 == x2 ? antiguoPtConjunto1.x :
Math.min(x1,x2));
                double xPrima = Math.max(x1, x2);

                // Cálculo de las pendientes
                double p1 = c1.ValorDePertenencia(xPrima) -
c1.ValorDePertenencia(x) / (xPrima - x);
                double p2 = c2.ValorDePertenencia(xPrima) -
c2.ValorDePertenencia(x) / (xPrima - x);
                // Cálculo de la delta
                double delta = 0;
                if ((p2-p1) != 0) {
                     delta = (c2.ValorDePertenencia(x) -
c1.ValorDePertenencia(x)) / (p1 - p2);
                }

                // Se agrega un punto de intersección al resultado
                resultado.Agregar(x + delta,
c1.ValorDePertenencia(x + delta));
```

```
                // Se pasa al punto siguiente
                if (x1 < x2) {
                    antiguoPtConjunto1 = ptConjunto1;
                    if (iterador1.hasNext()) {
                        ptConjunto1 = iterador1.next();
                    }
                    else {
                        lista1terminada = true;
                    ptConjunto1 = null;
                    }
                }
                else if (x1 > x2) {
                    if (iterador2.hasNext()) {
                       ptConjunto2 = iterador2.next();
                    }
                    else {
                        ptConjunto2 = null;
                        lista2terminada = true;
                    }
                }
            }
            else if (x1 == x2) {
                // Dos puntos en la misma abscisa,
                // basta con guardar el bueno
                resultado.Agregar(x1, Optimo(ptConjunto1.y,
ptConjunto2.y,  metodo));

                // Se pasa al punto siguiente
                if (iterador1.hasNext()) {
                    antiguoPtConjunto1 = ptConjunto1;
                    ptConjunto1 = iterador1.next();
                }
                else {
                    ptConjunto1 = null;
                    lista1terminada = true;
                }
                if (iterador2.hasNext()) {
                    ptConjunto2 = iterador2.next();
                }
                else {
                    ptConjunto2 = null;
                    lista2terminada = true;
                }
            }
            else if (x1 < x2) {
                // La curva C1 tiene un punto antes
                // Se calcula el grado para el segundo
                // y se guarda el bueno
                resultado.Agregar(x1, Optimo(ptConjunto1.y,
c2.ValorDePertenencia(x1), metodo));
                // Se desplaza
                if (iterador1.hasNext()) {
                    antiguoPtConjunto1 = ptConjunto1;
                    ptConjunto1 = iterador1.next();
                }
```

```
                else {
                    ptConjunto1 = null;
                    lista1terminada = true;
                }
            }
            else {
                // Último caso, la curva C2
                // tiene un punto antes
                // Se calcula el grado para el primero
                // y se guarda el bueno
                resultado.Agregar(x2,
Optimo(c1.ValorDePertenencia(x2), ptConjunto2.y, metodo));
                // Se desplaza
                if (iterador2.hasNext()) {
                    ptConjunto2 = iterador2.next();
                }
                else {
                    ptConjunto2 = null;
                    lista2terminada = true;
                }
            }
        }

        // Aquí, al menos una de las listas se ha terminado
        // Se agregan los puntos restantes
        if (!lista1terminada) {
            while(iterador1.hasNext()) {
                ptConjunto1 = iterador1.next();
                resultado.Agregar(ptConjunto1.x,
Optimo(ptConjunto1.y, 0, metodo));
            }
        }
        else if (!lista2terminada) {
            while(iterador2.hasNext()) {
                ptConjunto2 = iterador2.next();
                resultado.Agregar(ptConjunto2.x,
Optimo(ptConjunto2.y, 0, metodo));
            }
        }

        return resultado;
    }
```

e. Cálculo del baricentro

El último método de esta clase es el que permite determinar el centroide (baricentro o centro de gravedad) del conjunto, para realizar la defuzzificación. En realidad, es la coordenada en x de este punto lo que se busca.

Es posible calcular el baricentro por descomposición, y a continuación realizar la media ponderada de las coordenadas encontradas. Tenemos:

$$C_x = \frac{\sum C_{ix} A_i}{\sum A_i} \text{ y } C_y = \frac{\sum C_{iy} A_i}{\sum A_i},$$

con A_i el área de la forma i, y C_i las coordenadas de los centroides de estas formas.

Es preciso calcular para cada forma las coordenadas de su centroide, así como su área. Guardaremos en una variable el área total, y en otra la suma de las áreas ponderadas por las coordenadas. Bastará, a continuación, con realizar la división para obtener la abscisa buscada.

En nuestro caso, con funciones de pertenencia lineales por tramos, es posible dividir el conjunto en rectángulos y triángulos rectángulos. Los baricentros de los rectángulos estarán situados en su centro, su coordenada en x es la media de sus bordes. Para los triángulos rectángulos, el baricentro está situado a 1/3, en el lado del ángulo recto.

Aplicando estos cálculos obtenemos el siguiente método:

```
    public double Baricentro() {
        // Si hay menos de dos puntos, no hay Baricentro
        if (puntos.size() <= 2) {
            return 0;
        }
        else {
            // Inicialización de las áreas
            double areaPonderada = 0;
            double areaTotal = 0;
            double areaLocal;
            // Recorrer la lista conservando 2 puntos
            Punto2D antiguoPt = null;
            for(Punto2D pt : puntos) {
                if (antiguoPt != null) {
                    // Cálculo del baricentro local
                    if (antiguoPt.y == pt.y) {
                        // Es un rectángulo, el baricentro está en
                        // el centro
                        areaLocal = pt.y * (pt.x - antiguoPt.x);
                        areaTotal += areaLocal;
                        areaPonderada += areaLocal * ((pt.x -
antiguoPt.x) / 2.0  + antiguoPt.x);
                    }
                    else {
                        // Es un trapecio, que podemos descomponer en
                       // un rectángulo con un triángulo
                       // rectángulo adicional
                       // Separamos ambas formas
                       // Primer tiempo: rectángulo
                       areaLocal = Math.min(pt.y, antiguoPt.y) *
(pt.x -  antiguoPt.x);
                       areaTotal += areaLocal;
                       areaPonderada += areaLocal * ((pt.x -
antiguoPt.x) / 2.0 + antiguoPt.x);
                       // Segundo tiempo: triángulo rectángulo
                       areaLocal = (pt.x - antiguoPt.x) * Math.abs(pt.y
- antiguoPt.y) / 2.0;
                       areaTotal += areaLocal;
                       if (pt.y > antiguoPt.y) {
                           // Baricentro a 1/3 del lado pt
                           areaPonderada += areaLocal * (2.0/3.0 *
```

```
(pt.x - antiguoPt.x) + antiguoPt.x);
                        }
                        else {
                            // Baricentro a 1/3 del lado antiguoPt
                            areaPonderada += areaLocal * (1.0/3.0
* (pt.x - antiguoPt.x) + antiguoPt.x);
                        }
                    }
                }
                antiguoPt = pt;
            }
            // Devolvemos las coordenadas del baricentro
            return areaPonderada / areaTotal;
        }
    }
```

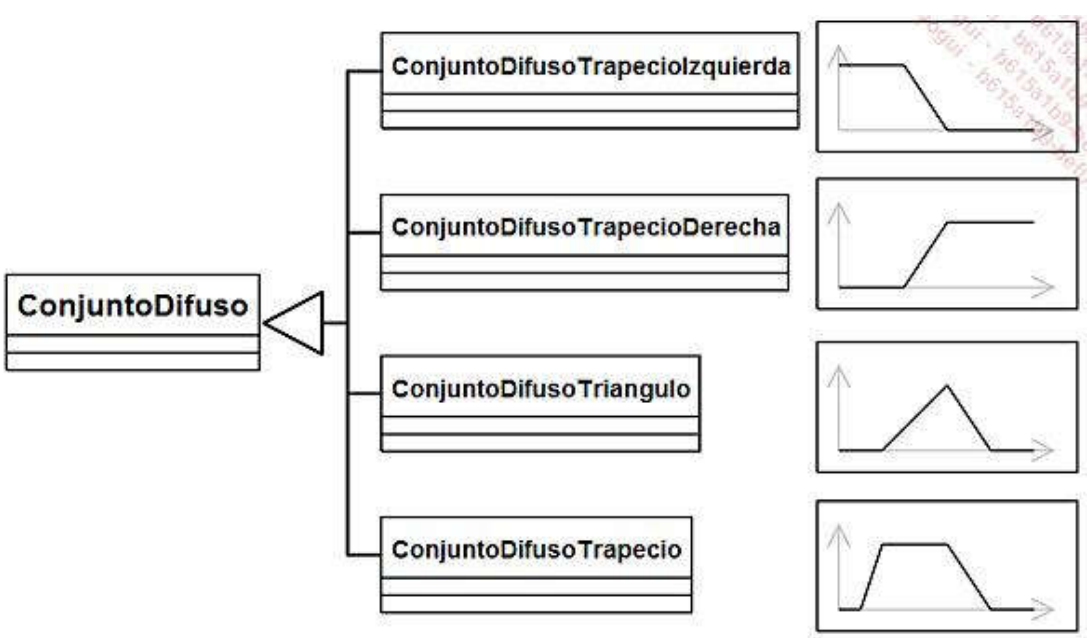
Esta clase es la más compleja de codificar, puesto que utiliza varias fórmulas propias de la geometría euclidiana. Sin embargo, agregar manualmente los puntos puede resultar largo y poco práctico.

2. Conjuntos difusos particulares

Se van a codificar varias clases sencillas. Heredan de la clase `ConjuntoDifuso` para facilitar la creación de los puntos que pertenecen a un conjunto.

Para ello, vamos a crear cuatro clases nuevas:

- **`ConjuntoDifusoTrapecioIzquierda`**, representa una función 1/2 trapecio a la izquierda.
- **`ConjuntoDifusoTrapecioDerecha`**, representa una función 1/2 trapecio a la derecha.
- **`ConjuntoDifusoTriangulo`**, que es una función triangular.
- **`ConjuntoDifusoTrapecio`**, que es una función trapezoidal.



La clase **`ConjuntoDifusoTrapecioIzquierda`** requiere, además de su intervalo de definición, la coordenada

del último punto con altura 1 (`finPlanoElevado`) y del primero con altura 0 (`inicioPlanoBajo`). Invocaremos el constructor de la clase `ConjuntoDifuso` y, a continuación, agregaremos los cuatro puntos necesarios:

```
public class ConjuntoDifusoTrapecioIzquierda extends ConjuntoDifuso {
    // Constructor
    public ConjuntoDifusoTrapecioIzquierda(double min, double max,
double finPlanoElevado, double inicioPlanoBajo) {
        super(min, max);
        Agregar(new Punto2D(min, 1));
        Agregar(new Punto2D(finPlanoElevado, 1));
        Agregar(new Punto2D(inicioPlanoBajo, 0));
        Agregar(new Punto2D(max, 0));
    }
}
```

Para la clase **ConjuntoDifusoTrapecioDerecha**, el principio es el mismo, salvo que hace falta saber hasta qué coordenada el grado es nulo (`finPlanoBajo`), y a continuación a partir de qué coordenada es 1 (`inicioPlanoElevado`).

Se obtiene el siguiente código:

```
public class ConjuntoDifusoTrapecioDerecha extends ConjuntoDifuso {
    // Constructor
    public ConjuntoDifusoTrapecioDerecha(double min, double max,
double finPlanoBajo, double inicioPlanoElevado) {
        super(min, max);
        Agregar(new Punto2D(min, 0));
        Agregar(new Punto2D(finPlanoBajo, 0));
        Agregar(new Punto2D(inicioPlanoElevado, 1));
        Agregar(new Punto2D(max, 1));
    }
}
```

Para la clase **ConjuntoDifusoTriangulo**, necesitamos tres puntos de interés: el comienzo del triángulo (`inicioBase`), el punto culminante (`cima`) y el final del triángulo (`finBase`).

```
public class ConjuntoDifusoTriangulo extends ConjuntoDifuso {
    // Constructor
    public ConjuntoDifusoTriangulo(double min, double max, double
inicioBase, double cima, double finBase) {
        super(min, max);
        Agregar(new Punto2D(min, 0));
        Agregar(new Punto2D(inicioBase, 0));
        Agregar(new Punto2D(cima, 1));
        Agregar(new Punto2D(finBase, 0));
        Agregar(new Punto2D(max, 0));
    }
}
```

Por último, para la clase **ConjuntoDifusoTrapecio**, necesitamos cuatro puntos: el extremo izquierdo de la

base (`inicioBase`), el extremo izquierdo de la meseta (`inicioMeseta`) y los dos extremos derechos (`FinMeseta` y `finBase`).

```
public class ConjuntoDifusoTrapecio extends ConjuntoDifuso {
   // Constructor
   public ConjuntoDifusoTrapecio(double min, double max, double
inicioBase, double inicioMeseta, double finMeseta, double finBase) {
       super(min, max);
       Agregar(new Punto2D(min, 0));
       Agregar(new Punto2D(inicioBase, 0));
       Agregar(new Punto2D(inicioMeseta, 1));
       Agregar(new Punto2D(finMeseta, 1));
       Agregar(new Punto2D(finBase, 0));
       Agregar(new Punto2D(max, 0));
   }
}
```

Estas cuatro clases no son obligatorias, pero van a permitir simplificar en gran medida la creación de conjuntos difusos.

3. Variables y valores lingüísticos

Una vez definidos los conjuntos difusos, podemos crear valores lingüísticos y, a continuación, variables lingüísticas.

a. Valor lingüístico

Un valor lingüístico (**`ValorLinguistico`**) es, simplemente, un nombre asociado a un conjunto difuso, por ejemplo "calor" y el conjunto difuso que lo caracteriza.

La clase no posee más que dos atributos: el conjunto difuso (`ConjuntoDifuso`) y `Nombre`, de tipo `String`.

Posee, también, dos métodos:

- Un constructor que permite inicializar sus atributos.
- Un método que devuelve el grado de pertenencia de un valor numérico determinado (este método simplemente invoca al método del conjunto difuso codificado anteriormente).

He aquí el código de esta clase:

```
public class ValorLinguistico {
    protected ConjuntoDifuso conjuntoDifuso;

    // Nombre del valor
    protected String nombre;

    public ValorLinguistico(String _nombre, ConjuntoDifuso _cd) {
        conjuntoDifuso = _cd;
        nombre = _nombre;
    }
```

```
    double ValorDePertenencia(double valor) {
        return conjuntoDifuso.ValorDePertenencia(valor);
    }
}
```

b. Variable lingüística

La clase variable lingüística (**`VariableLinguistica`**) es también muy sencilla. Una variable lingüística es, ante todo, un nombre (por ejemplo, "temperatura"), a continuación un rango de valores (valores min y max) y por último una lista de valores lingüísticos. Posee, por lo tanto, cuatro atributos: `nombre` (de tipo `String`), `valorMin` y `valorMax` (los valores límite) y `values`, una lista de `ValorLinguistico`.

Además de un constructor, proporciona tres métodos. El primero, `AgregarValorLinguistico`, agrega un nuevo valor lingüístico: bien pasando un objeto `ValorLinguistico` o bien creándolo a partir de un nombre y un conjunto difuso. El segundo, `BorrarValores`, permite borrar la lista de los valores registrados. El último, `ValorLinguisticoPorNombre`, permite encontrar un valor a partir de su nombre.

```
import java.util.ArrayList;

// Clase que representa una variable lingüística
public class VariableLinguistica {
    protected String nombre;
    protected ArrayList<ValorLinguistico> valores;
    protected double valorMin;
    protected double valorMax;

    // Constructor
    public VariableLinguistica(String _nombre, double _min,
double _max) {
        nombre = _nombre;
        valorMin = _min;
        valorMax = _max;
        valores = new ArrayList();
    }

    public void AgregarValorLinguistico(ValorLinguistico vl) {
        valores.add(vl);
    }

    public void AgregarValorLinguistico(String nombre,
ConjuntoDifuso conjunto) {
        valores.add(new ValorLinguistico(nombre, conjunto));
    }

    public void BorrarValores() {
        valores.clear();
    }

    ValorLinguistico ValorLinguisticoPorNombre(String nombre) {
        for(ValorLinguistico vl : valores) {
            if (vl.nombre.equalsIgnoreCase(nombre)) {
            return vl;
            }
```

```
        }
        return null;
    }
}
```

4. Reglas difusas

Las reglas difusas se expresan de la siguiente manera:

```
IF variable IS valor AND ... THEN variable IS valor
```

Por ejemplo:

```
IF temperatura IS calor AND claridad IS fuerte THEN persiana IS bajada
```

a. Expresión difusa

Empezamos definiendo una expresión difusa (**`ExpresionDifusa`**) para expresar la forma "variable IS valor". Para ello, vamos a asociar una variable lingüística `varL` y una cadena correspondiente al valor lingüístico deseado (`nombreValorLinguistico`). Por ejemplo, para "temperatura IS calor", `varL` sería la variable lingüística temperatura, y `nombreValorLinguistico` sería la cadena "calor".

El código de esta clase es el siguiente:

```
public class ExpresionDifusa {
    protected VariableLinguistica varL;
    protected String nombreValorLinguistico;

    // Constructor
    public ExpresionDifusa(VariableLinguistica _vl, String
_valor) {
        varL = _vl;
        nombreValorLinguistico = _valor;
    }
}
```

b. Valor numérico

Vamos a definir también un "valor numérico" (**`ValorNumerico`**), que nos permitirá indicar a continuación a nuestras reglas que actualmente la temperatura es de 21 ºC, por ejemplo. Es preciso asociar una variable lingüística con su valor numérico correspondiente.

El código de esta clase es muy sencillo:

```
public class ValorNumerico {
    protected VariableLinguistica vl;
    protected double valor;
```

```
    // Constructor
    public ValorNumerico(VariableLinguistica _vl, double _valor) {
        vl = _vl;
        valor = _valor;
    }
}
```

c. Regla difusa

Una vez escritas ambas clases, vamos a poder definir reglas difusas (**ReglaDifusa**). Estas contienen una lista de expresiones difusas como premisas (la parte previa al "THEN") y una expresión difusa como conclusión.

La base de esta clase es, por tanto, la siguiente (vamos a completarla a continuación):

```
ArrayList;

// Clase que representa una regla difusa, con varias premisas
// y una conclusión
public class ReglaDifusa {
    protected ArrayList<ExpresionDifusa> premisas;
    protected ExpresionDifusa conclusion;

    // Constructor
    public ReglaDifusa(ArrayList<ExpresionDifusa> _premisas,
ExpresionDifusa _conclusion) {
        premisas = _premisas;
        conclusion = _conclusion;
    }
}
```

El método `Aplicar` permite aplicar una lista de `ValorDifuso` (que es la definición del caso que se ha de resolver) a la regla en curso. Esto produce un conjunto difuso. Todos los conjuntos difusos de todas las reglas se acumularán, a continuación, para obtener la salida difusa, que sufrirá la defuzzificación.

Para ello, recorremos cada premisa de la regla y, cada vez, buscamos el valor numérico correspondiente al valor lingüístico (por ejemplo, buscamos "21" para el valor "calor"). Una vez encontrado, se calcula el grado de pertenencia. Se repite esta búsqueda sobre todas las premisas, guardando cada vez el valor mínimo obtenido. En efecto, el grado de aplicación de una regla se corresponde con el menor grado de pertenencia de las premisas.

Una vez obtenido el grado, se calcula el conjunto difuso resultado, que es la multiplicación del conjunto por el grado (implicación de Larsen).

El código es el siguiente:

```
    // Aplicar la regla a un problema numérico concreto
    // Esto produce un conjunto difuso
    ConjuntoDifuso Aplicar(ArrayList<ValorNumerico> problema) {
        double grado= 1;
        for (ExpresionDifusa premisa : premisas) {
          double gradoLocal = 0;
            ValorLinguistico vl = null;
```

```
            for (ValorNumerico pb : problema) {
                 if (premisa.varL.equals(pb.vl)) {
                    vl =
premisa.varL.ValorLinguisticoPorNombre(premisa.nombreValorLinguistico);
                    if (vl != null) {
                        gradoLocal = vl.ValorDePertenencia(pb.valor);
                        break;
                    }
                }
            }
            if (vl == null) {
                return null;
            }
            grado= Math.min(grado, gradoLocal);
        }
        return conclusion.varL.ValorLinguisticoPorNombre(
conclusion.nombreValorLinguistico).conjuntoDifuso.MultiplicarPor(grado);
    }
}
```

Más adelante se agregará un nuevo método a esta clase para simplificar la escritura de las reglas. No obstante, vamos a definir primero el controlador general.

5. Sistema de control difuso

Para gestionar todo nuestro sistema, se implementa un sistema difuso (**ControladorDifuso**). Este sistema permite gestionar las distintas variables lingüísticas recibidas como entrada, la variable lingüística de salida, las distintas reglas y el problema completo que hay que resolver.

Nuestra versión básica contiene cinco atributos:

- `nombre`: el nombre del sistema.
- `entradas`: la lista de variables lingüísticas de entrada.
- `salida`: la variable lingüística de salida.
- `reglas`: la lista de reglas difusas que hay que aplicar.
- `problema`: la lista de valores numéricos del problema que hay que resolver.

Se agrega un constructor y los métodos que permiten agregar las variables lingüísticas (`AgregarVariableEntrada` y `AgregarVariableSalida`) y las reglas (`AgregarRegla`). Dos métodos permiten crear un nuevo caso para resolver insertando nuevos valores (`AgregarValorNumerico`) o la puesta a cero (`BorrarValoresNumericos`). Por último, se agrega un método que permite encontrar una variable lingüística a partir de su nombre (`VariableLinguisticaPorNombre`).

El código resultante es el siguiente:

```
import java.util.ArrayList;

// Clase que gestiona todo el sistema difuso
public class ControladorDifuso {
    protected String nombre;
```

```
    protected ArrayList<VariableLinguistica> entradas;
    protected VariableLinguistica salida;
    protected ArrayList<ReglaDifusa> reglas;
    protected ArrayList<ValorNumerico> problema;

    // Constructor
    public ControladorDifuso(String _nombre) {
        nombre = _nombre;
        entradas = new ArrayList();
        reglas = new ArrayList();
        problema = new ArrayList();
    }

    // Agregar una variable lingüística como entrada
    public void AgregarVariableEntrada(VariableLinguistica vl) {
        entradas.add(vl);
    }

    // Agregar una variable lingüística como salida
    // Una única posibilidad: reemplaza la existente si es necesario
    public void AgregarVariableSalida(VariableLinguistica vl) {
        salida = vl;
    }

    // Agregar una regla
    public void AgregarRegla(ReglaDifusa regla) {
        reglas.add(regla);
    }

    // Agregar un valor numérico como entrada
    public void AgregarValorNumerico(VariableLinguistica var,
double valor) {
        problema.add(new ValorNumerico(var,valor));
    }

    // Poner a cero el problema (para pasar al siguiente caso)
    public void BorrarValoresNumericos() {
        problema.clear();
    }

    // Encontrar una variable lingüística a partir de su nombre
    public VariableLinguistica VariableLinguisticaPorNombre(String nombre) {
        for (VariableLinguistica var : entradas) {
             if (var.nombre.equalsIgnoreCase(nombre)) {
                 return var;
             }
        }
        if (salida.nombre.equalsIgnoreCase(nombre)) {
             return salida;

        }
        return null;
    }
}
```

En esta clase encontramos el método principal que permite resolver un problema difuso y devolver el valor numérico esperado: `Resolver()`. Sigue las etapas vistas anteriormente, es decir la aplicación de las reglas, una a una, y a continuación la defuzzificación del conjunto difuso resultante.

```
    public double Resolver() {
        // Inicialización del conjunto difuso resultante
        ConjuntoDifuso resultado = new
ConjuntoDifuso(salida.valorMin, salida.valorMax);
        resultado.Agregar(salida.valorMin, 0);
        resultado.Agregar(salida.valorMax, 0);

        // Aplicación de las reglas
        // y modificación del conjunto difuso resultante
        for(ReglaDifusa regla : reglas) {
             resultado = resultado.O(regla.Aplicar(problema));
        }

        // Defuzzificación
        return resultado.Baricentro();
    }
```

No es tan fácil, por el contrario, definir reglas mediante expresiones difusas. Es más sencillo utilizar las reglas basándose en su forma textual. Se agrega un nuevo constructor a la clase **`ReglaDifusa`**. Este descompone la regla escrita en premisas y en una conclusión, y a continuación descompone cada parte en una expresión difusa. Este método hace un uso intenso de funciones de manipulación de cadenas, en particular `split` (para dividir una cadena en subcadenas), `toUpper` (para poner una cadena en mayúsculas) y `replaceFirst` (para eliminar un cierto número de caracteres).

El código comentado es el siguiente:

```
    // Constructor a partir de una cadena de caracteres
    public ReglaDifusa(String reglastr, ControladorDifuso controlador) {
        reglastr = reglastr.toUpperCase();

        String[] regla = reglastr.split(" THEN ");
        if (regla.length == 2) {
            regla[0] = regla[0].replaceFirst("IF ", "").trim();
            String[] premisasStr = regla[0].split(" AND ");
            premisas = new ArrayList();
            for(String exp : premisasStr) {
                String[] partes = exp.trim().split(" IS ");
                if (partes.length == 2) {
                    ExpresionDifusa expDifusa = new
ExpresionDifusa(controlador.VariableLinguisticaPorNombre(partes[0]),
partes[1]);
                    premisas.add(expDifusa);
                }
            }
            String[] concluStr = regla[1].trim().split(" IS ");
            if (concluStr.length == 2) {
                conclusion = new
ExpresionDifusa(controlador.VariableLinguisticaPorNombre(concluStr[0]),
```

```
concluStr[1]);
            }
        }
    }
```

Ahora podemos agregar a nuestro controlador un método que permitirá crear una regla no a partir de un objeto `ReglaDifusa`, sino a partir de una cadena de caracteres.

```
    // Agregar una regla (formato texto)
    public void AgregarRegla(String reglastr) {
        ReglaDifusa regla = new ReglaDifusa(reglastr, this);
        reglas.add(regla);
    }
```

6. Resumen del código creado

Nuestro sistema está ahora completo. Hemos tenido que crear varias clases:

- **`Punto2D`**, que sirve como utilidad para trabajar con conjuntos difusos.
- **`ConjuntoDifuso`**, que representa un conjunto difuso y posee operadores de unión, de intersección, de multiplicación, de comparación y de cálculo del baricentro.
- Cuatro clases que heredan de **`ConjuntoDifuso`** para simplificar la creación de nuevos conjuntos difusos.
- **`VariableLinguistica`** y **`ValorLinguistico`**, que permiten definir variables y valores difusos.
- **`ExpresionDifusa`** y **`ValorNumerico`**, que permiten definir las secciones de las reglas o un caso para tratar.
- **`ReglaDifusa`**, que gestiona las reglas y las crea a partir de una cadena de caracteres. Esta clase también permite crear el conjunto difuso resultante.
- **`ControladorDifuso`**, que gestiona el conjunto.

Implementación de un caso práctico

Vamos a utilizar la lógica difusa para controlar el GPS de un coche, y en particular su nivel de zoom. En efecto, en función de la distancia al próximo cambio de dirección y de la velocidad a la que se circule, el nivel de zoom no es el mismo: cuando nos aproximamos a un cambio de dirección o disminuimos la velocidad, el zoom aumenta para mostrarnos el mapa con un mayor nivel de detalle.

Para obtener una visualización difusa y no entrecortada, se utiliza un controlador difuso. Para ello, creamos en primer lugar una nueva clase que contiene de momento un único método `main`:

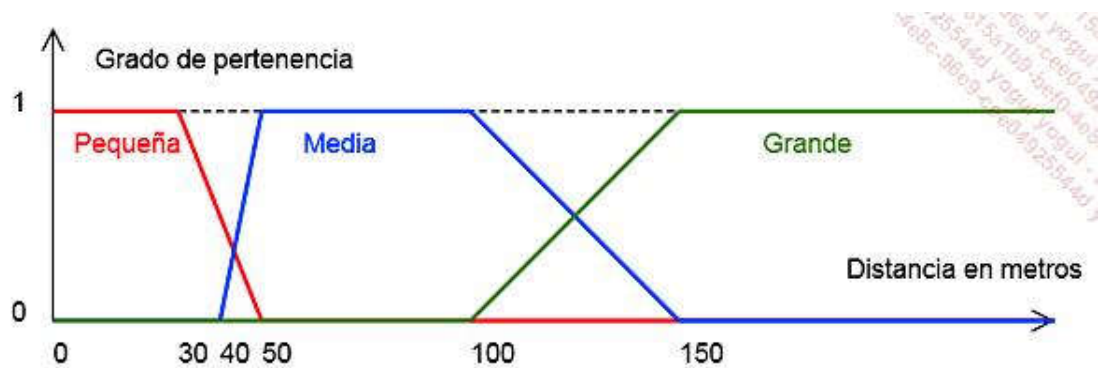
```
public class ZoomGPS {
    public static void main(String[] arg) {
        System.out.println("Lógica difusa: caso del zoom de un GPS");
        // Aquí el código
}
```

Creamos, en primer lugar, un nuevo controlador difuso:

```
        // Creación del sistema
        ControladorDifuso controlador = new ControladorDifuso("Gestión del zoom de un GPS");
```

La siguiente etapa consiste en definir las distintas variables lingüísticas. En nuestro caso tendremos tres: `Distancia` y `Velocidad` como entrada, y `Zoom` como salida. Para la distancia (medida en metros hasta el siguiente cambio de dirección), crearemos tres valores lingüísticos: "`Pequeña`", "`Media`" y "`Grande`".

He aquí el esquema que representa estos conjuntos difusos:

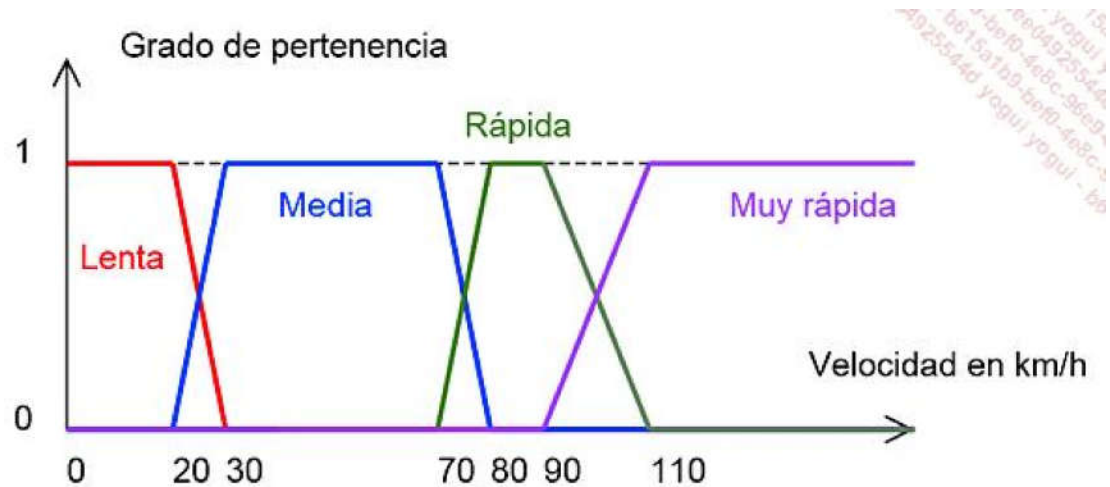


En el código, crearemos la variable lingüística `Distancia`, y a continuación le agregaremos los tres valores lingüísticos; por último, agregaremos esta variable como entrada al sistema:

```
        System.out.println("Agregar las variables de entrada");
        // Variable lingüística de entrada:
        // distancia (en m, de 0 a 500 000)
        VariableLinguistica distancia = new
VariableLinguistica("Distancia", 0, 500000);
        distancia.AgregarValorLinguistico(new
ValorLinguistico("Pequeña", new ConjuntoDifusoTrapecioIzquierda(0,
500000, 30, 50)));
```

```
        distancia.AgregarValorLinguistico(new
ValorLinguistico("Media", new ConjuntoDifusoTrapecio(0, 500000,
40, 50, 100, 150)));
        distancia.AgregarValorLinguistico(new
ValorLinguistico("Grande", new ConjuntoDifusoTrapecioDerecha(0,
500000, 100, 150)));
        controlador.AgregarVariableEntrada(distancia);
```

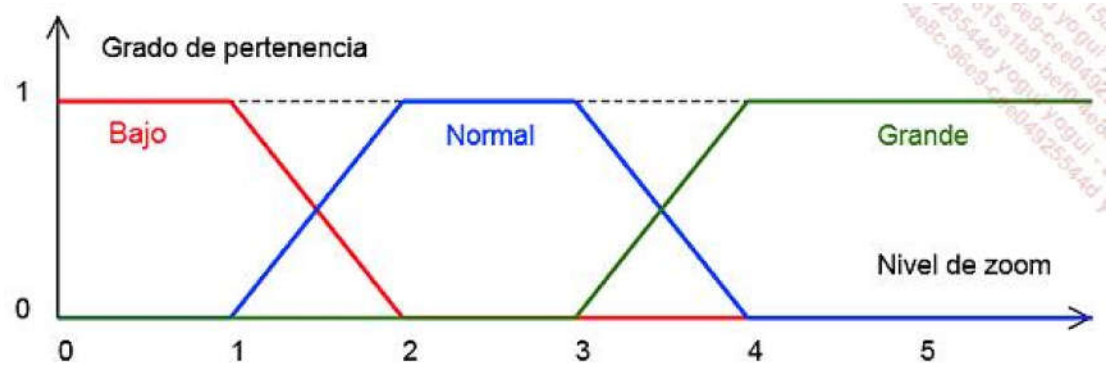
Realizamos la misma tarea con la velocidad. He aquí los distintos valores y los conjuntos difusos correspondientes (nos detendremos a **200 km/h**):



Obtenemos el siguiente código:

```
        // Variable lingüística de entrada:
        // velocidad (en km/h, de 0 a 200)
        VariableLinguistica velocidad = new
VariableLinguistica("Velocidad", 0, 200);
        velocidad.AgregarValorLinguistico(new
ValorLinguistico("Lenta", new ConjuntoDifusoTrapecioIzquierda(0, 200,
20, 30)));
        velocidad.AgregarValorLinguistico(new
ValorLinguistico("Media", new ConjuntoDifusoTrapecio(0, 200,
20, 30, 70, 80)));
        velocidad.AgregarValorLinguistico(new
ValorLinguistico("Rapida", new ConjuntoDifusoTrapecio(0, 200, 70,
80, 90, 110)));
        velocidad.AgregarValorLinguistico(new
ValorLinguistico("MuyRapida", new ConjuntoDifusoTrapecioDerecha(0,
200, 90, 110)));
        controlador.AgregarVariableEntrada(velocidad);
```

Por último, para el nivel de zoom, se definen tres valores lingüísticos. El nivel se define mediante un valor numérico entre 0 (el mapa no tiene ningún zoom, se ve a lo lejos) y 5 (nivel máximo de zoom, con muchos detalles pero una visibilidad menor).



El código de creación de esta variable (de salida del sistema) es el siguiente:

```
     System.out.println("Agregar la variable de salida");
     // Variable lingüística de salida: nivel de zoom (de 1 a 5)
     VariableLinguistica zoom = new VariableLinguistica("Zoom", 0, 5);
     zoom.AgregarValorLinguistico(new
ValorLinguistico("Bajo", new ConjuntoDifusoTrapecioIzquierda(0, 5, 1,
2)));
     zoom.AgregarValorLinguistico(new
ValorLinguistico("Normal", new ConjuntoDifusoTrapecio(0, 5, 1, 2, 3,
4)));
     zoom.AgregarValorLinguistico(new
ValorLinguistico("Grande", new ConjuntoDifusoTrapecioDerecha(0, 5,
3, 4)));
     controlador.AgregarVariableSalida(zoom);
```

Una vez creadas las variables, tenemos que definir las reglas. Se deciden aplicar las reglas siguientes (que indican el nivel de zoom en función de la velocidad y de la distancia). Por ejemplo, si la velocidad es lenta y la distancia es mucha, entonces el nivel de zoom debe ser bajo.

Distancia → **Velocidad ↓**	**Pequeña**	**Media**	**Grande**
Lenta	Normal	Bajo	Bajo
Media	Normal	Normal	Bajo
Rápida	Grande	Normal	Bajo
Muy rápida	Grande	Grande	Bajo

Codificaremos, por lo tanto, estas reglas. Para ganar algo de tiempo, como el nivel de zoom debe ser bajo si la distancia es grande, independientemente de la velocidad, podemos agrupar todos estos casos en una única regla. Tenemos, por tanto, 9 reglas que permiten cubrir los 12 casos posibles.

```
   System.out.println("Agregar las reglas");
   // Se agregan las distintas reglas (9 para cubrir los 12 casos)
   controlador.AgregarRegla("IF Distancia IS Grande THEN Zoom IS
Bajo");
    controlador.AgregarRegla("IF Distancia IS Pequeña
AND Velocidad IS Lenta THEN Zoom IS Normal");
```

```
    controlador.AgregarRegla("IF Distancia IS Pequeña
AND Velocidad IS Media THEN Zoom IS Normal");
    controlador.AgregarRegla("IF Distancia IS Pequeña
AND Velocidad IS Rapida THEN Zoom IS Grande");
    controlador.AgregarRegla("IF Distancia IS Pequeña
AND Velocidad IS MuyRapida THEN Zoom IS Grande");
    controlador.AgregarRegla("IF Distancia IS Media
AND Velocidad IS Lenta THEN Zoom IS Bajo");
    controlador.AgregarRegla("IF Distancia IS Media
AND Velocidad IS Media THEN Zoom IS Normal");
    controlador.AgregarRegla("IF Distancia IS Media
AND Velocidad IS Rapida THEN Zoom IS Normal");
    controlador.AgregarRegla("IF Distancia IS Media
AND Velocidad IS MuyRapida THEN Zoom IS Grande");
```

A continuación vamos a resolver cinco casos diferentes, resumidos en la siguiente tabla:

Caso n.º	1	2	3	4	5
Distancia	70 m	70 m	40 m	110 m	160 m
Velocidad	35 km/h	25 km/h	72.5 km/h	100 km/h	45 km/h

Vamos a codificar los distintos casos:

```
    System.out.println("Resolución de casos prácticos");
    // Caso práctico 1: velocidad de 35 km/h,
    // y próximo cambio de dirección a 70 m
    System.out.println("Caso 1:");
    controlador.AgregarValorNumerico(velocidad, 35);
    controlador.AgregarValorNumerico(distancia, 70);
    System.out.println("Resultado: " + controlador.Resolver() + "\n");

    // Caso práctico 2: velocidad de 25 km/h,
    // y próximo cambio de dirección a 70 m
    controlador.BorrarValoresNumericos();
    System.out.println("Caso 2:");
    controlador.AgregarValorNumerico(velocidad, 25);
    controlador.AgregarValorNumerico(distancia, 70);
    System.out.println("Resultado: " + controlador.Resolver() + "\n");

    // Caso práctico 3: velocidad de 72.5 km/h,
    // y próximo cambio de dirección a 40 m
    controlador.BorrarValoresNumericos();
    System.out.println("Caso 3:");
    controlador.AgregarValorNumerico(velocidad, 72.5);
    controlador.AgregarValorNumerico(distancia, 40);
    System.out.println("Resultado: " + controlador.Resolver() + "\n");

    // Caso práctico 4: velocidad de 100 km/h,
    // y próximo cambio de dirección a 110 m
    controlador.BorrarValoresNumericos();
    System.out.println("Caso 4:");
    controlador.AgregarValorNumerico(velocidad, 100);
```

```
    controlador.AgregarValorNumerico(distancia, 110);
    System.out.println("Resultado: " + controlador.Resolver() + "\n");

    // Caso práctico 5: velocidad de 45 km/h, y cambio a 160 m
    controlador.BorrarValoresNumericos();
    System.out.println("Caso 5:");
    controlador.AgregarValorNumerico(velocidad, 45);
    controlador.AgregarValorNumerico(distancia, 160);
    System.out.println("Resultado: " + controlador.Resolver() + "\n");
    }
```

He aquí el resultado obtenido ejecutando el programa:

```
Lógica difusa: caso del zoom de un GPS
Agregar las variables de entrada
Agregar la variable de salida
Agregar las reglas
Resolución de casos prácticos
Caso 1:
Resultado: 2.5

Caso 2:
Resultado: 1.7820512820512822

Caso 3:
Resultado: 2.9318996415770604

Caso 4:
Resultado: 2.89196256537297

Caso 5:
Resultado: 0.7777777777777777
```

Comprobamos que el nivel de zoom, según el caso, irá de 0.78 para el quinto caso hasta 2.93 para el tercer caso. Esto certifica el comportamiento esperado: cuanto más próximo esté el siguiente cambio de dirección o más lentamente se conduzca, más importante es ver el detalle en el GPS (mediante un nivel de zoom más elevado).

Resumen

La lógica difusa permite tomar decisiones en función de reglas poco precisas, es decir, cuya evaluación esté sometida a cierta interpretación.

Para ello, se definen variables lingüísticas (como la temperatura) y se les asocia valores lingüísticos ("calor", "frío"...). A cada valor se le hace corresponder un conjunto difuso, determinado por su función de pertenencia: para todos los valores numéricos posibles, se asocia un grado de pertenencia entre 0 (el valor lingüístico es totalmente falso) y 1 (es totalmente verdadero), pasando por estados intermedios.

Una vez definidas las variables y los valores lingüísticos, se indican al sistema difuso las reglas que hay que aplicar. Se le indican, a continuación, los valores numéricos medidos.

La primera etapa para proporcionar una decisión es la fuzzificación, que consiste en asociar a cada valor numérico su grado de pertenencia a los distintos valores lingüísticos. A continuación se pueden aplicar las reglas, y la suma de las reglas (constituida por la unión de los conjuntos difusos resultantes) proporciona un nuevo conjunto difuso.

La defuzzificación es la etapa que permite pasar de este conjunto difuso al valor numérico que representa la decisión final. Existen varios métodos para llevar a cabo esta tarea, aunque el cálculo del baricentro es la más precisa; además, tiene un coste de cálculo adicional bastante despreciable en relación con la capacidad actual de los ordenadores.

Tan solo queda aplicar esta decisión. El sistema de gestión difuso permite también realizar un control en tiempo real, y aplica pequeñas modificaciones en la salida si las entradas varían ligeramente, lo que permite una mayor flexibilidad, y un menor desgaste en los sistemas mecánicos controlados. Además, resulta más próximo al comportamiento humano, lo que permite múltiples usos.

Presentación del capítulo

Muchos dominios se enfrentan a un problema de búsqueda de rutas, denominada "pathfinding" en inglés. Recordamos, en primer lugar, los GPS y los programas de búsqueda de itinerarios (en coche, en tren, en transporte público...), también los videojuegos en los que los enemigos deben alcanzar al jugador por la ruta más corta.

La búsqueda de rutas es, en realidad, un dominio más bien vasto. En efecto, muchos problemas pueden representarse bajo la forma de un grafo, como la sucesión de movimientos en un juego de ajedrez.

La búsqueda de una ruta en este caso puede verse como la búsqueda de la serie de movimientos que se deben realizar para ganar.

Este capítulo presenta, en primer lugar, los distintos conceptos de la teoría de grafos y las definiciones asociadas. A continuación se presentan los algoritmos fundamentales, con su funcionamiento y sus restricciones.

Más adelante se exponen los principales dominios en los que se puede utilizar esta búsqueda de rutas y se presenta un ejemplo de implementación de estos algoritmos en Java, aplicado a una búsqueda de rutas en un entorno en 2D.

El capítulo termina con un resumen.

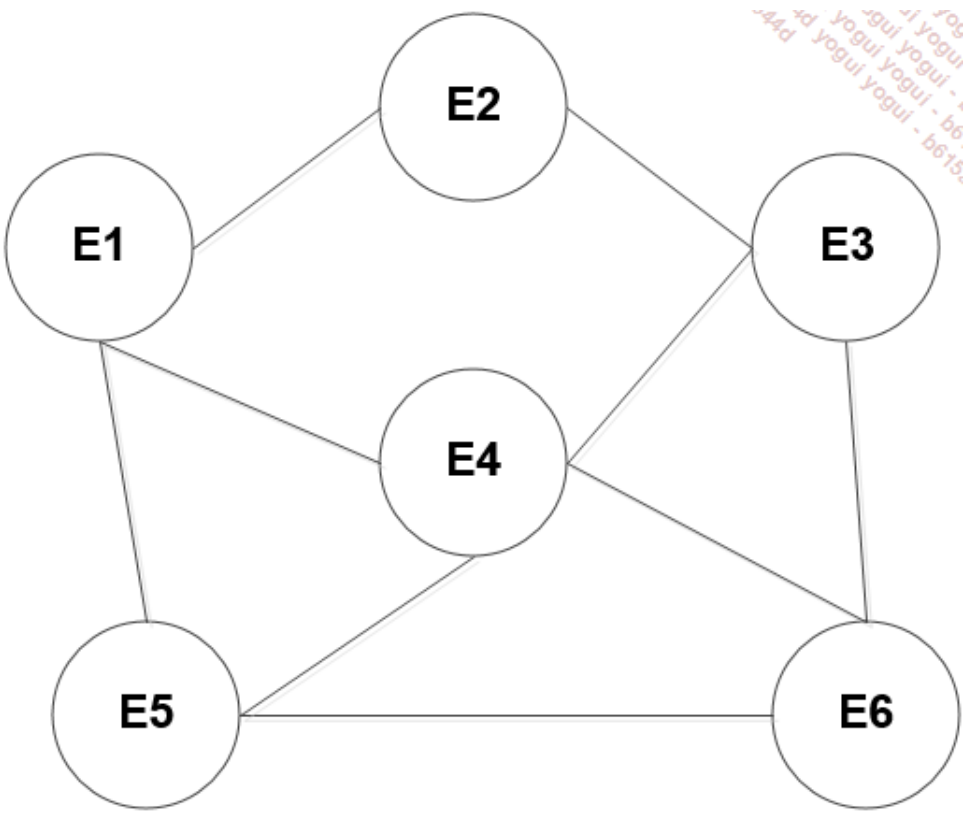
Rutas y grafos

Una ruta puede verse como un recorrido por un grafo. Los principales algoritmos se basan, por lo tanto, en la **teoría de grafos**.

1. Definición y conceptos

Un **grafo** es un conjunto de **nodos** o **vértices** (que pueden representar, por ejemplo, ciudades) ligados por **arcos**, que serán las rutas.

He aquí un grafo que representa las estaciones y los vínculos que existen entre ellas (en tren, sin trasbordo):



Las estaciones E1 a E6 son los nodos. El arco que va de E5 a E6 indica la presencia de un enlace directo entre estas dos estaciones. Se escribe (E5, E6) o (E6, E5) según el sentido deseado.

Por el contrario, para ir de E1 a E6 no existe ningún enlace directo, será preciso pasar por E4 o por E5 si se desea realizar un único trasbordo, o por E2 y, a continuación, por E3 con dos trasbordos.

Una **ruta** permite alcanzar varios destinos unidos entre sí mediante arcos. De este modo, E1-E2-E3-E6 es una ruta de **distancia** 3 (la distancia es el número de arcos seguidos).

Hablamos de **circuito** cuando es posible partir de un nodo y volver a él. Aquí, el grafo contiene varios circuitos, como por ejemplo E1-E4-E5-E1 o E4-E5-E6-E4.

El **orden** de un grafo se corresponde con el número de destinos que contiene. Nuestro ejemplo contiene 6 estaciones; se trata por tanto de un grafo de orden 6.

Dos nodos se dice que son **adyacentes** (o vecinos) si existe un vínculo que permite ir de uno al otro. E5 es, por tanto, adyacente a E1, E4 y E6.

2. Representaciones

a. Representación gráfica

Existen varias maneras de representar un grafo. La primera es la **representación gráfica**, como hemos visto antes.

El orden y la situación de los nodos no son importantes; no obstante, se va a intentar buscar situar siempre los destinos de forma que el grafo sea lo más legible posible.

Se dice que el grafo está **orientado** si los arcos tienen un sentido, representando por ejemplo calles de sentido único en una ciudad. Si todos los arcos pueden tomarse en ambos sentidos, se dice que el grafo está **no orientado**, como es generalmente el caso en los grafos utilizados en la búsqueda de rutas.

b. Matriz de adyacencia

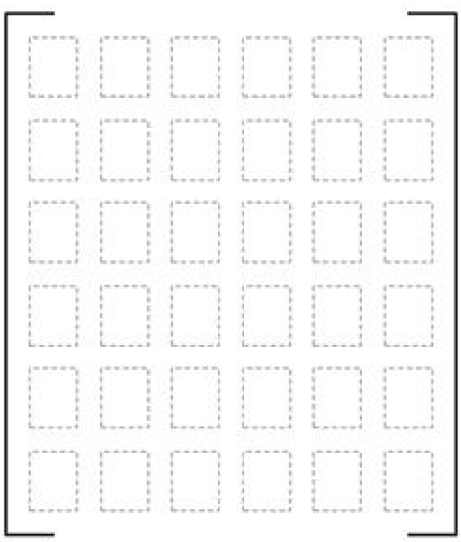
Las representaciones gráficas no siempre son prácticas, en particular cuando se trata de aplicar algoritmos o escribirlos en un ordenador.

Se prefiere, a menudo, utilizar una matriz, llamada **matriz de adyacencia**.

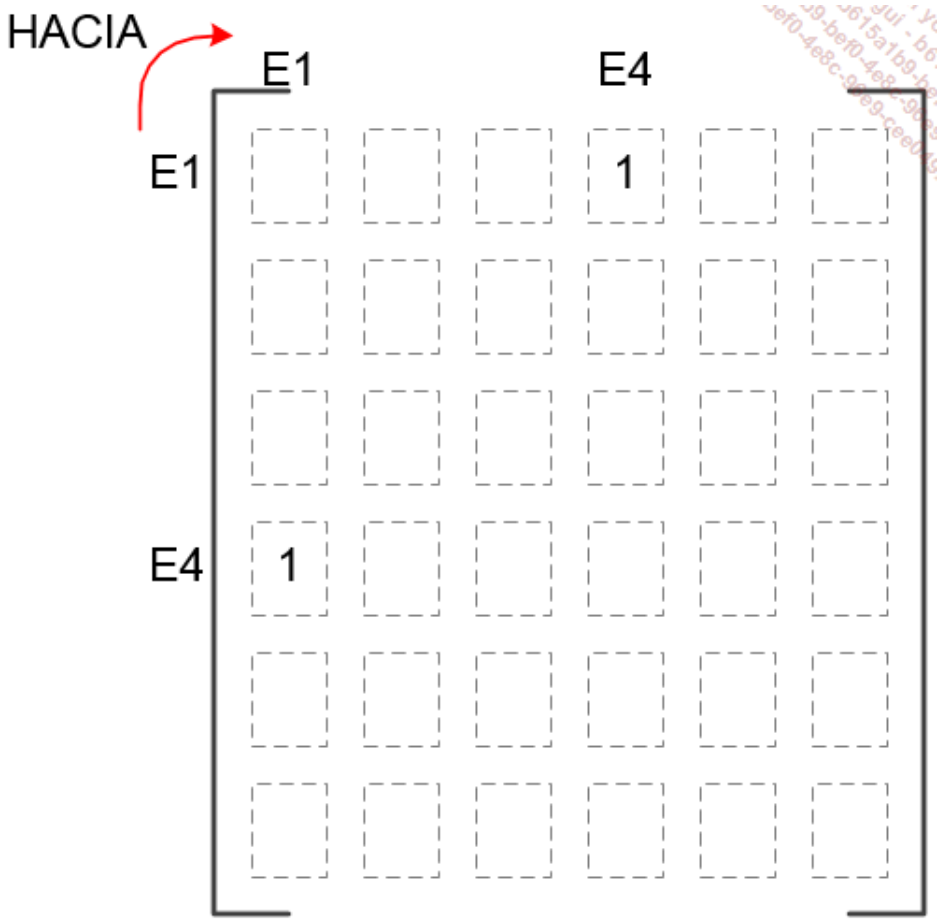
Una matriz es una estructura matemática particular que puede verse simplemente como una tabla de dos dimensiones.

En esta matriz, la ausencia de un arco se representa por un 0, y su presencia mediante un 1.

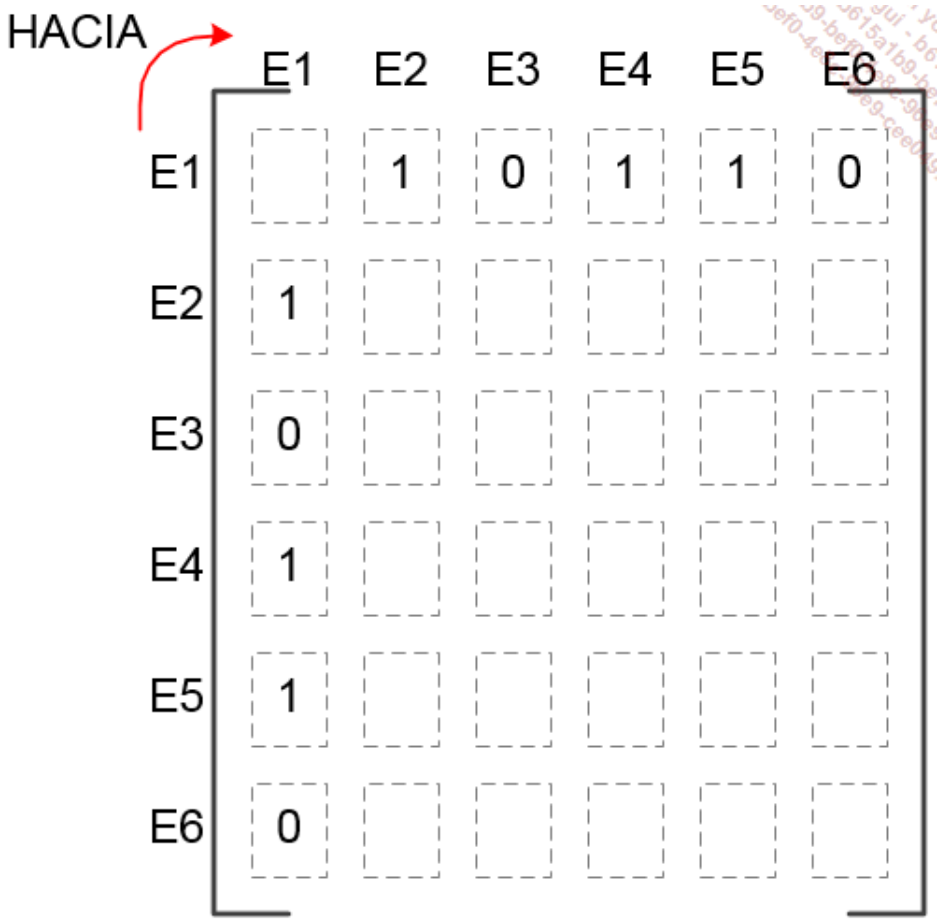
En el ejemplo de las estaciones, tenemos una matriz de 6 por 6 (puesto que existen 6 estaciones):



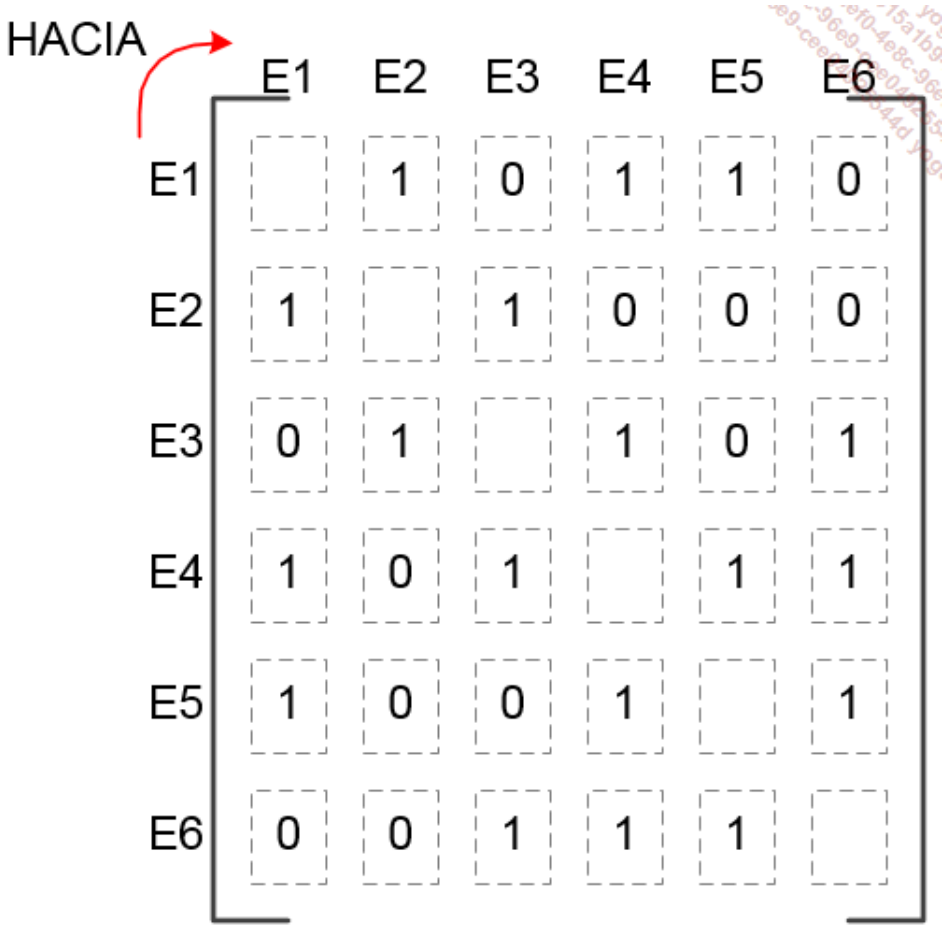
Vemos en el grafo que existe un vínculo entre E1 y E4. La casilla correspondiente al trayecto de E1 a E4 contiene, por lo tanto, un 1, igual que la de E4 a E1 (el trayecto es de doble sentido). Obtenemos la siguiente matriz:



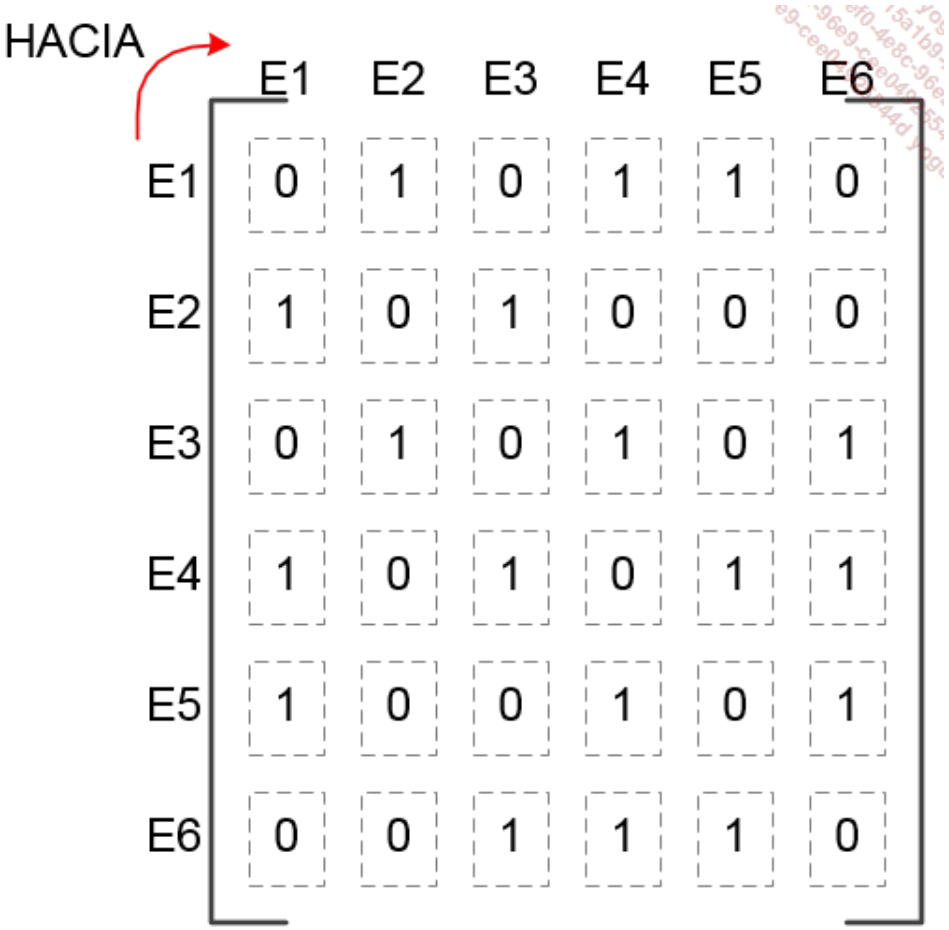
Del mismo modo, existe un arco desde E1 hacia E2 y E5, pero no hacia E3 ni E6. Podemos completar nuestra matriz:



Hacemos lo mismo para los demás nodos y los demás arcos:



Solo nos queda la diagonal. Representa la posibilidad de ir desde un nodo hacia sí mismo, lo cual se denomina un bucle. En este caso, no existe ningún trayecto directo que permita ir de una estación a sí misma, de modo que rellenamos con 0 esta diagonal.

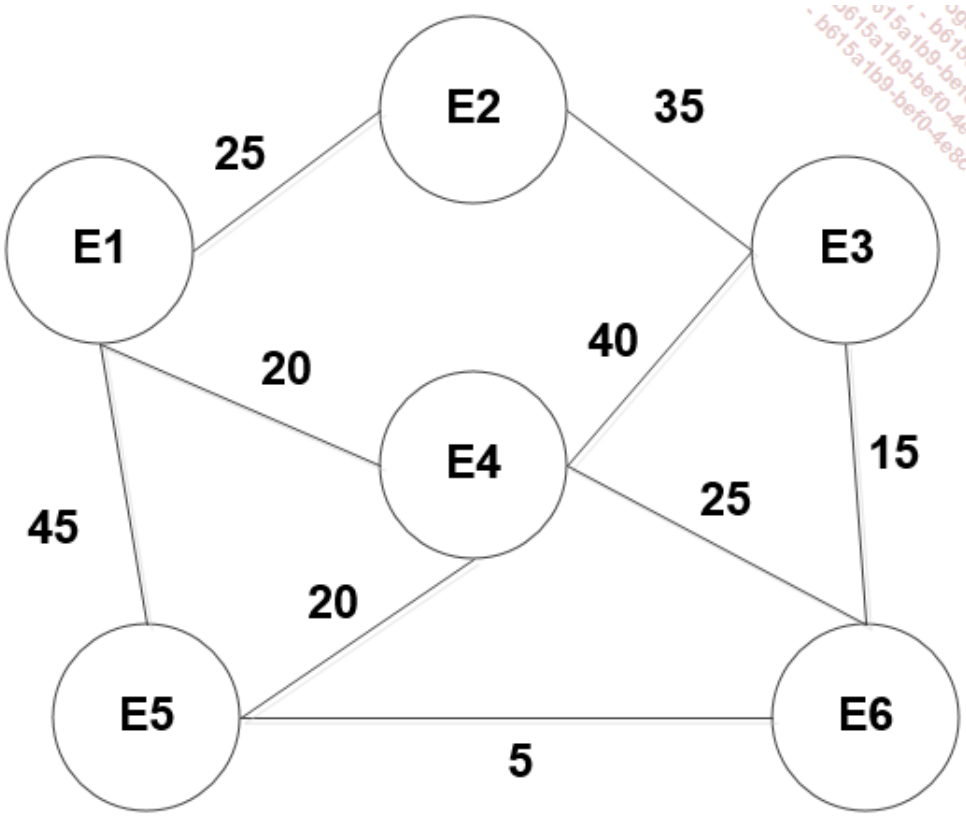


La matriz de adyacencia está ahora completa.

3. Coste de una ruta y matriz de distancias

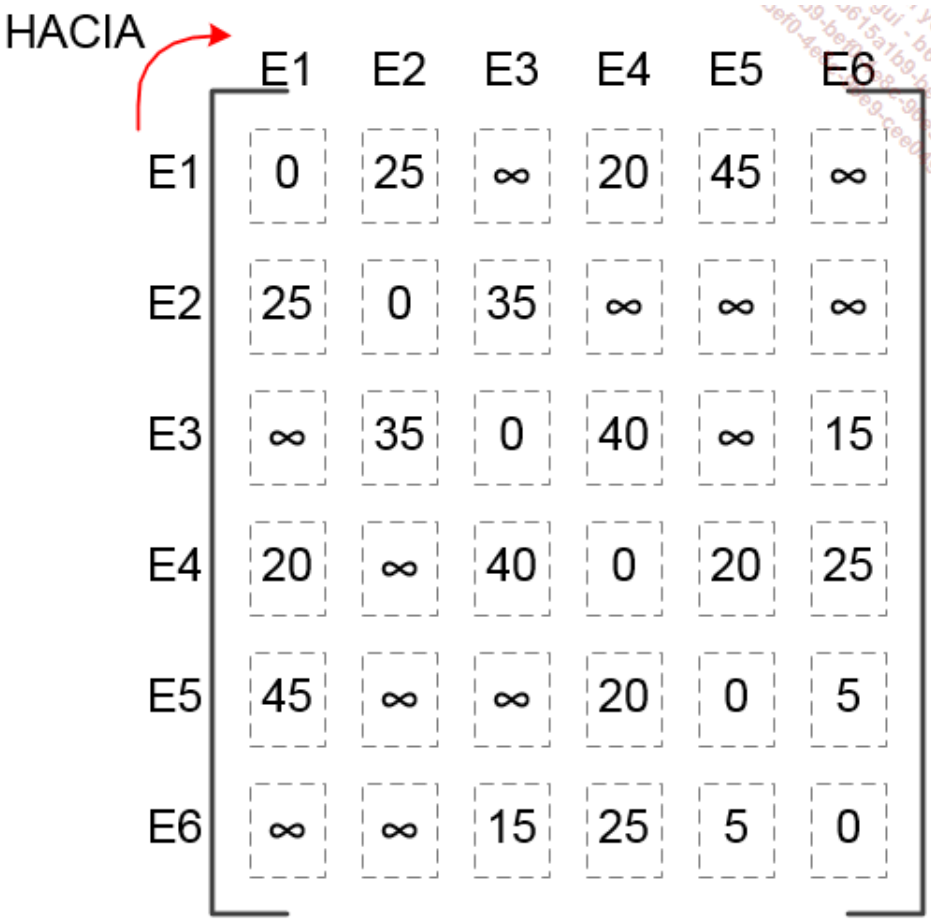
En el marco de la búsqueda de la ruta más corta, la menos cara o la más rápida, debemos agregar cierta información. Esta se denomina de forma arbitraria **distancias**, sin precisar la unidad. Puede tratarse de kilómetros, de euros, de minutos, de kilos...

En la representación gráfica, agregamos las distancias sobre el dibujo de los arcos. Para el ejemplo de las estaciones, resulta interesante conocer el tiempo necesario, en minutos, para realizar el recorrido entre las estaciones, y el gráfico queda:



La matriz de adyacencia se convierte en la **matriz de distancias**. Los "1" que representaban los enlaces se han reemplazado por la distancia del arco. Cuando no existe ningún enlace, se reemplaza el 0 por +∞, que indica que para ir de un nodo al otro haría falta un tiempo/kilometraje/coste infinito (por lo que resulta imposible). Para la diagonal, se informan 0, que indican que para ir de un lugar a sí mismo no es necesario moverse, de modo que resulta "gratuito" e inmediato.

La matriz de distancias queda, entonces:



La matriz de distancias es la que se utiliza principalmente en el código, incluso aunque los algoritmos se expliquen con ayuda de la representación gráfica más adelante.

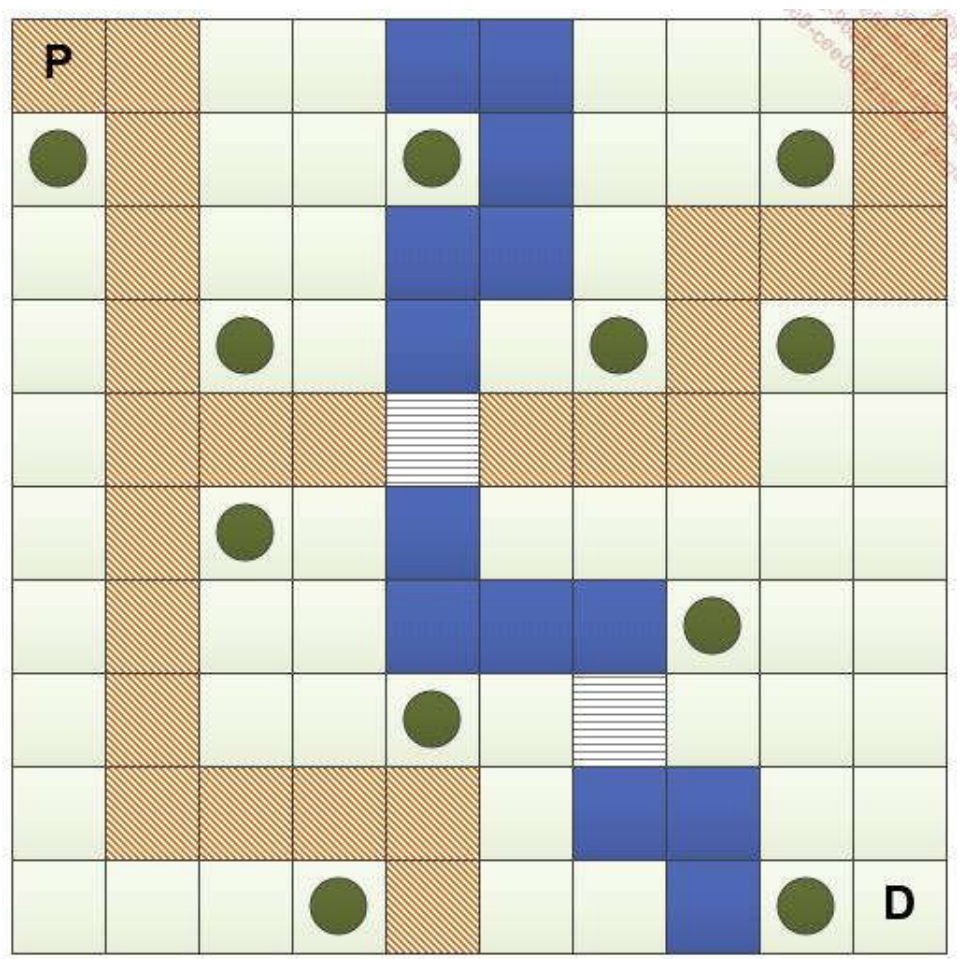
Ejemplo en cartografía

Para estudiar los distintos algoritmos, vamos a interesarnos en un pequeño juego en el que controlamos a un personaje que representa un explorador. Como en muchos juegos de rol, nuestro héroe está limitado en cada turno (no tiene derecho más que a cierto número de puntos de acción). Para ir lo más rápido posible desde un punto a otro, buscaremos la ruta más corta en el mapa, teniendo en cuenta el tipo de terreno.

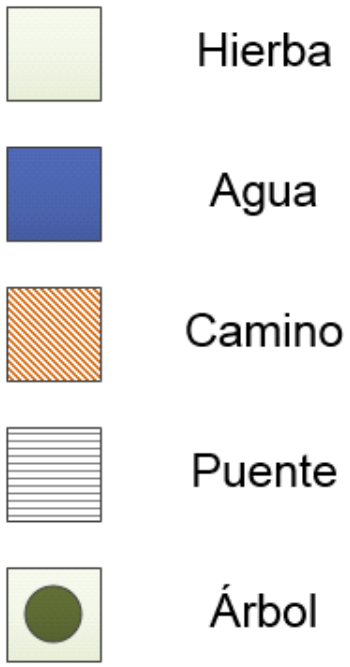
Existen varias formas, que requieren más o menos energía (y en consecuencia puntos de acción):

- Caminos, que requieren un punto de acción por casilla.
- Hierba, que requiere dos puntos de acción.
- Puentes, que requieren dos puntos de acción.
- Agua, infranqueable.
- Árboles, infranqueables también.

El mapa es el siguiente:



Y la leyenda:



Vamos a buscar la ruta que permita ir desde el punto de partida (P) hasta el destino (D), utilizando la menor cantidad posible de puntos de acción. La ruta más corta cuesta 27 puntos de acción (siguiendo la ruta y pasando el primer puente, a continuación acortando por hierba hasta el final).

Algoritmos exhaustivos de búsqueda de rutas

Estos primeros algoritmos no son "inteligentes": se denominan **exhaustivos**, puesto que no utilizan ningún conocimiento relativo al problema para proceder. En el peor de los casos, comprueban todas las rutas posibles para determinar si existe alguna ruta entre ambos nodos.

Además, nada indica que la ruta encontrada sea la más corta. Sin embargo, son muy fáciles de implementar.

1. Búsqueda en profundidad

Se trata del algoritmo que probamos de manera natural en un laberinto: buscamos avanzar lo más rápido posible y, si nos bloqueamos, volvemos a la última intersección que hemos encontrado y probamos una nueva ruta.

Este algoritmo no permite determinar el camino más corto, sino simplemente encontrar un camino.

a. Principio y pseudocódigo

Su funcionamiento es bastante sencillo.

En primer lugar, seleccionamos el orden en que recorreremos los nodos y, a continuación, aplicaremos este orden para avanzar lo máximo posible. Si nos bloqueamos, volvemos atrás y probamos la siguiente posibilidad.

El recorrido en profundidad permite, por tanto, determinar la existencia de una ruta, pero no tiene en cuenta la longitud de los arcos, y por tanto no permite saber si se ha encontrado el camino más corto.

Además, el resultado obtenido depende en gran medida del orden escogido para recorrer el grafo, el orden óptimo depende de cada problema y no puede determinarse a priori. A menudo, no es eficaz. En el peor de los casos, debe comprobar todas las posibilidades para encontrar un camino.

Es, sin embargo, muy fácil de implementar. En efecto, vamos a conservar una lista de los nodos visitados. Cada vez que nos encontremos sobre un nodo, vamos a agregar todos sus vecinos no visitados a una pila en el orden seleccionado. A continuación, extraeremos el primer elemento de la pila.

Una pila (o LIFO, del inglés Last In, First Out) es una estructura algorítmica en la que los elementos se agregan en la parte superior y se extraen partiendo de la parte superior, como una pila de platos, de papeles... No es posible extraer un elemento de la mitad de la pila. Agregar un elemento por encima se denomina apilar, y extraerlo, desapilar.

Para facilitar la reconstrucción del camino obtenido, se conserva el predecesor de cada nodo (es decir, el nodo que nos ha permitido avanzar).

El pseudocódigo es el siguiente:

```
// Inicialización del array
Crear array Precursor
Para cada nodo n
   Precursor[n] = null

// Creación de la lista de nodos no visitados, y de la pila
Crear lista NodosNoVisitados = conjunto de nodos
Crear pila PorVisitar
PorVisitar.Apilar(inicio)
```

```
// Bucle principal
Mientras PorVisitar no vacío
   Nodo encurso = PorVisitar.Desapilar
   Si encurso = destino
          Fin (OK)
   Si no
          Para cada vecino v de n
                 Si v en NodosNoVisitados
                       Extraer v de NodosNoVisitados
                       Precursor[v] = n
                       PorVisitar.Apilar(v)
```

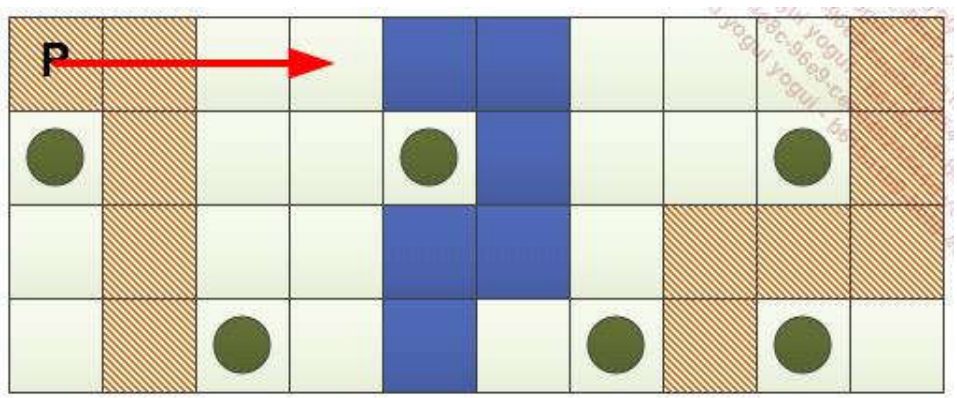
Los vecinos deben apilarse en el orden inverso al orden seleccionado para poner el primero que se deba visitar el último en la pila y que quede arriba del todo.

b. Aplicación al mapa

En nuestro caso, vamos a ver con detalle cómo aplicar el orden siguiente:

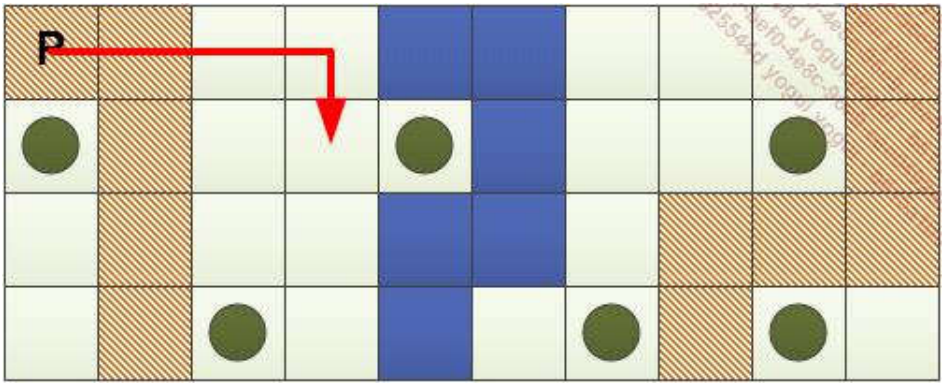
- a la derecha,
- hacia abajo,
- a la izquierda,
- y por último hacia arriba.

Empezamos en el nodo de partida y vamos a intentar aplicar lo máximo posible el camino que va a la derecha, hasta vernos bloqueados.



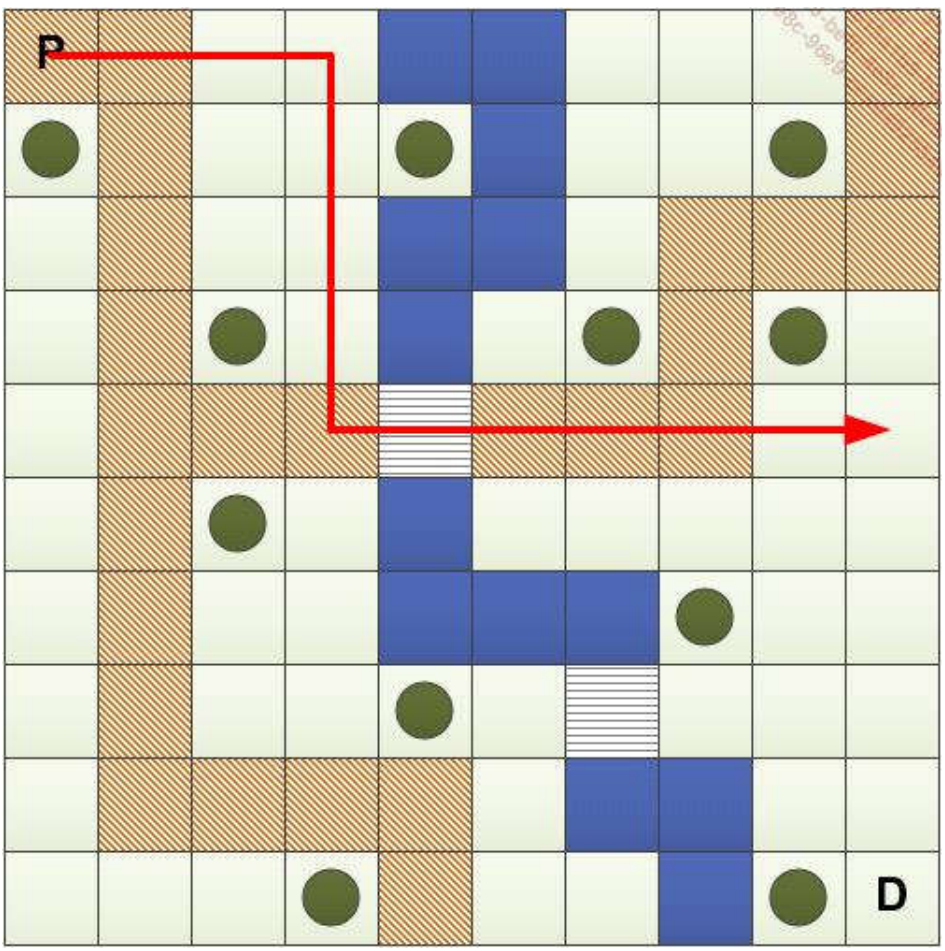
Por motivos de legibilidad, solo se representa la zona superior del mapa.

Una vez bloqueados, vamos a cambiar de dirección e ir hacia abajo (la segunda opción) en la siguiente casilla.

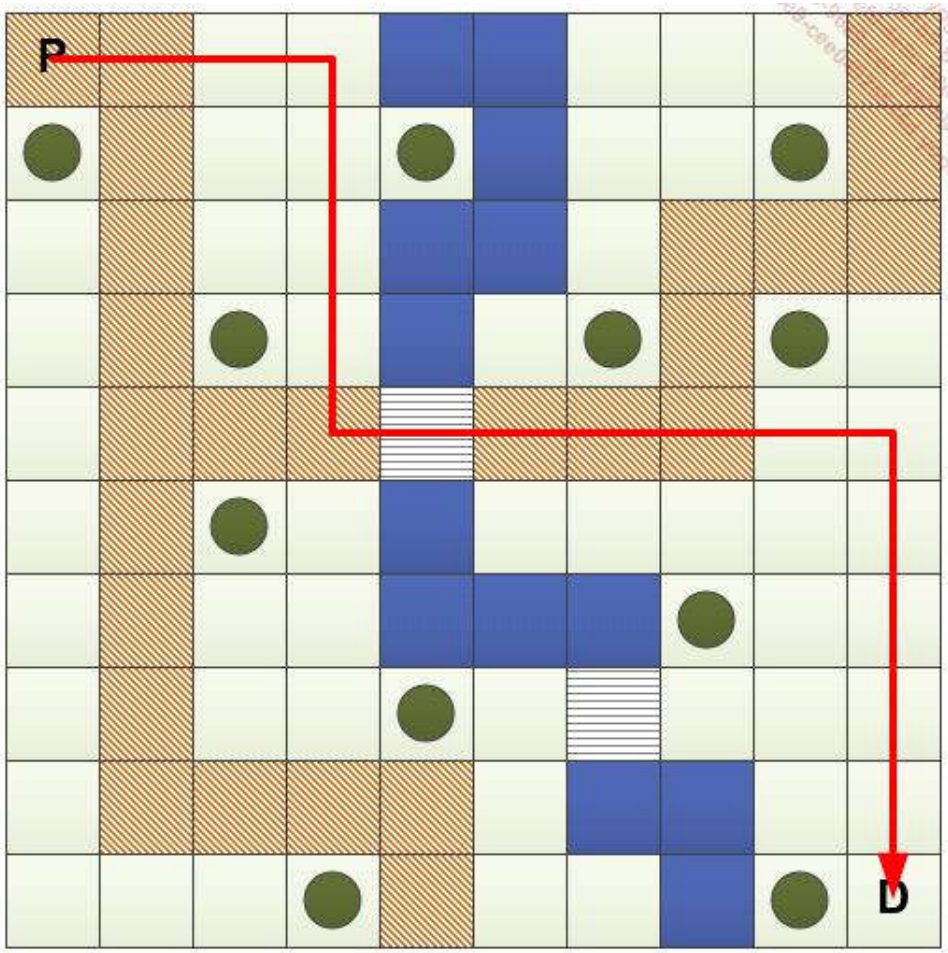


En la nueva casilla, volvemos a intentar recorrer las rutas en nuestro orden de prioridad: a la derecha (imposible porque encontramos un árbol), y a continuación hacia abajo.

Seguimos con las casillas situadas debajo hasta que podemos ir hacia la derecha de nuevo.

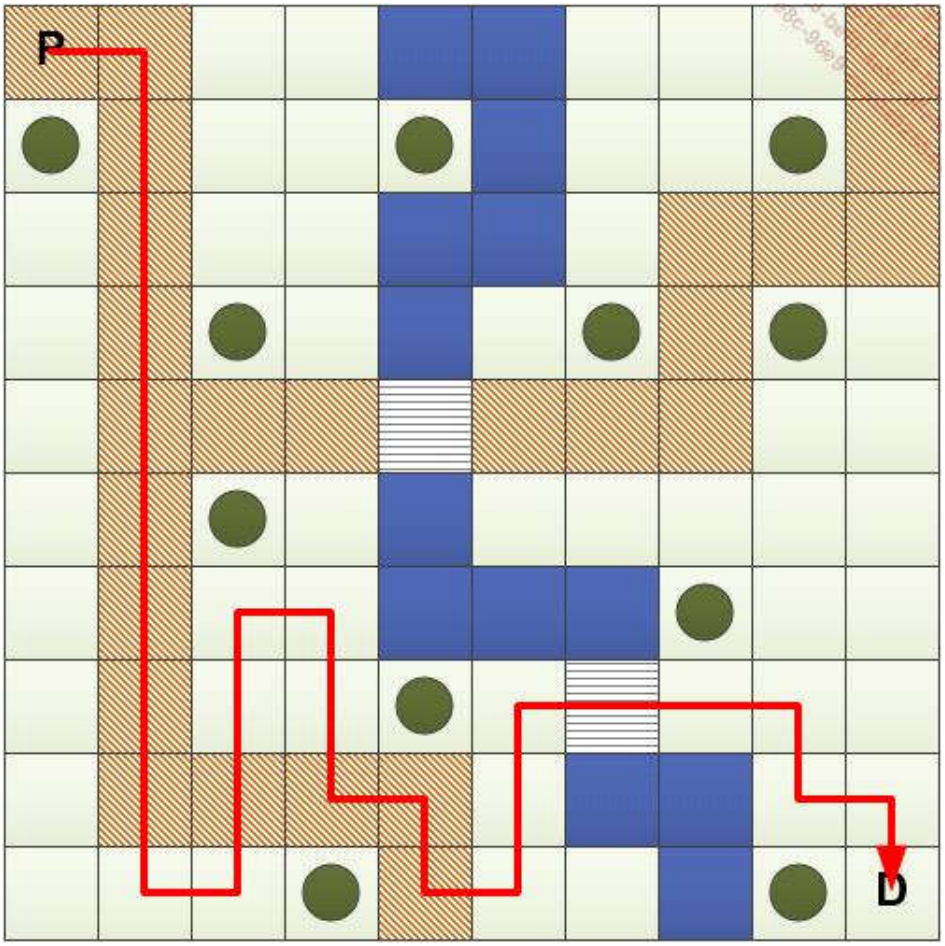


Como llegamos al final del mapa y es imposible ir más hacia la derecha, probamos con la segunda dirección posible, es decir, hacia abajo. Encontramos el destino.



Ahora que hemos encontrado un camino, podemos calcular su longitud. En efecto, el recorrido en profundidad no tiene en cuenta los pesos de las distintas rutas. Aquí, encontramos un camino que costaría 32 puntos de acción. No se trata del camino más corto, que cuesta únicamente 27 puntos.

Además, hemos tenido "suerte" seleccionando el orden de recorrido de las casillas vecinas. En efecto, si hubiéramos escogido el orden "abajo - arriba - derecha - izquierda", habríamos obtenido el siguiente camino:



Su longitud es de 44, lo cual resulta muy superior al camino encontrado antes.

2. Búsqueda en anchura

La búsqueda en anchura es la que utiliza por defecto la policía para encontrar a un criminal. Se parte del último punto donde se le haya visto, y a continuación se va a intentar avanzar progresivamente en círculos concéntricos.

a. Principio y pseudocódigo

Empezamos en el nodo de partida y comprobamos si el destino está en un nodo inmediatamente adyacente. Si no es el caso, intentamos comprobar los vecinos de estos nodos en un orden fijo, y así sucesivamente.

Gráficamente, vamos a alejarnos de forma progresiva del punto de partida, comprobando todas las casillas en un radio determinado. Se recorre toda la anchura del árbol cada vez, mientras que en el algoritmo anterior intentábamos ir lo más lejos posible en primer lugar (de ahí su nombre de búsqueda en profundidad).

La búsqueda en anchura permite comprobar nivel por nivel los distintos nodos accesibles. Se aleja progresivamente del punto de partida. Si todos los arcos tuvieran el mismo peso, encontraríamos el camino óptimo; en caso contrario nada permite saber si el camino encontrado es el más corto.

Además, este algoritmo no es, en absoluto, eficaz. En efecto, corre el riesgo de probar muchos nodos antes de encontrar una ruta, puesto que no utiliza ninguna información acerca de ellos.

Es, no obstante, bastante fácil de implementar. A diferencia de la búsqueda en profundidad, que utiliza una pila, esta utiliza una fila. El resto del algoritmo es, sin embargo, el mismo.

La fila es otra estructura algorítmica. Se denomina FIFO, del inglés "First In, First Out" (el primero en llegar es el primero en salir). En la vida cotidiana encontramos filas de espera por todas partes: el primero que llega realiza la cola, hasta poder pasar (por caja, por ejemplo) y será el primero en salir. Se dice que se encola un elemento (agregándolo al final) y se desencola cuando se recupera el elemento situado al principio.

El pseudocódigo es, por lo tanto, el siguiente (las líneas que difieren respecto a la búsqueda en profundidad se marcan en negrita):

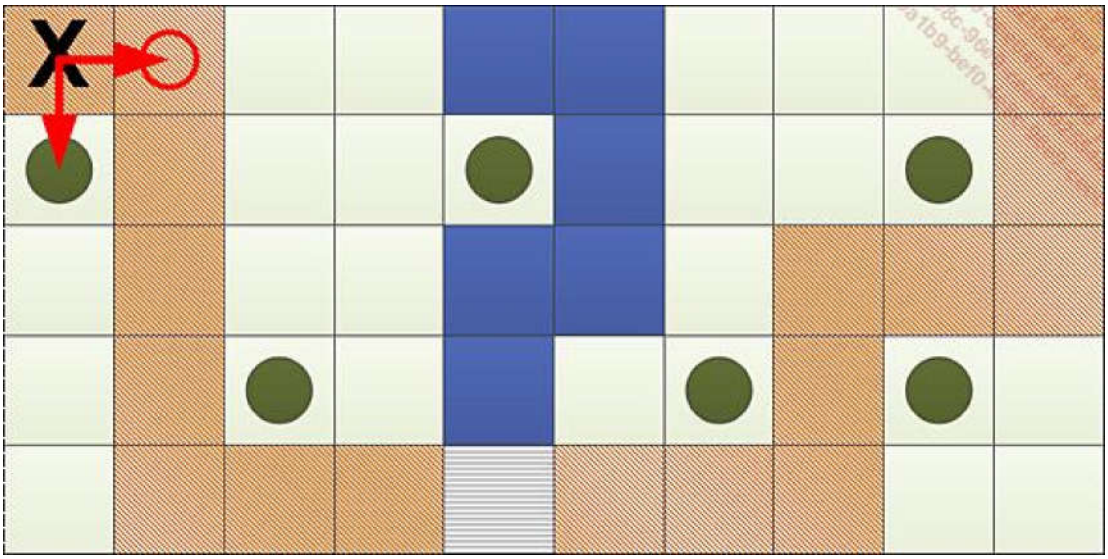
```
// Inicialización del array
Crear array Precursor
Para cada nodo n
   Precursor[n] = null

// Creación de la lista de nodos no visitados, y de la pila
Crear lista NodosNoVisitados = conjunto de nodos
Crear pila PorVisitar
PorVisitar.Apilar(inicio)

// Bucle principal
Mientras PorVisitar no vacío
   Nodo encurso = PorVisitar.Desapilar
   Si encurso = destino
          Fin (OK)
   Si no
          Para cada vecino v de n
                 Si v en NodosNoVisitados
                        Extraer v de NodosNoVisitados
                        Precursor[v] = n
                        PorVisitar.Apilar(v)
```

Los vecinos deben encolarse en el orden seleccionado esta vez.

b. Aplicación al mapa

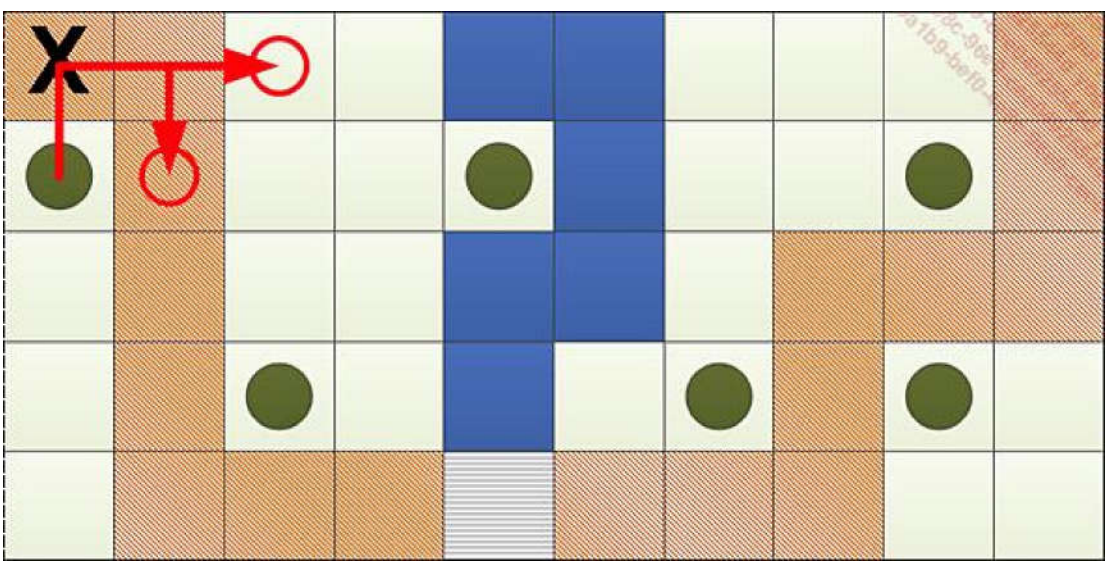
La aplicación al mapa es algo más compleja que para la búsqueda en profundidad. Comenzamos en nuestro punto de partida y consultamos si el punto de destino está alrededor. Solo mostramos el principio del mapa para las primeras etapas.

La casilla marcada con una X es nuestro punto de partida. La casilla marcada con un círculo se agrega a la fila.

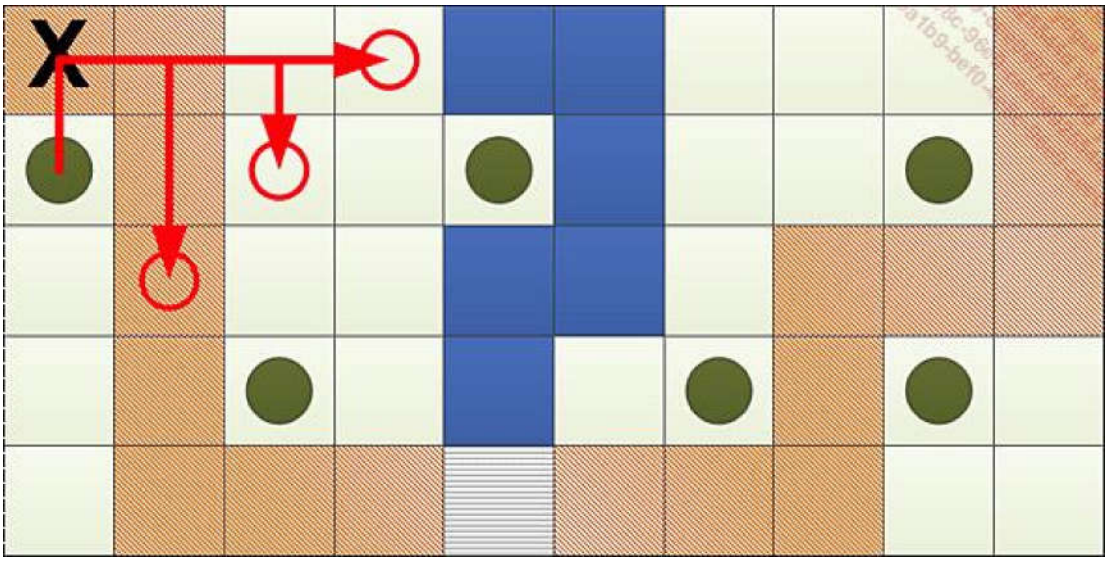


En la primera etapa, ningún vecino es el punto de llegada. Además, la casilla situada al sur es un árbol, y resulta imposible ir a ella. Solo es posible agregar la casilla derecha a la fila.

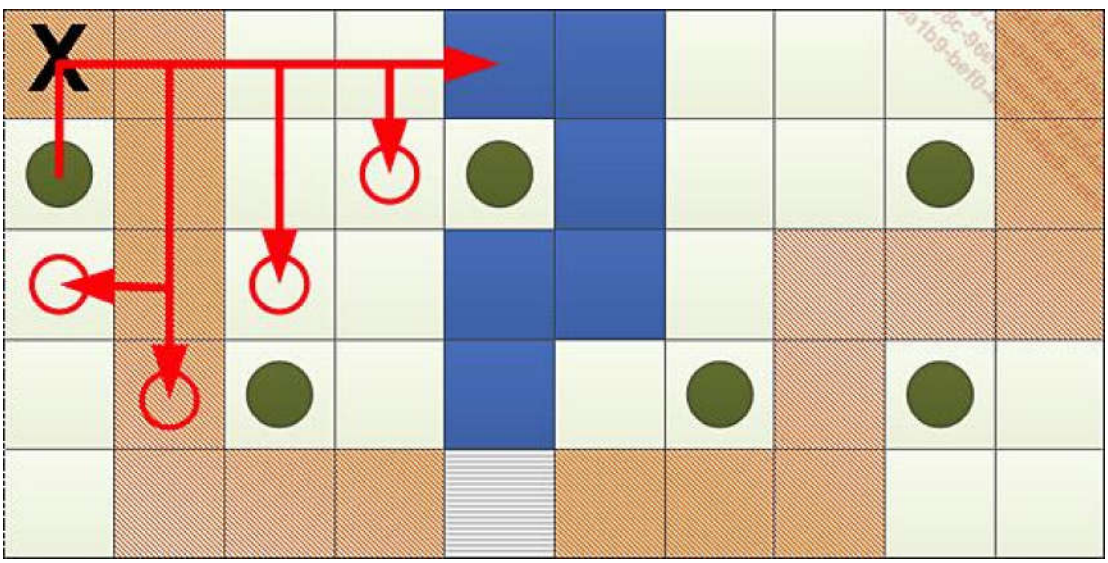
Partimos, a continuación, de esta casilla y consultamos los nodos vecinos. Se agregan dos casillas a la fila.



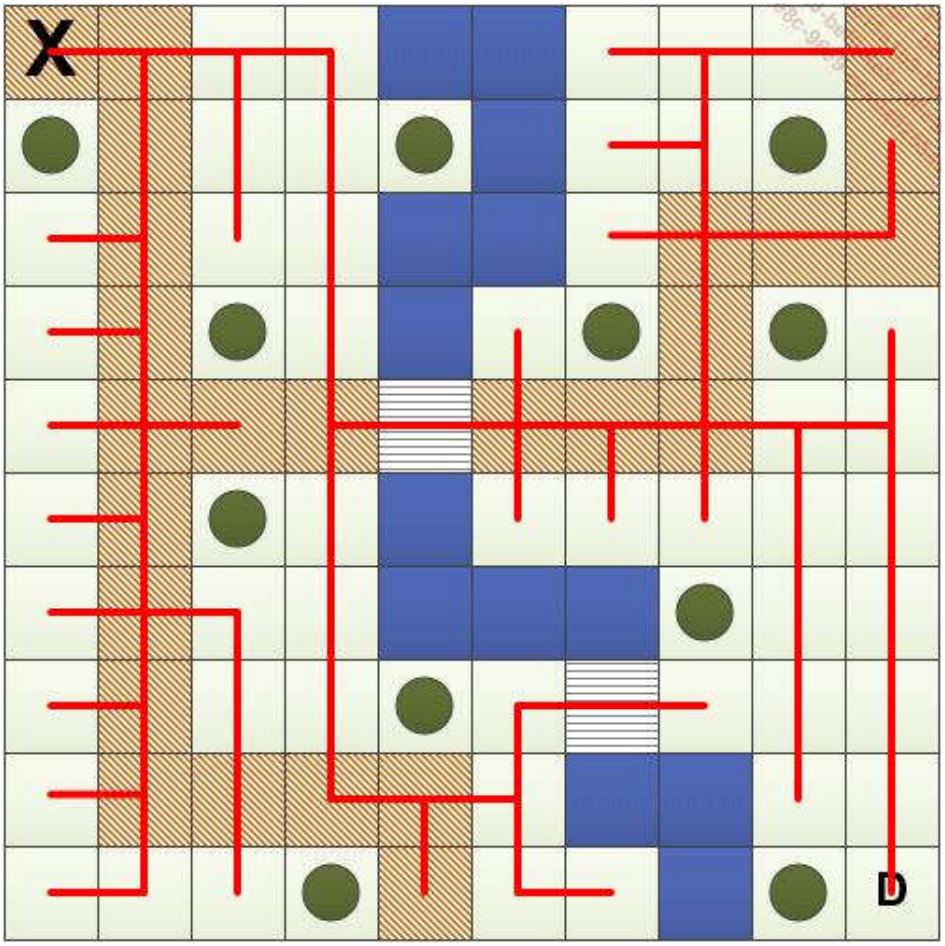
Se continúa, extendiendo poco a poco nuestra zona de búsqueda:



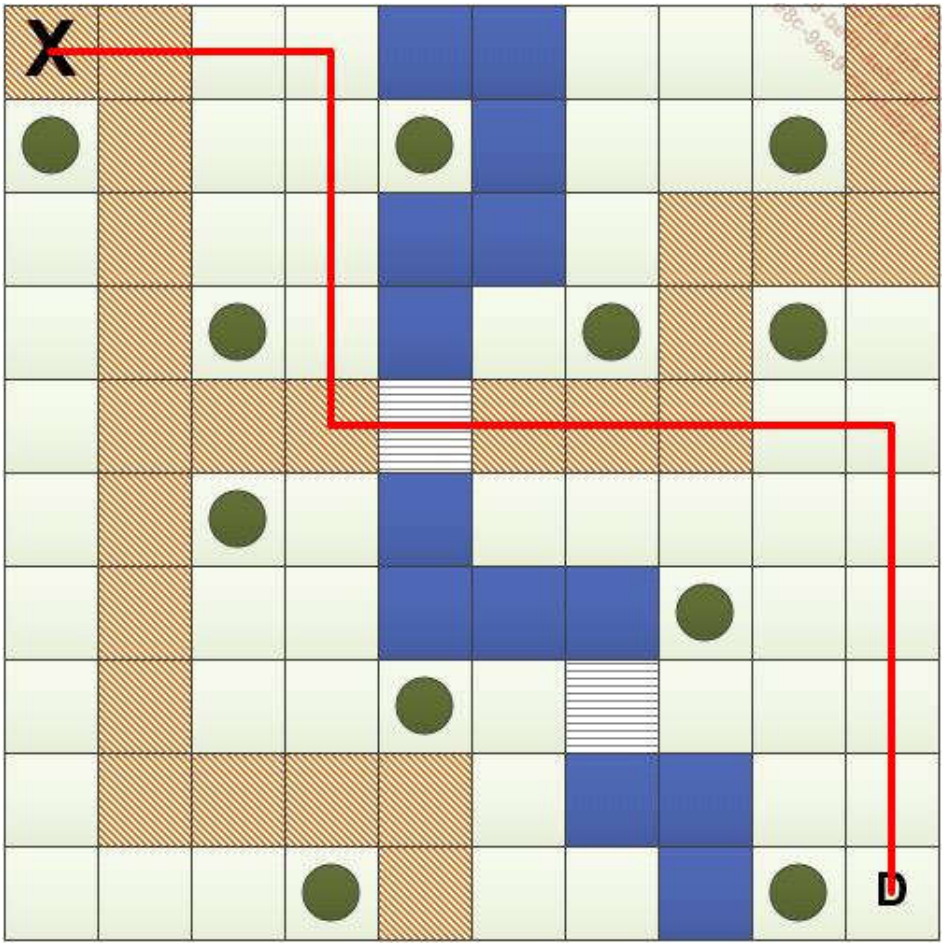
En la siguiente iteración, la primera casilla va a encontrarse con el agua. Como es infranqueable, no se agrega en la fila de casillas por recorrer.



Continuamos así hasta el destino:



Conservamos el camino que nos ha permitido encontrar el destino, representado en la figura siguiente. Su tamaño es de 32, lo cual no es óptimo (que sería 27).



Podemos ver también que, cuando alcanzamos el destino, todas las casillas se han explorado, lo cual supone muchas más etapas. El algoritmo no es, por tanto, eficaz en términos de rendimiento.

Algoritmos inteligentes

Las búsquedas en profundidad y en anchura no permiten encontrar el camino más corto, aunque sí el primero que permite alcanzar el punto de destino desde el punto de partida.

Existen otros algoritmos que permiten determinar el camino más corto, o al menos un camino optimizado, sin tener que comprobar necesariamente todas las rutas posibles.

1. Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford permite encontrar el camino más corto, si existe. No es el óptimo, pero es el que funciona en la mayoría de los casos. En efecto, permite definir longitudes negativas para los arcos, y no solamente positivas.

Además, si hay algún circuito cuya longitud sea negativa (y, por tanto, que permita disminuir el peso total), es capaz de detectarlo. Esto es importante, pues en este caso no existe ruta más corta.

a. Principio y pseudocódigo

Este algoritmo va a utilizar la matriz de longitudes. Su funcionamiento es iterativo.

Al principio, se inicializa a +∞ la longitud mínima de cada nodo (lo que significa que no se ha encontrado todavía ningún camino hasta allí desde el destino). Se va a guardar, también, para cada nodo el nodo anterior (el que permite llegar a él de la manera óptima en longitud) e inicializarlo con un nodo vacío.

Se aplica, a continuación, tantas veces como número de nodos menos 1 el mismo bucle. De este modo, si tenemos 7 nodos, se aplica 6 veces.

Con cada iteración, se sigue cada arco (u,v). Se calcula la distancia desde el punto de partida hasta v como la distancia del inicio hasta u más la longitud del arco (u,v). Se obtiene también una nueva longitud para ir hasta el nodo v que comparamos con la que ya tenemos registrada. Si esta longitud es mejor que la obtenida hasta el momento, se cambia la longitud de este nodo y se indica que su predecesor es ahora u.

Solo deben utilizarse los arcos que parten de un nodo cuya distancia calculada es diferente de +∞. En efecto, en caso contrario se guarda una distancia infinita, que no puede mejorar las rutas que ya se han encontrado.

Si en una iteración no hay ningún cambio, puesto que ningún arco permite encontrar una distancia menor que la conocida, podemos detener prematuramente el algoritmo.

Además, si se aplica el algoritmo tantas veces como números de nodos del grafo y se realiza alguna modificación en los pesos, entonces se sabe que existe un circuito de tamaño negativo y no es posible resolver el problema.

El pseudocódigo es el siguiente:

```
// Inicialización de los arrays
Crear array Longitud_min
Crear array Precursor

Para cada nodo n
   Longitud_min[n] = +8
```

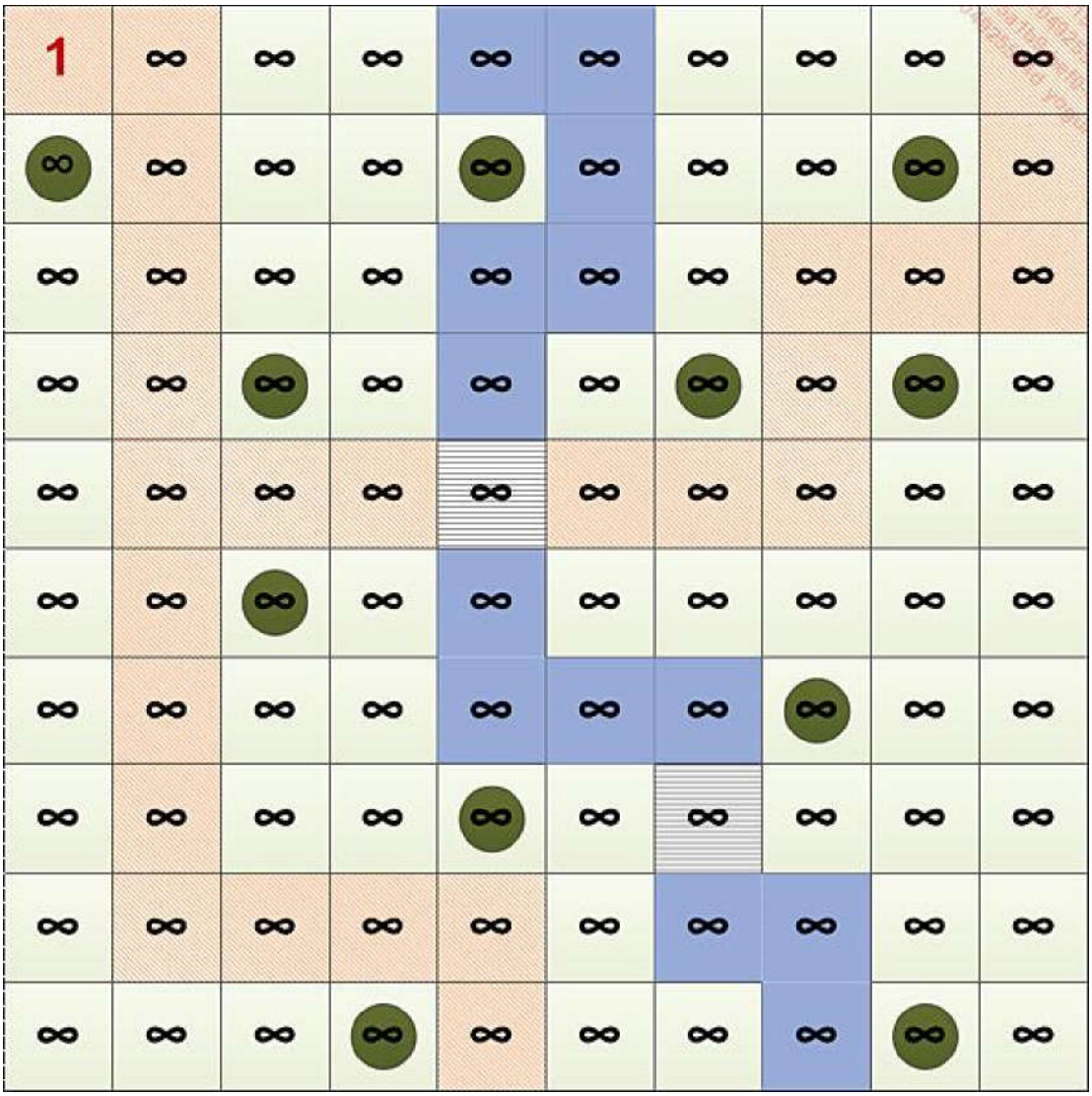
```
   Precursor[n] = null
Longitud_min[inicio] = 0

// Bucle principal
Para i de 1 hasta el número de nodos - 1
   Cambios = FALSO
   Para cada arco (u, v)
          Si Longitud_min[u] + longitud(u, v) < Longitud_min[v]
                 // Hemos encontrado un camino más corto
                 Longitud_min[v] = Longitud_min[u]+longitud(u, v)
                 Precursor[v] = u
                 Cambios = VERDADERO
   Si Cambios = FALSO
          // Ningún cambio: hemos terminado
          Fin (OK)

Para cada arco (u,v)
   Si Longitud_min[u] + longitud(u, v) < Longitud_min[v]
          // No podemos resolver este problema
          Fin (Error: existen bucles negativos)
```

b. Aplicación al mapa

Para comprender mejor el algoritmo, lo aplicaremos a nuestro mapa con nuestro explorador, que busca el punto de destino.

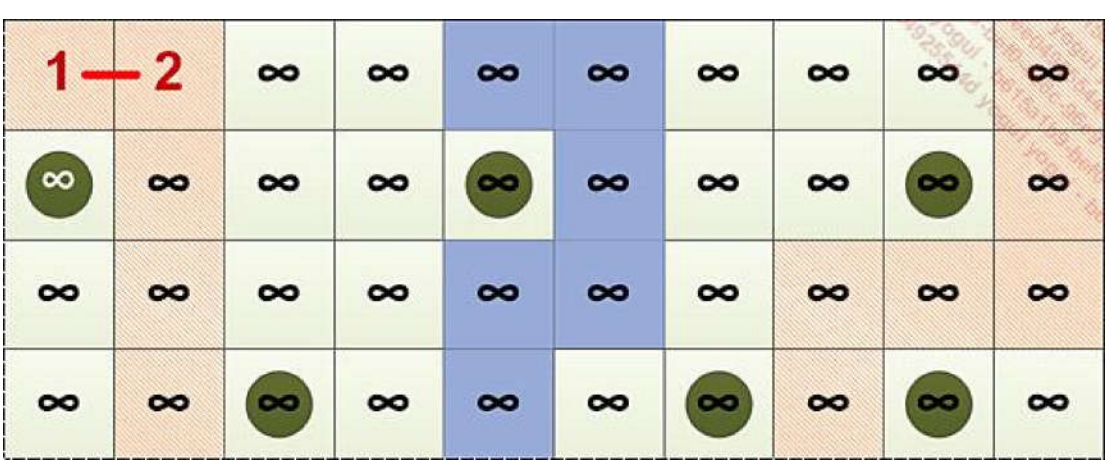
Iniciaremos todas las distancias para llegar a una casilla cuyo valor sea +∞. Solo la casilla de partida posee una distancia diferente, igual a 1 (puesto que es un camino cuyo coste es 1).

Se obtiene el siguiente estado inicial:

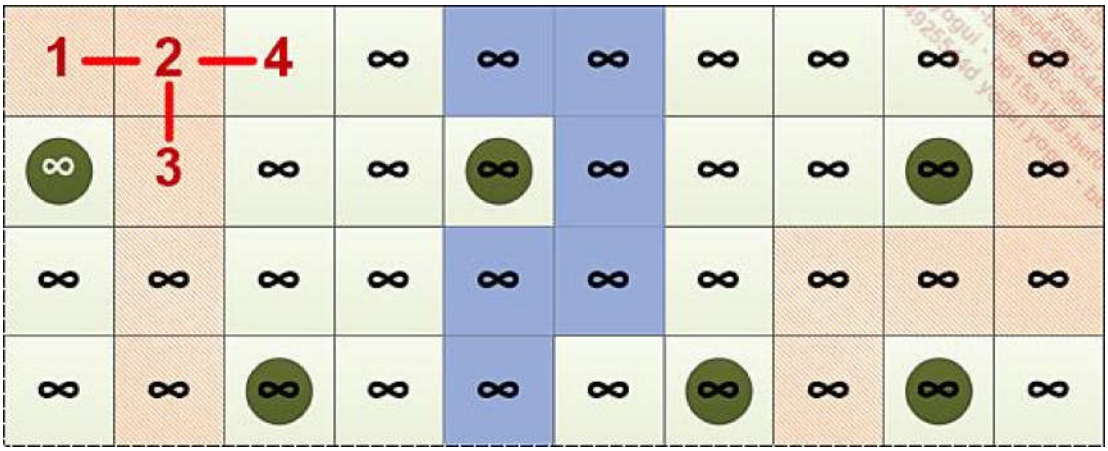


Tenemos 100 casillas, así que como máximo tendremos que realizar 99 iteraciones para encontrar el camino más corto.

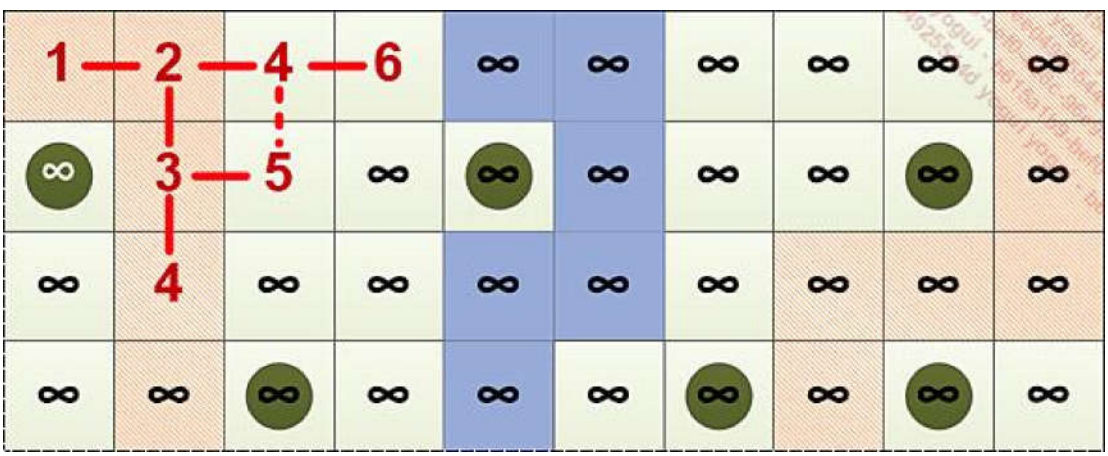
En la primera iteración, tenemos únicamente dos arcos que aplicar: son los que salen de la casilla de partida y que permiten ir a la derecha y hacia abajo. El arco para ir a la derecha vale 1 (es un camino), mientras que el que permite desplazarse hacia el sur tiene una longitud de $+\infty$ (puesto que el árbol es infranqueable). Actualizamos la segunda casilla para indicar que su nueva distancia respecto al origen es igual a 2, y que su precursor es la primera casilla.



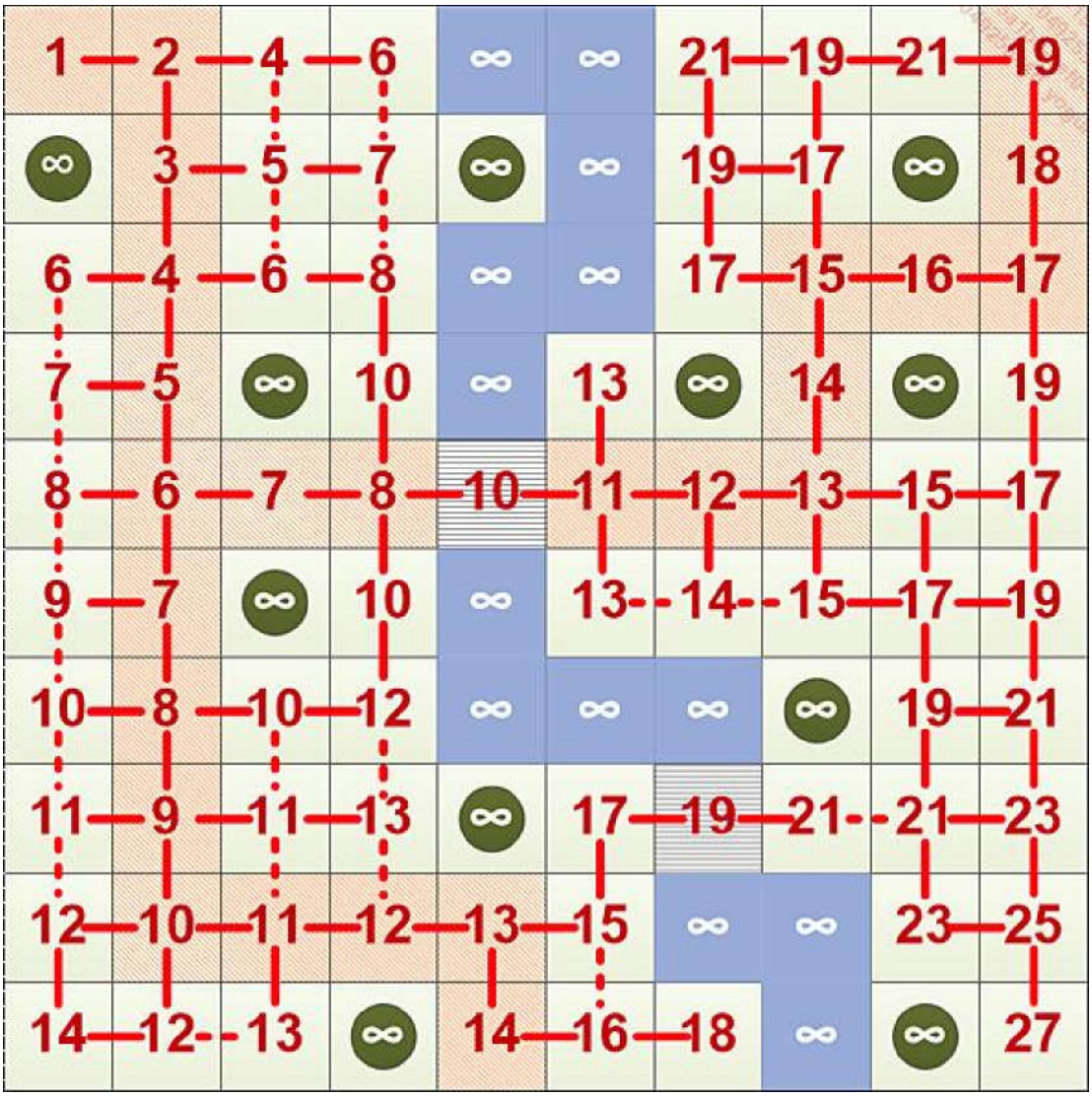
En la segunda iteración, vamos a aplicar los arcos que salen de las dos primeras casillas. Encontramos:



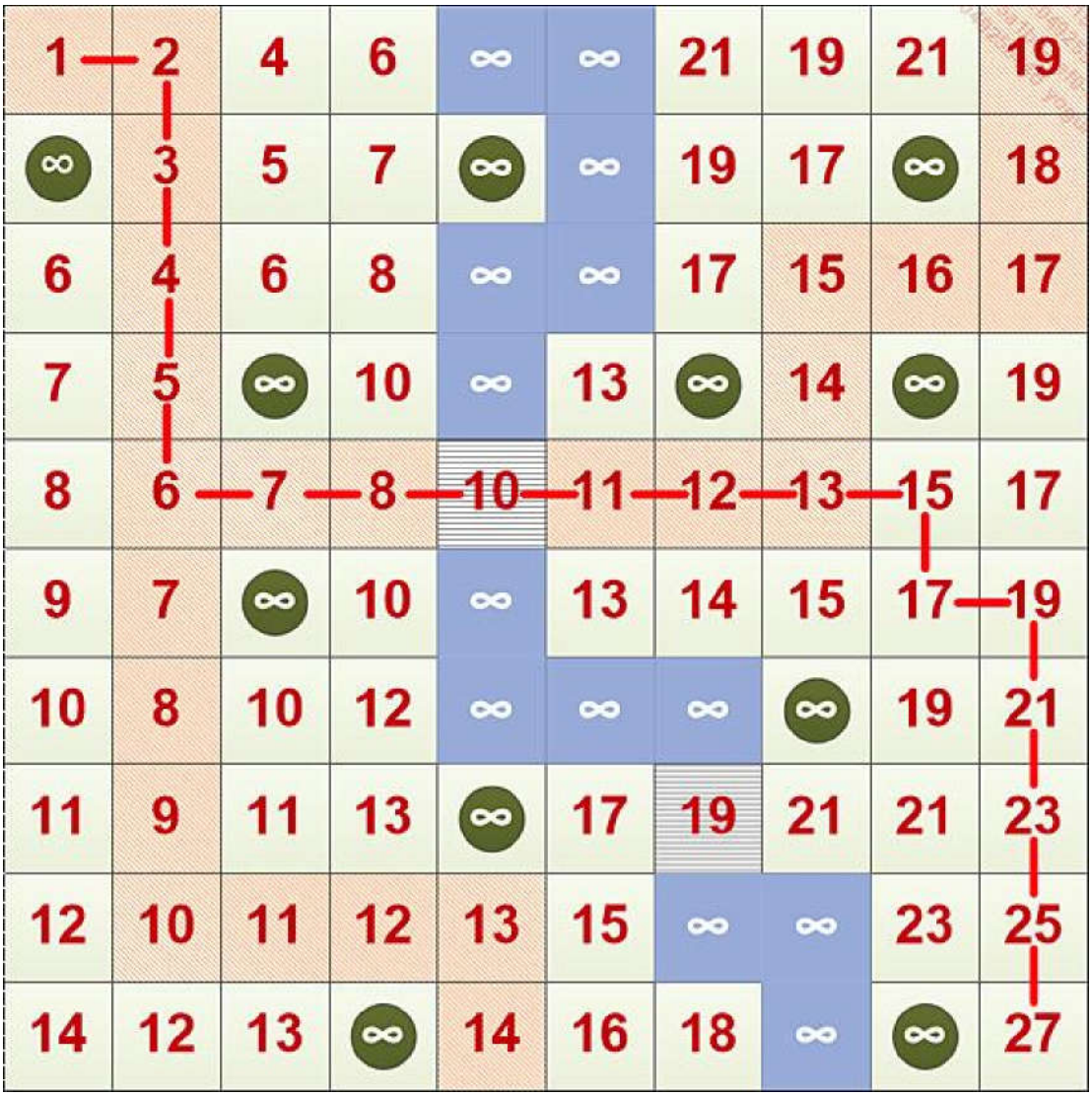
En la siguiente iteración, vamos a encontrar dos rutas para ir a la casilla al sur de 4: una que va de 4 hasta esta casilla cubierta por hierba, y otra que pasa por un camino. Conservaremos la ruta desde el camino, que es más corta (5 frente a 6 puntos de acción). La que no se conserva la marcaremos con un trazo discontinuo.



Continuamos con las siguientes iteraciones (unas veinte), hasta alcanzar la casilla de destino:



Reconstruimos a continuación un recorrido que nos permita encontrar el camino más corto que tiene un coste, efectivamente, de 27 puntos de acción.



Existen varios recorridos que nos dan caminos de tamaño 27. Hemos escogido uno arbitrariamente, remontando los sucesivos predecesores.

El algoritmo no es eficaz, puesto que se aplican de nuevo todos los arcos utilizados anteriormente. Además, todos los caminos más cortos se calculan, y no solo el que nos interesa.

2. Algoritmo de Dijkstra

El algoritmo de Dijkstra supone una mejora del algoritmo de Bellman-Ford. Funciona solo si todas las distancias son positivas, como es generalmente el caso en problemas reales.

Permite escoger de manera más inteligente el orden de aplicación de las distancias, y sobre todo en lugar de aplicarse sobre los arcos se aplica sobre los nodos, y no vuelve nunca atrás. De este modo, cada arco se aplica una única vez.

a. Principio y pseudocódigo

En primer lugar inicializamos dos arrays, como con el algoritmo de Bellman-Ford: uno que contiene las distancias desde el nodo inicial (a +∞, salvo para el nodo de partida, a 0), y otro que contiene los precursores (todos vacíos).

Buscamos, a continuación, con cada iteración, el nodo que no ha sido visitado todavía y cuya distancia sea la menor respecto al nodo de partida. Se aplica, a continuación, todos los arcos que salen y se modifican las distancias menores si se encuentran (así como el precursor para llegar).

Se repite la operación hasta haber agotado todos los nodos, o bien hasta llegar al nodo de destino.

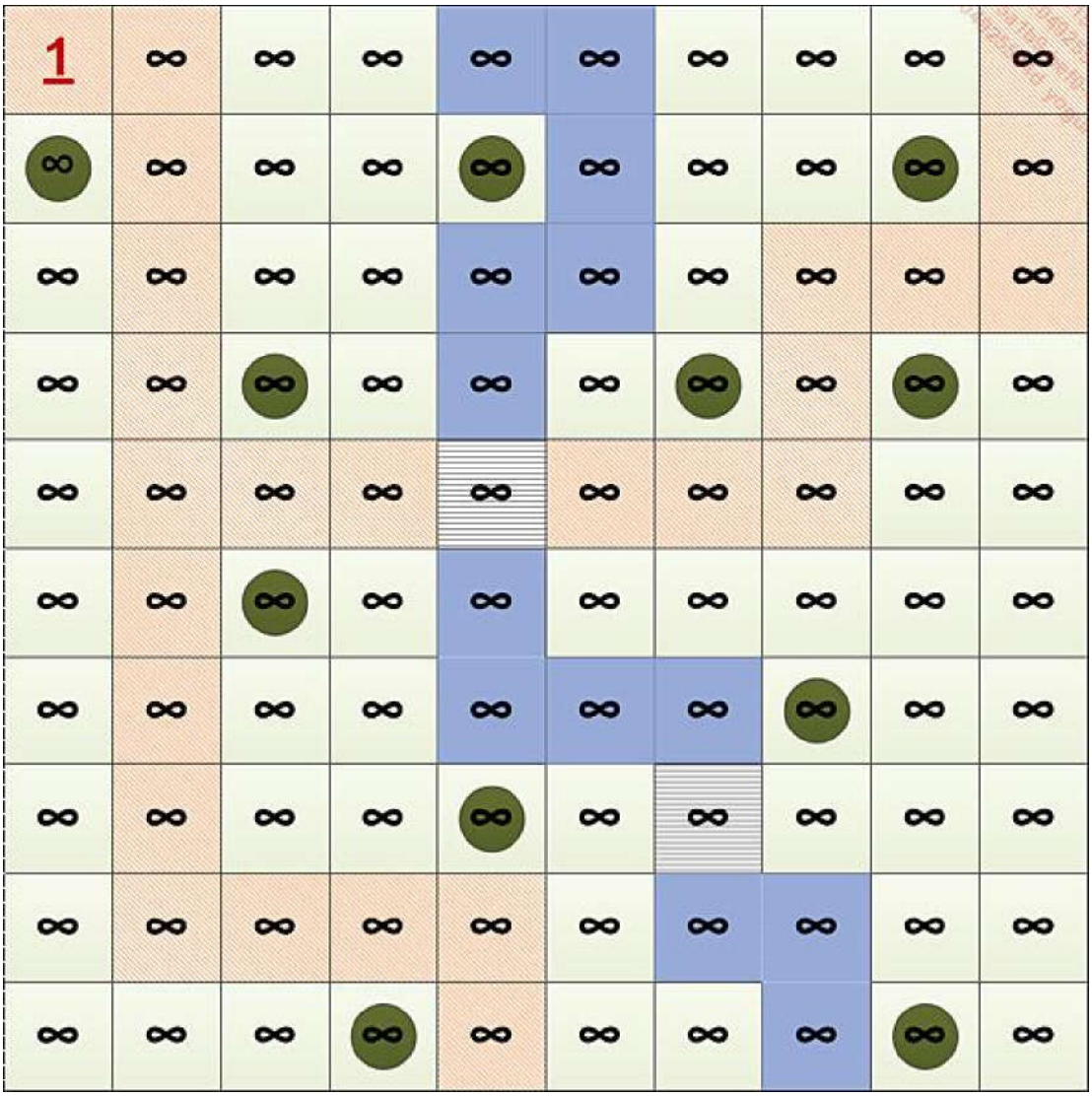
El pseudocódigo correspondiente es:

```
// Inicialización de los arrays
Crear array Longitud_min
Crear array Precursor

Para cada nodo n
   Longitud_min[n] = +8
   Precursor[n] = null
Longitud_min[inicio] = 0

Lista_no_visitados = conjunto de nodos

// Bucle principal
Mientras Lista_no_visitados no vacío
   u = nodo en Lista_no_visitados cuya Longitud_min[u] sea min
   Para cada arco (u, v)
         Si Longitud_min[u] + longitud(u, v) < Longitud_min[v]
                // Se ha encontrado un camino más corto
                Longitud_min[v] = Longitud_min[u]+longitud(u, v)
                Precursor[v] = u
   Eliminar u de Lista_no_visitados
   Si u = destino
         Fin (OK)
```

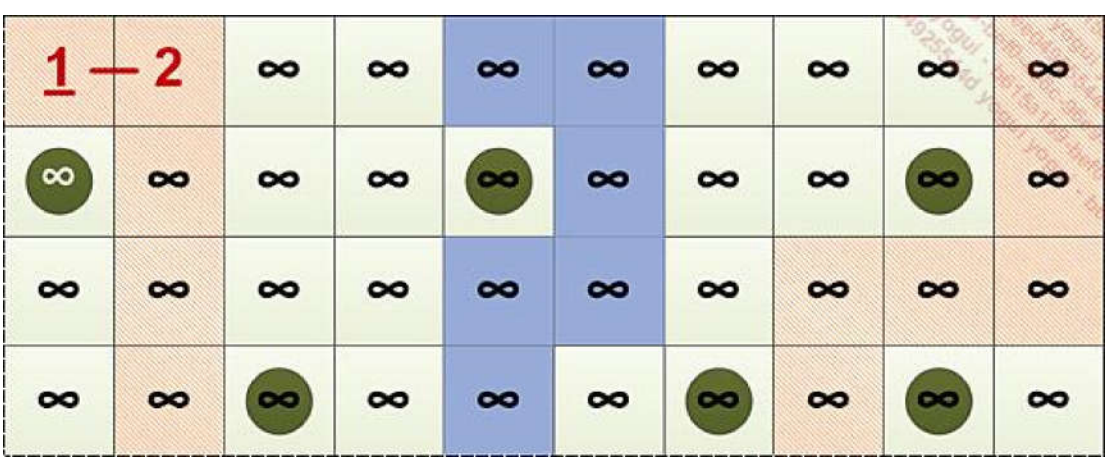
La principal diferencia respecto a Bellman-Ford es la aplicación de las distancias nodo a nodo una única vez, lo que aporta una gran optimización en términos de tiempo y de cálculos realizados.

b. Aplicación al mapa

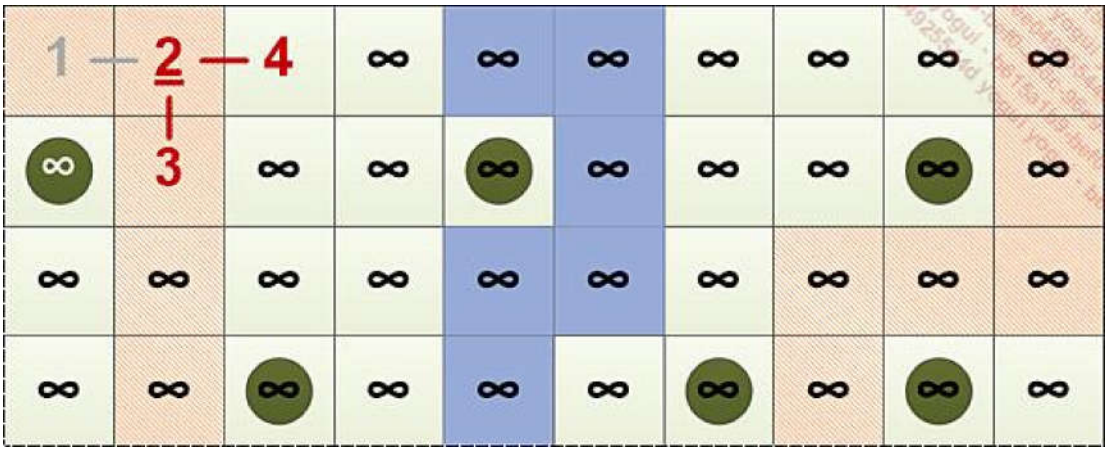
Aplicaremos ahora nuestro algoritmo a nuestro mapa en 2D. Las casillas ya visitadas se marcan en gris. La casilla en curso se subraya. La inicialización es la misma que para Bellman-Ford:



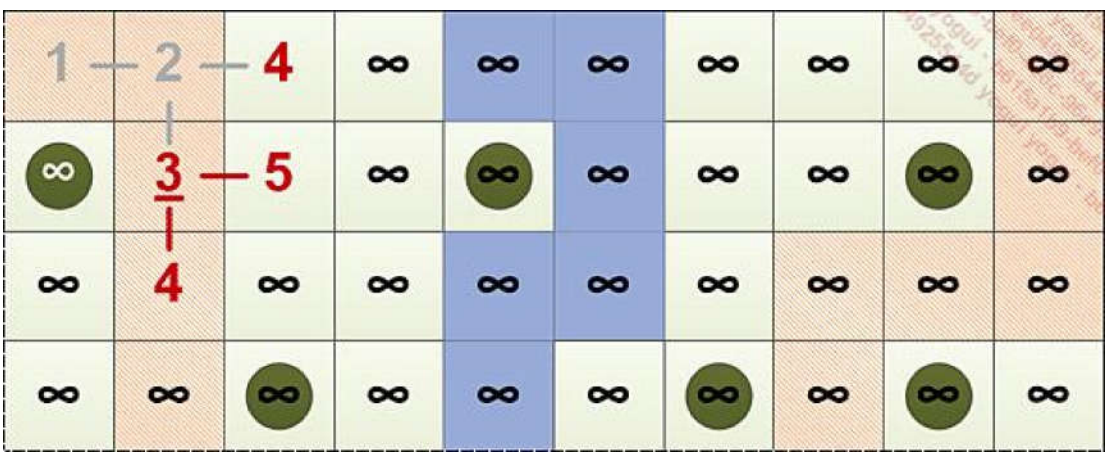
La primera casilla está seleccionada, puesto que es la única que contiene una distancia no nula. Se aplica, por tanto, a este nodo, y se encuentra la distancia a la casilla de la derecha (la casilla inferior contiene un árbol, de modo que es imposible de alcanzar).



La primera casilla se ha visitado, de modo que es la segunda la que sirve de punto de partida para la siguiente iteración:

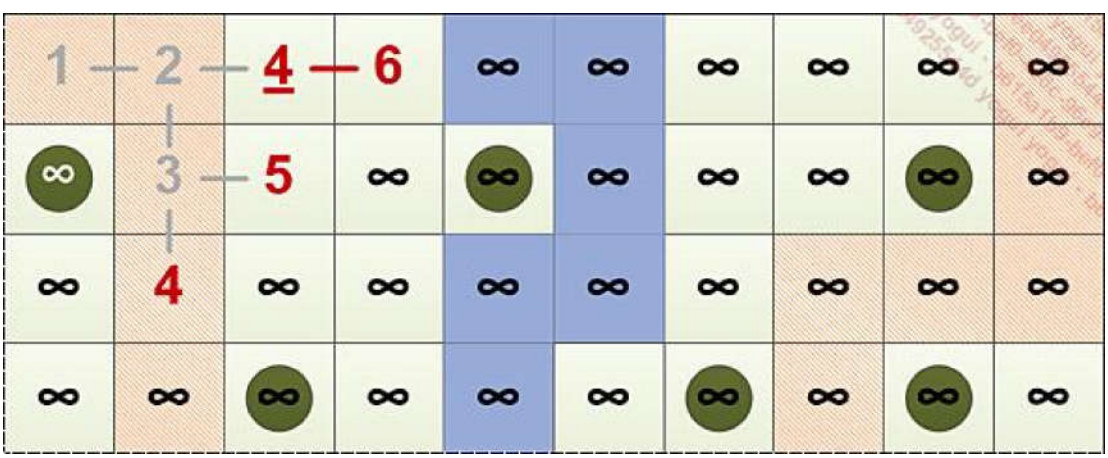


La casilla con la menor distancia al origen es la que contiene un 3 en coste del camino. Partiremos a continuación desde esta casilla.

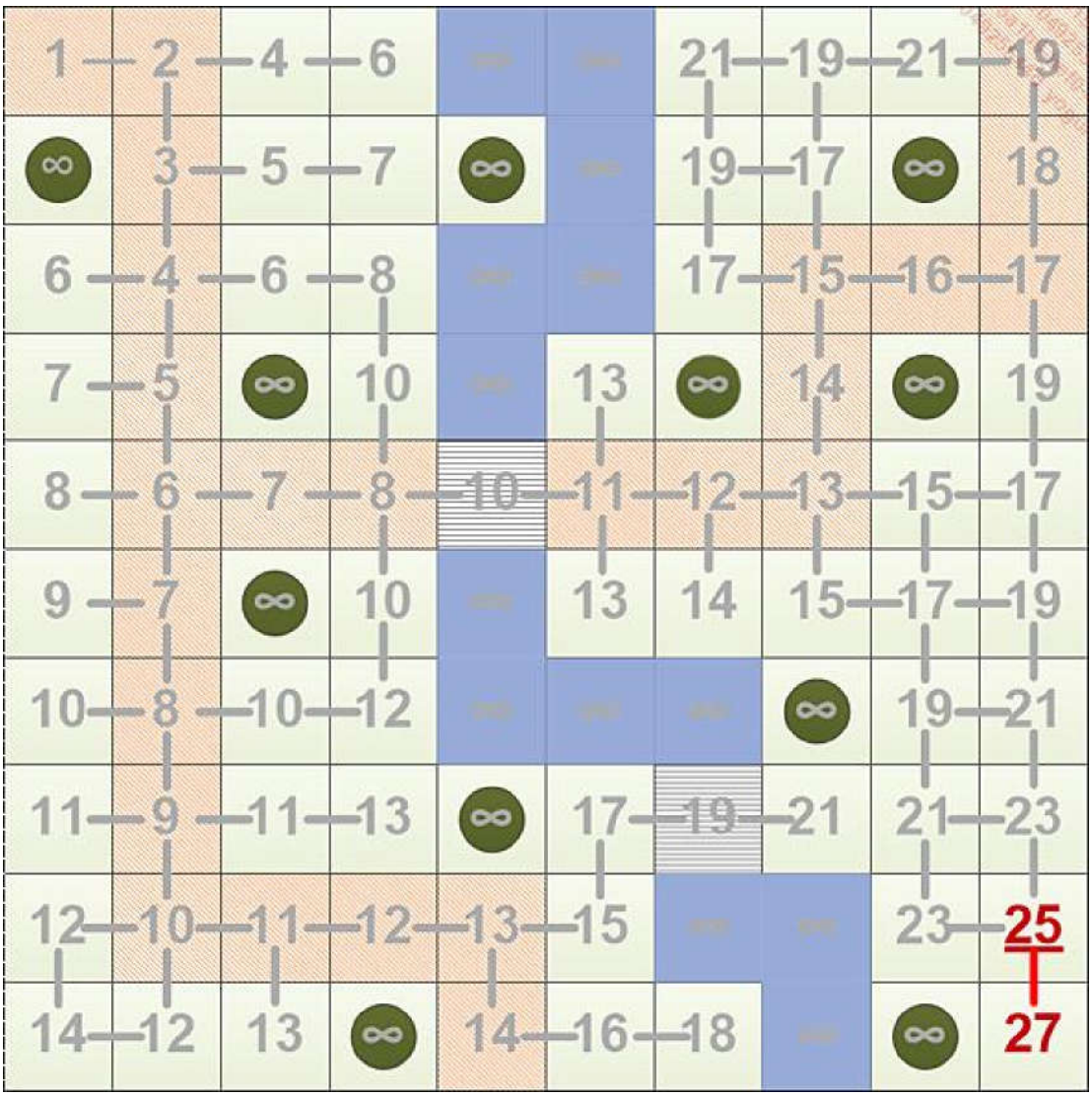


En este caso, tenemos dos casillas con una distancia igual a 4. Se aplicarán, por lo tanto, los arcos que salen en el orden de lectura (es decir, a la izquierda, a la derecha, y luego arriba y abajo). Aplicamos en primer lugar el que está situado más al norte. Como pasando por el norte llegamos a la casilla marcada actualmente con un 5 con 6 puntos de acción, ignoramos el arco.

Tenemos la siguiente situación:



Seguimos con las iteraciones hasta encontrar la casilla de destino. Tenemos la siguiente situación:



Obtenemos exactamente los mismos resultados que con Bellman-Ford, pero realizando muchos menos cálculos, puesto que cada arco se ha aplicado (y por tanto calculado las distancias) una única vez. Los caminos encontrados son los mismos.

3. Algoritmo A*

El algoritmo A* (pronunciado "A estrella", o en inglés "A star") funciona sobre grafos en los que todas las distancias sean positivas (como con Dijkstra).

A diferencia de los dos algoritmos anteriores, A* no comprueba todas las rutas, y no puede asegurar que el camino encontrado sea el más corto. Se trata simplemente de una aproximación, que devuelve por lo general el mejor camino o uno de los mejores. Existen casos, sin embargo, como por ejemplo los laberintos, en los que A* es muy poco o incluso nada eficaz. En espacios abiertos con pocos obstáculos, como ocurre con nuestro mapa, presenta por el contrario muy buen rendimiento.

a. Principio y pseudocódigo

El algoritmo A* utiliza información relativa al lugar donde se encuentra el objetivo para seleccionar la siguiente dirección que hay que seguir. En efecto, se basa en el hecho de que la distancia más corta es, generalmente, la línea recta, y no hacer demasiados giros.

Este es el algoritmo que utilizamos de manera intuitiva cuando estamos perdidos en una ciudad: buscamos en qué dirección debemos ir e intentamos seguir esa dirección. Si no es posible (por ejemplo, porque no hay camino), entonces lo variamos ligeramente sin desviarnos demasiado para poder volver hacia nuestro objetivo.

Nos hace falta una manera de estimar la distancia restante entre cada nodo y el destino. Cuanto más precisa sea esta aproximación, más lo serán los resultados. Por el contrario, no debe sobrestimar jamás la distancia, sino subestimarla o devolver la distancia exacta. Nada indica en el algoritmo cómo seleccionar esta distancia. Depende de cada problema.

Como con los algoritmos anteriores, guardamos la distancia mínima para llegar a un nodo y su predecesor. Además, se almacena la lista de nodos que todavía no se han visitado, y para cada nodo se guarda la distancia estimada al destino.

A continuación, entre los nodos que todavía no se han visitado, buscamos el que indica una distancia total menor. Se corresponde con la suma de la distancia desde el origen más la distancia estimada al destino. Obtenemos así el nodo más prometedor. Aplicamos, a continuación, los arcos que salen de él, lo marcamos como visitado y continuamos.

```
// Inicialización de los arrays
Crear array Longitud_min
Crear array Precursor
Crear array DistanciaEstimada

Para cada nodo n
   Longitud_min[n] = +∞
   Precursor[n] = null
   DistanciaEstimada[n] = heurístico a calcular
Longitud_min[inicio] = 0

Lista_no_visitados = conjunto de nodos

// Bucle principal
Mientras Lista_no_visitados no vacía
   u = nodo en Lista_no_visitados o Longitud_min[u]
+DistanciaEstimada[u] es min
   Eliminar u de Lista_no_visitados
   Para cada arco (u, v)
          Si Longitud_min[u] + longitud(u, v) < Longitud_min[v]
                // Encontramos un camino más corto
                Longitud_min[v] = Longitud_min[u]+longitud(u, v)
                Precursor[v] = u

   Si u = destino
          Fin (OK)
```

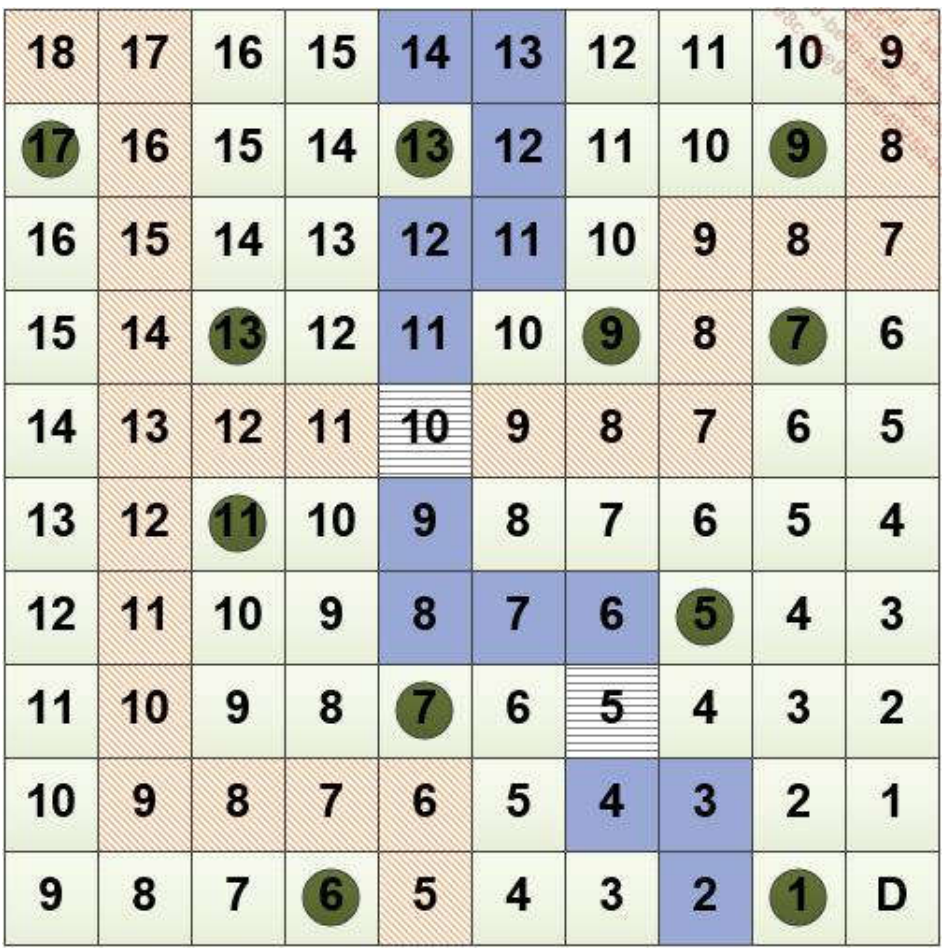
En el peor caso, el algoritmo A* es equivalente al algoritmo de Dijkstra.

b. Aplicación al mapa

En nuestro mapa, la distancia de aproximación seleccionada es la distancia de Manhattan. Está definida por el número de casillas horizontales sumada al número de casillas verticales que nos separan del destino.

Se trata, necesariamente, de una cantidad subestimada. En efecto, supone que todas las casillas tienen un costo de 1 punto de acción, aunque los puentes y los prados tienen un coste de 2, y ciertas casillas son infranqueables (y por lo tanto tienen una distancia infinita). La distancia seleccionada está, por tanto, adaptada y subestima la distancia real.

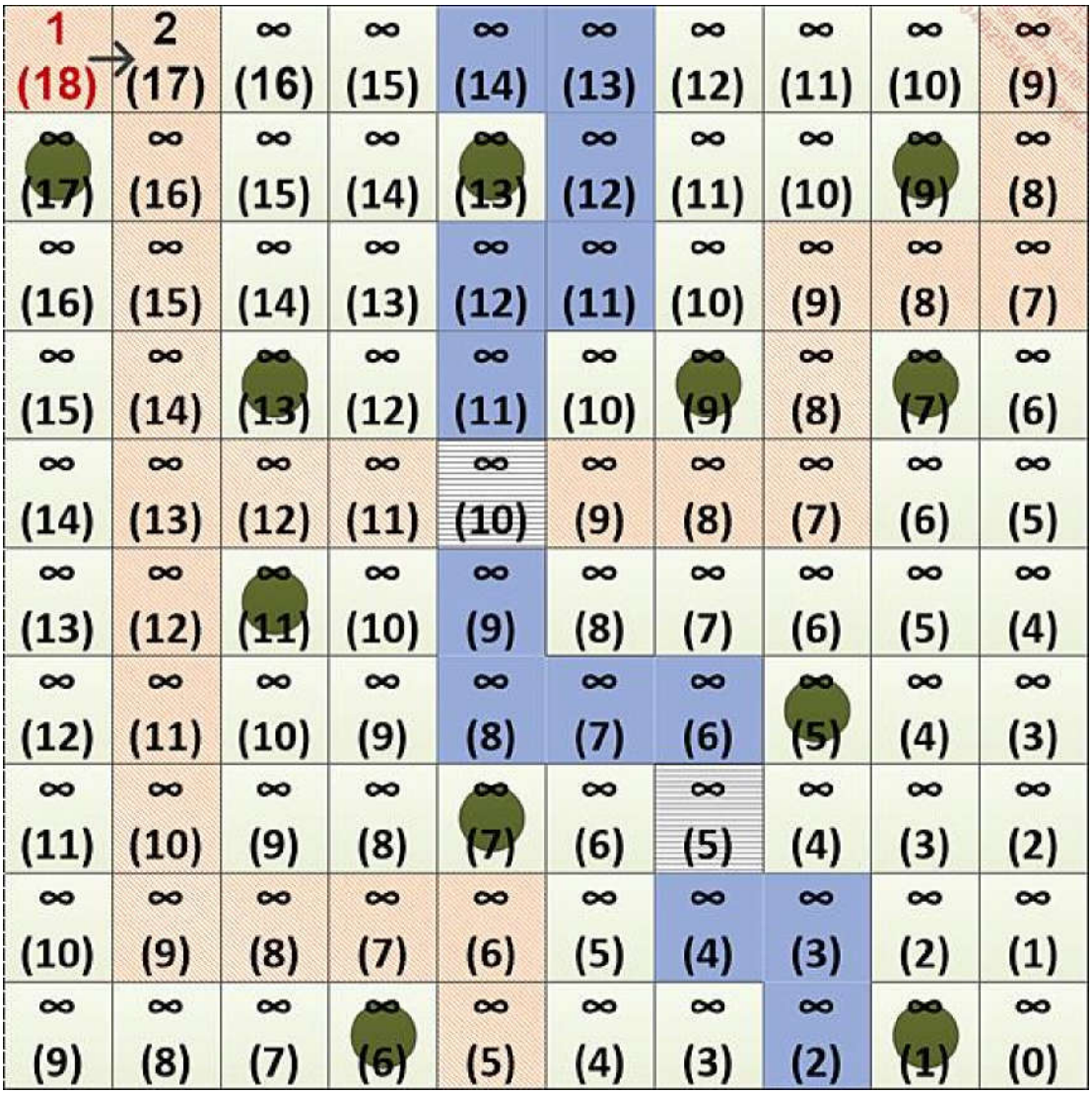
He aquí las distancias estimadas para cada casilla:



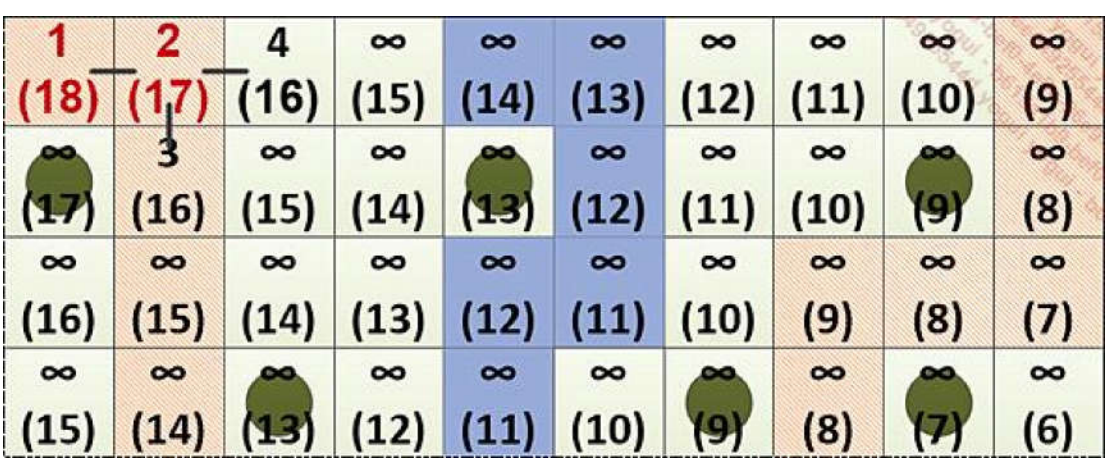
Tras la inicialización, no se ha visitado ningún nodo, y el único que tiene una distancia diferente de +∞ es la casilla de destino (que vale 1).

A partir del inicio, se calcula la distancia real hasta las dos casillas alrededor (debajo y a la derecha). La de debajo contiene un árbol y no es alcanzable, su distancia se mantiene en +∞. La que se encuentra a la derecha es un camino, su coste es 1. La distancia desde el inicio es, en este caso, 2.

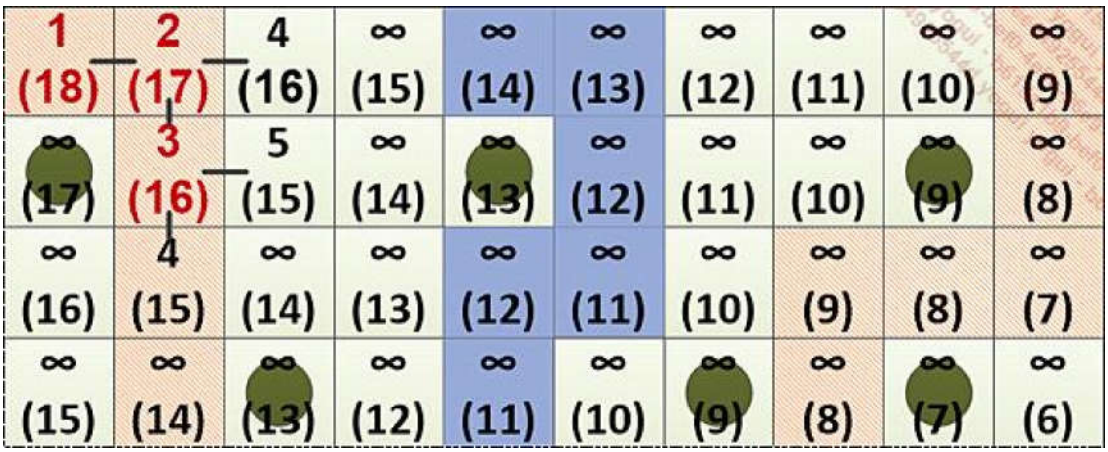
Obtenemos las siguientes distancias (las flechas permiten indicar los nodos predecesores, y las distancias estimadas restantes se indican entre paréntesis):



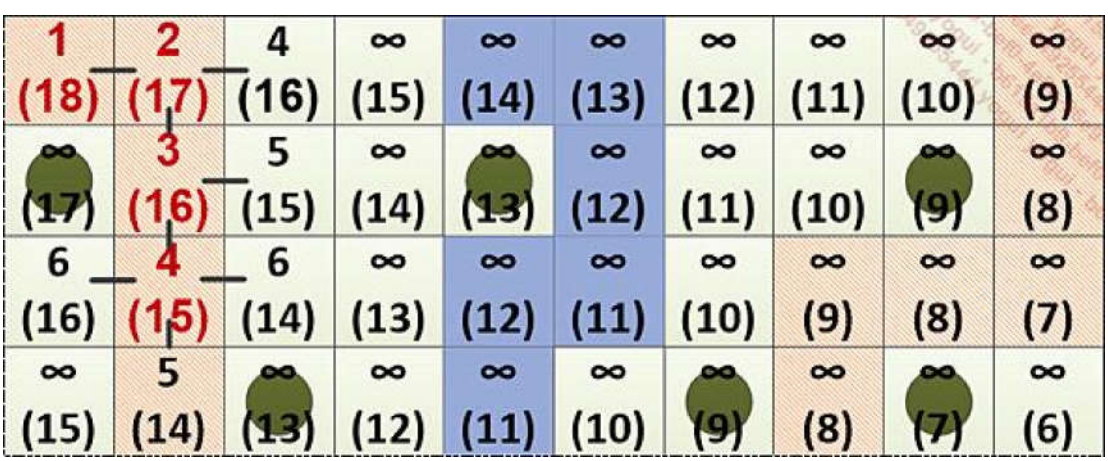
El nodo que tiene la distancia total menor y que todavía no se ha visitado es el segundo nodo (distancia total de 19). Posee dos nodos vecinos, de modo que se calculan las distancias para llegar, a saber, 2 para la casilla con hierba y 1 para la casilla con camino. Obtenemos las nuevas distancias siguientes (solo se muestra la parte superior del mapa):



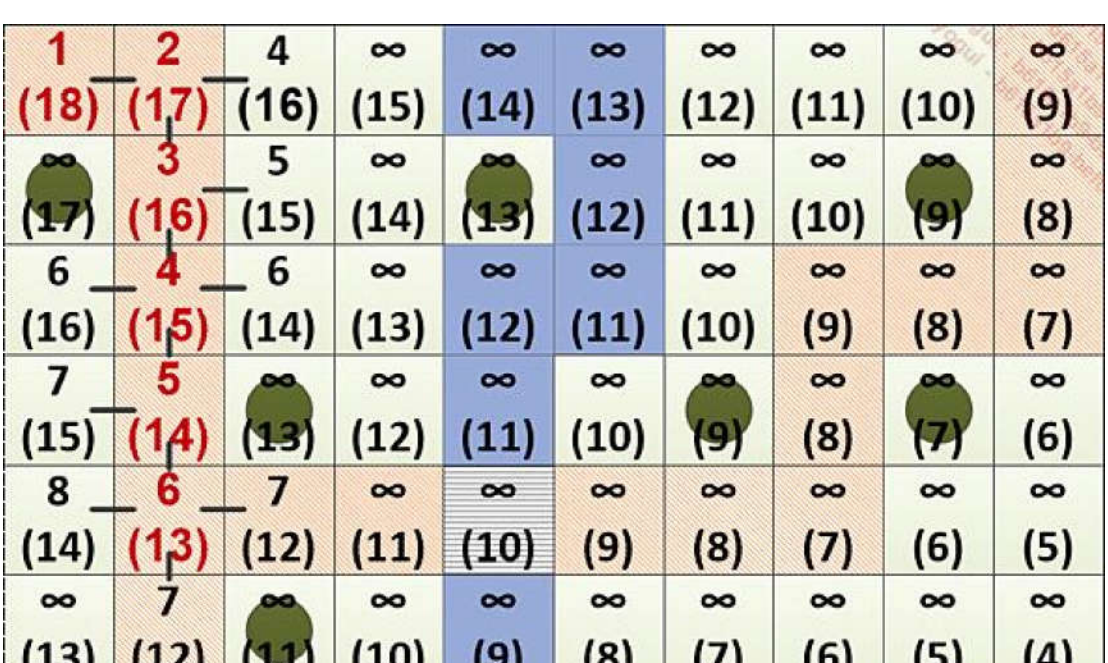
Si agregamos las distancias desde el origen a las distancias estimadas para llegar al destino, vemos que el nodo que está sobre el camino parece el más prometedor. En efecto, obtenemos una distancia total de 3+16 = 19 frente a 4+16 = 20. Lo utilizaremos en la próxima iteración.



De nuevo, es el nodo sobre el camino el que parece más prometedor, con una distancia total de 4+15 = 19. Utilizaremos este nodo y aplicaremos los arcos de salida.

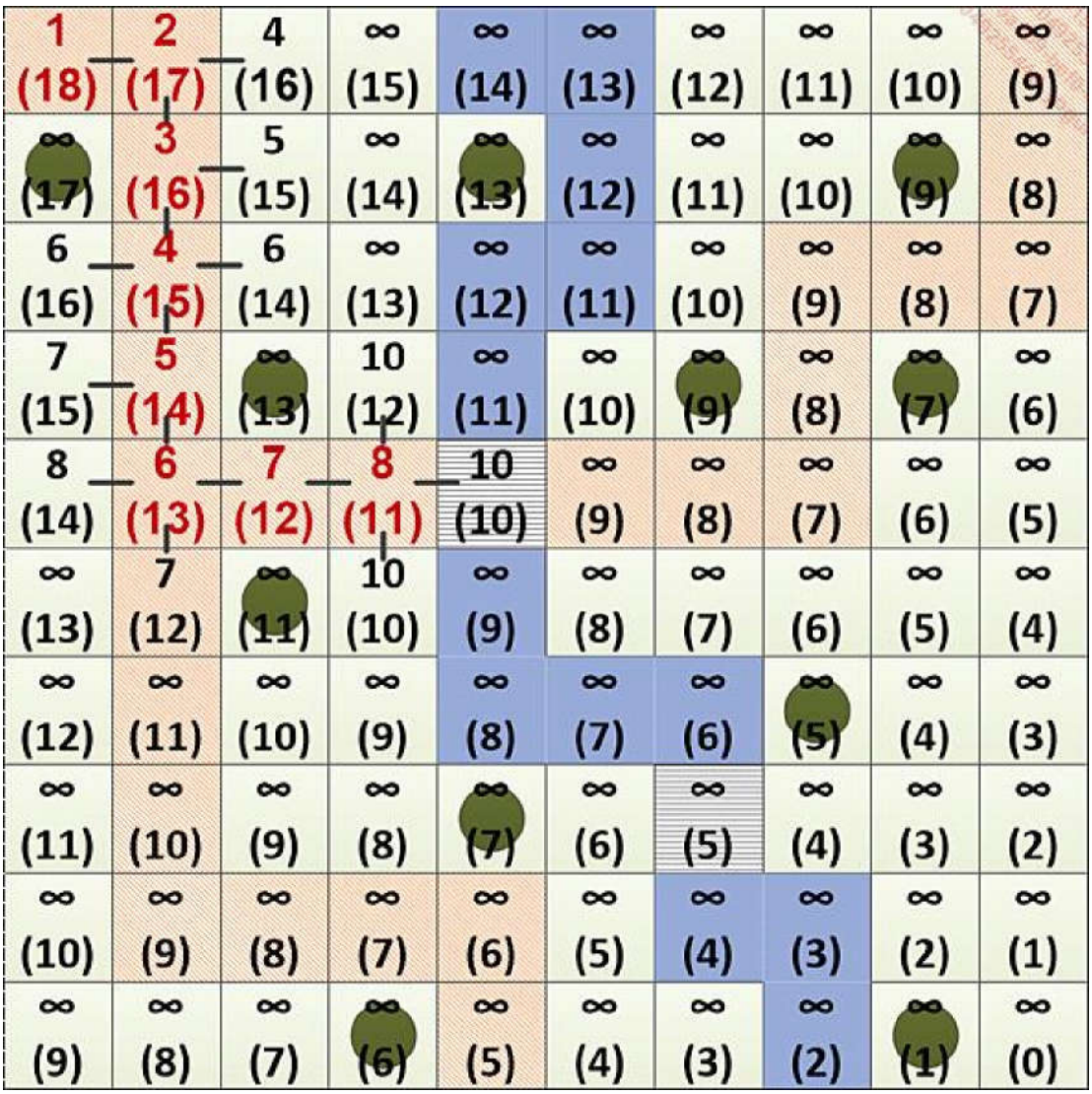


Continuamos con dos nuevas iteraciones:

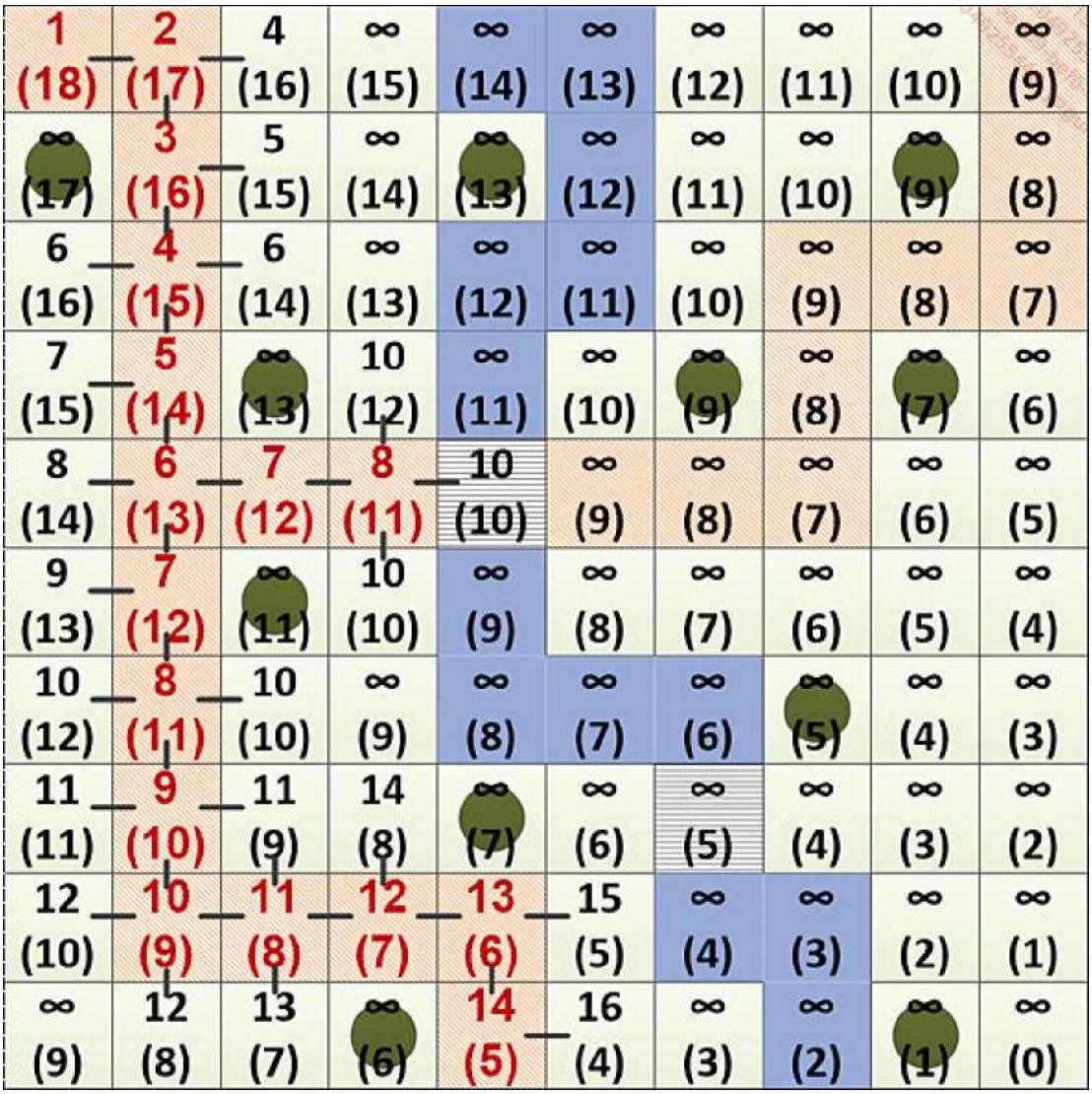


Esta vez, tenemos dos nodos iguales: los dos nodos a 7 sobre los caminos tienen una distancia estimada hasta el destino igual a 12, es decir, una distancia total de 19. Como no podemos diferenciarlos, el algoritmo va a utilizar el primer nodo en el orden de lectura, en este caso el situado a la derecha.

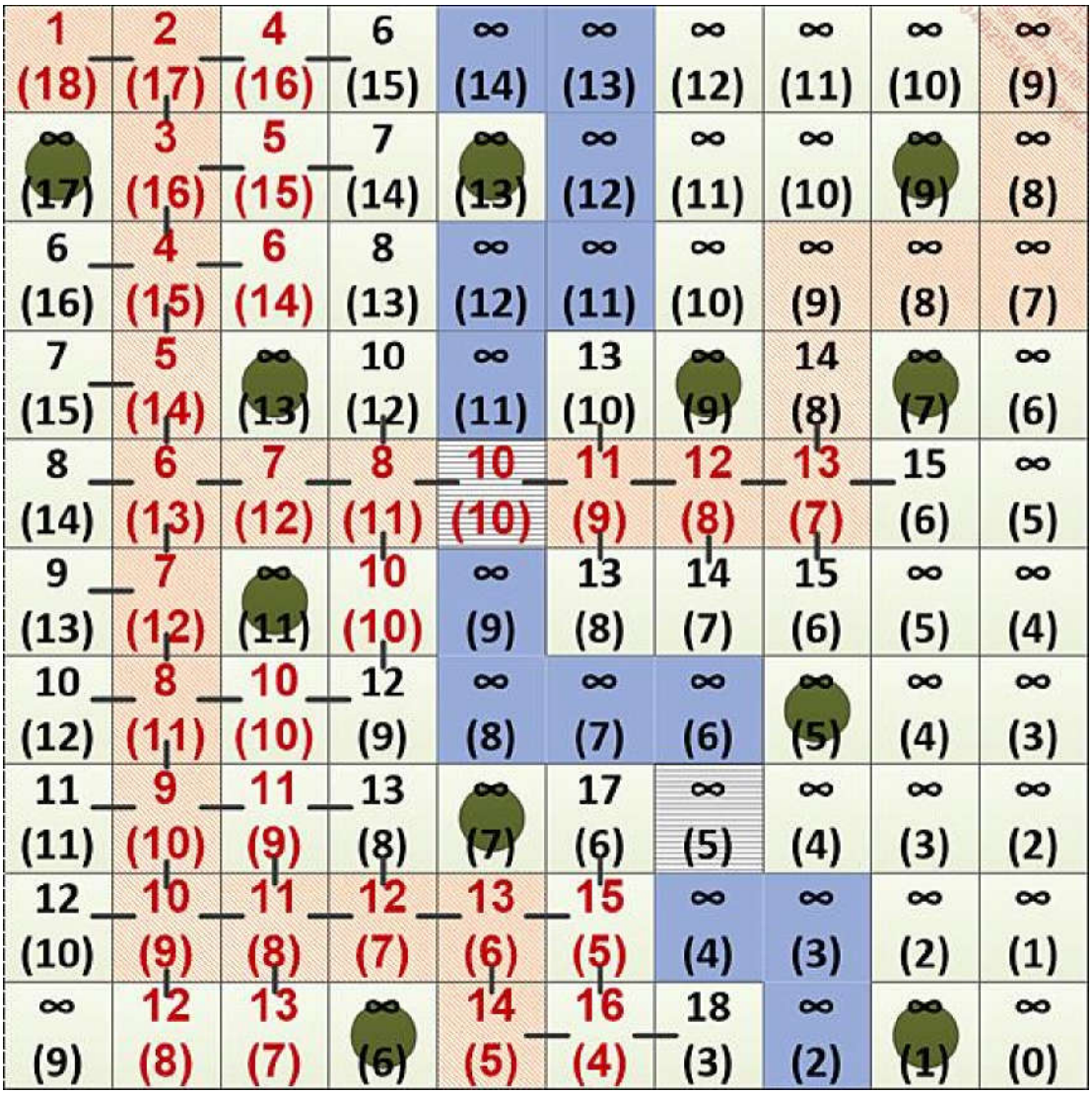
Aplicamos de nuevo el algoritmo durante dos nuevas iteraciones:



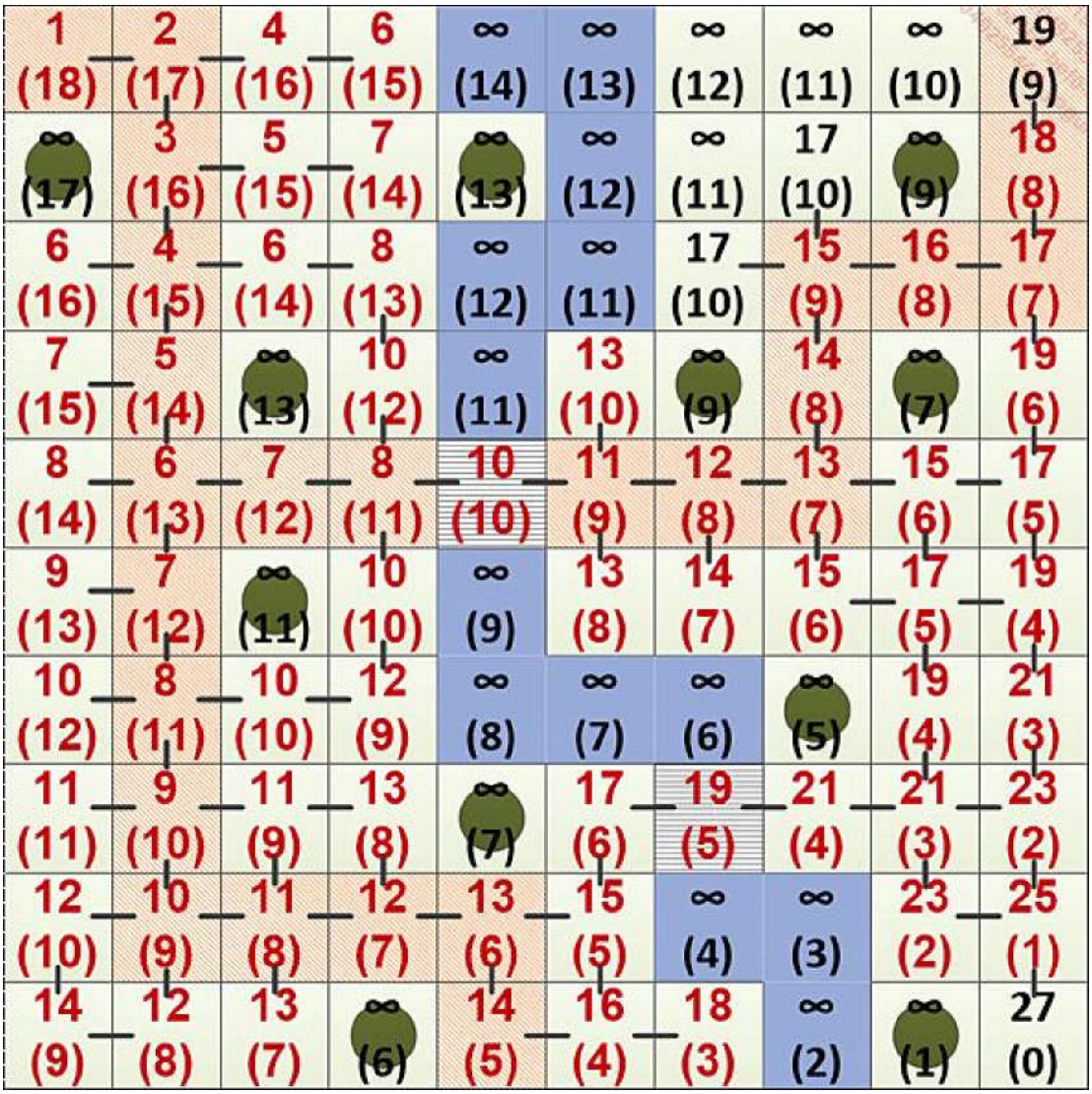
Aquí, encontramos un puente, que tiene un coste igual a 2. Con una distancia estimada de 10 hasta el destino, la distancia total es igual a 20, que es superior al camino que hemos dejado de lado más abajo. Volvemos a partir desde esta ruta, que tiene una distancia total de 7+12 = 19 puntos de acción. Continuamos mientras existan casillas con un coste total de 19, hasta obtener el siguiente resultado:



Como no existen más rutas de 19, volvemos a partir con los caminos cuya suma es igual a 20, es decir, el nodo situado más arriba (4+16). Con cada iteración, vamos a avanzar a partir de la primera casilla que tenga una distancia de 20.



No quedan más rutas de tamaño 20, de modo que pasamos a aquellos caminos con un coste total igual a 21, a continuación 22, 23... Cuando llegamos al objetivo, obtenemos el siguiente resultado:



Hemos alcanzado el destino: el algoritmo se detiene. Encontramos una distancia de 27, que es en efecto el óptimo (y por lo tanto el mismo recorrido que para los dos algoritmos anteriores), salvo que en esta ocasión no se han evaluado todas las casillas (las situadas en la zona superior derecha tienen, todavía, una distancia infinita hasta el destino, puesto que no se han calculado).

Cuanto más espacio exista alrededor de un punto de partida, más se notará esta optimización. Por el contrario, en el caso de rutas muy complejas con grandes restricciones, como ocurre, por ejemplo, en un laberinto, el algoritmo no tiene tan buen rendimiento.

Dominios de aplicación

Estos algoritmos de búsqueda de rutas se utilizan en muchos dominios.

El primer dominio es el de la **búsqueda de itinerarios**. Todos los GPS y las aplicaciones que permiten ir desde un lugar a otro (en tren, en bus, en metro, a pie...) utilizan algoritmos de pathfinding. Tienen en cuenta la longitud del camino o su tiempo. Vista la complejidad de los mapas que a menudo se utilizan (por ejemplo, en Google Maps), resulta evidente que los algoritmos deben optimizarse, dando preferencia a los grandes ejes siempre que sea posible. El detalle de los algoritmos obviamente no se ha expuesto.

Esta búsqueda de rutas se encuentra en los **videojuegos**. El objetivo es desplazar a un personaje (controlado por el jugador o representando a un enemigo) desde un lugar a otro. Los mapas pueden ser muy grandes, y el número de personajes, importante. En este caso, conviene optimizar los algoritmos utilizados. Podemos destacar, sin embargo, que el algoritmo A* es el más implementado.

La **robótica** es otro dominio que utiliza búsqueda de itinerarios. Se trata de llevar a un robot de un punto a otro lo más rápido posible. Estos algoritmos se modifican, generalmente por dos motivos. El primero es que el entorno fluctúa: si el robot avanza entre humanos, estos se desplazarán y le bloquearán el camino o, por el contrario, le despejarán rutas. Es preciso, por tanto, recalcular permanentemente el mejor camino. El segundo motivo es que el robot no posee necesariamente un conocimiento completo del mapa. En efecto, no conoce más que las zonas que ya ha visitado, e incluso estas zonas pueden cambiar (por ejemplo, si se cierra una puerta). El mapa no es estático en el tiempo.

Se utilizan muchos algoritmos de pathfinding en las redes para el **enrutado**. Internet es un muy buen ejemplo, con muchos algoritmos que permiten decidir la mejor manera para enlazar un cliente y un servidor o enviar consultas. El protocolo RIP (del inglés *Routing Information Protocol*) utiliza, también, Bellman-Ford, enviando cada 30 segundos las nuevas rutas (las máquinas pueden conectarse o desconectarse en cualquier momento). La distancia utilizada es simplemente el número de saltos (que se corresponde con el número de máquinas en la ruta). El protocolo OSPF (*Open Shortest Path First*), creado para reemplazar a RIP, funciona con el algoritmo de Dijkstra.

Por último, la búsqueda de rutas puede utilizarse en otros dominios. En **teoría de juegos** podemos aplicarla para buscar un camino que vaya desde la posición inicial hasta una posición ganadora, o al menos una posición interesante. En efecto, a menudo hay demasiadas posibilidades para realizar una búsqueda exhaustiva.

Así funcionan también muchos adversarios electrónicos en el juego de ajedrez: se realiza una búsqueda en anchura sobre varios niveles que permite determinar cuál es el movimiento que parece más ventajoso. Se considera que un humano puede comprobar tres niveles de profundidad para decidir su movimiento, mientras que Deep Blue, el superordenador que venció a Kasparov, es capaz de comprobar unos 8.

Además, DeepMind batió en diciembre del 2016 a los dos mejores jugadores del mundo de Go, gracias a su algoritmo AlphaGo, basado en una búsqueda de rutas (búsqueda de Monte Carlo). Por lo tanto, esta filial de Google ha conseguido sobrepasar al ser humano en el juego considerado, en la actualidad, como el más complicado.

Al final, todo problema que pueda expresarse bajo la forma de un grafo (como la planificación de un proyecto, o un procedimiento) puede utilizar un algoritmo de pathfinding para encontrar el camino más corto, el más rápido o incluso el más económico. Las posibilidades de aplicación son, por lo tanto, muy numerosas.

Implementación

Vamos a pasar a la implementación de estos algoritmos. El código será, sin embargo, muy genérico, lo que permitirá agregar fácilmente nuevos métodos de resolución o nuevos problemas para resolver.

A continuación, lo aplicaremos al problema del mapa, mediante una aplicación de consola.

1. Nodos, arcos y grafos

La primera etapa consiste en definir nuestros grafos. Vamos a empezar por los nodos, a continuación veremos los arcos que los enlazan y por último el grafo completo.

a. Implementación de los nodos

Los nodos son las estructuras básicas de nuestros grafos. Sin embargo, el contenido real de un nodo depende en gran medida del problema que se desea resolver: puede tratarse de estaciones, de casillas en una cuadrícula, de servidores, de ciudades...

Creamos, por tanto, una clase abstracta **`Nodo`**, que contendrá la información necesaria para los algoritmos. Esta clase debe ser heredada para la resolución práctica de un problema.

Los nodos necesitan tres datos:

- El precursor, que es también un nodo.
- La distancia desde el inicio.
- La distancia estimada hasta el destino (si es necesario).

Utilizaremos atributos públicos para simplificar los accesos (esto evitará tener accesores/mutadores). Para los dos primeros, tendremos valores por defecto.

```
public abstract class Nodo {
    public Nodo precursor = null;
    public double distanciaDelInicio = Double.POSITIVE_INFINITY;
    public double distanciaEstimada;
}
```

b. Clase que representa los arcos

Una vez definidos los nodos, podemos definir los arcos gracias a la clase **`Arc`**. Esta clase contiene tres propiedades:

- El nodo de partida del arco.
- El nodo de llegada.
- La longitud o coste del arco.

Se agrega un constructor para inicializar más rápidamente estas tres propiedades:

```
public class Arco {
    protected Nodo origen;
    protected Nodo destino;
    protected double coste;

    public Arco(Nodo _origen, Nodo _destino, double _coste) {
        origen = _origen;
        destino = _destino;
        coste = _coste;
    }
}
```

c. Grafos

Pasemos ahora a los grafos. Utilizaremos una interfaz, **`Grafo`**, que contendrá todos los métodos que tendrá que definir cada grafo. En efecto, la mayoría de estos métodos dependerán en gran medida del problema.

```
import java.util.ArrayList;

// Interfaz que define los grafos
public interface Grafo {
    // Aquí el código
}
```

Necesitamos, en primer lugar, dos métodos que nos permitan obtener el nodo de partida o de llegada:

```
    Nodo NodoPartida();
    Nodo NodoSalida();
```

Agregamos también dos métodos que permitan recuperar todos los nodos bajo la forma de una lista y otro dos para recuperar todos los arcos. En ambos casos, el primero no recibirá ningún parámetro, y devolverá la lista completa, mientras que el segundo recibirá un nodo y devolverá los nodos adyacentes o los arcos que salen.

```
    ArrayList<Nodo> ListaNodos();
    ArrayList<Nodo> ListaNodosAdyacentes(Nodo origen);
    ArrayList<Arco> ListaArcos();
    ArrayList<Arco> ListaArcosSalientes(Nodo origen);
```

Se agregan algunas funciones complementarias para:

- Contar el número de nodos.
- Devolver la distancia entre dos nodos.
- Calcular la distancia estimada hasta el destino.
- Reconstruir el camino a partir de los nodos predecesores.
- Restablecer el grafo a su estado inicial.

```
    int NumeroNodos();
    double Coste(Nodo salida, Nodo llegada);
    String RecostruirRuta();
    void CalcularDistanciasEstimadas();
    void Borrar();
```

Una vez codificados nuestros grafos, podemos pasar a las últimas clases e interfaces genéricas.

2. Fin del programa genérico

Para terminar el programa genérico, nos faltan dos elementos: una interfaz IHM y una clase abstracta de la que heredarán los distintos algoritmos.

a. IHM

El programa se utiliza, en este caso, desde una consola, pero podría utilizarse desde una interfaz gráfica o incluso publicarse a través de la red como un servicio web. Debemos separar el programa genérico de su salida.

Crearemos, por tanto, una interfaz **`IHM`** para definirla. Esta mostrará el resultado como un camino en forma de cadena e indicará la distancia obtenida.

```
public interface IHM
{
   void MostrarResultado(string ruta, double distancia);
}
```

b. Algoritmo genérico

La última clase es la correspondiente al algoritmo genérico. Se trata de la clase abstracta **`Algoritmo`**, de la que heredará cada uno de los cinco algoritmos que vamos a implementar.

Esta clase contiene, en primer lugar, dos atributos para el grafo que se ha de tratar y la IHM para la salida, así como un constructor que las inicializará:

```
public abstract class Algoritmo {
    protected Grafo grafo;
    protected IHM ihm;

    public Algoritmo(Grafo _grafo, IHM _ihm) {
        grafo = _grafo;
        ihm = _ihm;
    }
}
```

A continuación, nos hace falta el método principal, `Resolver()`.

Este sigue el patrón de diseño "patrón de método", es decir, que nos permite fijar el comportamiento general del método. Los descendentes no tendrán que redefinir más que ciertos métodos. Se declara por tanto `final` para

estar seguros de que no se redefinirá en las clases descendientes.

Este método tiene tres etapas:

- Reiniciar el problema y, por lo tanto, el grafo (si es necesario).
- Ejecutar el algoritmo.
- Mostrar el resultado mediante la IHM.

El método `Run` es abstracto y es el único que debemos redefinir.

```
    public final void Resolver() {
        grafo.Borrar();
        Run();
        ihm.MostrarResultado(grafo.RecostruirRuta(),
grafo.NodoSalida().distanciaDelInicio);
    }

    protected abstract void Run();
```

3. Implementación de los diferentes algoritmos

Hemos terminado el programa genérico. Tan solo queda codificar los distintos algoritmos, que heredan todos de la clase `Algoritmo`. Es posible, también, agregar fácilmente nuevos algoritmos.

Además, por motivos de legibilidad del código, las implementaciones serán muy próximas a los pseudocódigos. Es posible optimizarlos si es necesario.

a. Búsqueda en profundidad

Empezamos con la búsqueda de rutas en profundidad. Como ocurre con todos los algoritmos, esta clase **`BusquedaEnProfundidad`** hereda de la clase abstracta `Algoritmo`. El constructor invoca al de su clase madre.

Dentro del método `Run`, que es el núcleo de nuestra clase, se reproduce el pseudocódigo que hemos visto antes. Consiste en conservar la lista de todos los nodos que todavía no se han visitado, y una pila de nodos. Se parte del nodo inicial y se agrega a la pila todos los nodos vecinos actualizándolos (precursor y distancia desde el origen).

Si el nodo que desapilamos es el nodo de destino, entonces hemos terminado.

Utilizaremos la clase `Stack` para gestionar la pila, que tiene dos métodos importantes para nosotros: `push`, que agrega un elemento a la pila, y `pop`, que recupera el primer elemento situado en su parte superior.

Obtenemos el siguiente código:

```
import java.util.ArrayList;
import java.util.Stack;

// Algoritmo de búsqueda en profundidad
public class BusquedaEnProfundidad extends Algoritmo {
```

```
    // Constructor
    public BusquedaEnProfundidad(Grafo _grafo, IHM _ihm) {
        super(_grafo,_ihm);
    }

    // Método de resolución
    @Override
    protected void Run() {
        // Creación de la lista de nodos no visitados y de la pila
        ArrayList<Nodo> nodosNoVisitados = grafo.ListaNodos();
        Stack<Nodo> nodosAVisitar = new Stack();
        nodosAVisitar.push(grafo.NodoPartida());
        nodosNoVisitados.remove(grafo.NodoPartida());

        // Inicialización de la salida
        Nodo nodoSalida = grafo.NodoSalida();
        boolean salidaEncontrada = false;

        // Bucle principal
        while(!salidaEncontrada && nodosAVisitar.size() != 0) {
            Nodo nodoEnCurso = nodosAVisitar.pop();
            if (nodoEnCurso.equals(nodoSalida)) {
                salidaEncontrada = true;
            }
            else {
                for (Nodo n :
grafo.ListaNodosAdyacentes(nodoEnCurso)) {
                    if (nodosNoVisitados.contains(n)) {
                        nodosNoVisitados.remove(n);
                        n.precursor = nodoEnCurso;
                        n.distanciaDelInicio =
nodoEnCurso.distanciaDelInicio + grafo.Coste(nodoEnCurso, n);
                        nodosAVisitar.push(n);
                    }
                }
            }
        }
    }
}
```

b. Búsqueda en anchura

La búsqueda de un camino mediante una búsqueda en anchura se parece bastante a la búsqueda en profundidad. Por lo tanto, la clase **`BusquedaEnAnchura`** se parece bastante a la clase anterior.

El código es el mismo, salvo por las líneas en negrita, que permiten reemplazar la pila (`Stack`) por una fila (representada por una lista encadenada `LinkedList`) y los métodos `push/pop` por `add` (para agregar al final) y `removeFirst` (para eliminar al principio).

```
import java.util.ArrayList;
import java.util.LinkedList;
```

```
// Algoritmo de búsqueda en anchura
public class BusquedaEnAnchura extends Algoritmo {

    // Constructor
    public BusquedaEnAnchura(Grafo _grafo, IHM _ihm) {
        super(_grafo,_ihm);
    }

    // Método de resolución
    @Override
    protected void Run() {
        // Creación de la lista de nodos no visitados y de la pila
        ArrayList<Nodo> nodosNoVisitados = grafo.ListaNodos();
        LinkedList<Nodo> nodosAVisitar = new LinkedList();
        nodosAVisitar.add(grafo.NodoPartida());
        nodosNoVisitados.remove(grafo.NodoPartida());

        // Inicialización de la salida
        Nodo nodoSalida = grafo.NodoSalida();
        boolean salidaEncontrada = false;

        // Bucle principal
        while(!salidaEncontrada && nodosAVisitar.size() != 0) {
            Nodo nodoEnCurso = nodosAVisitar.removeFirst();
            if (nodoEnCurso.equals(nodoSalida)) {
                // Se ha terminado el algoritmo
                salidaEncontrada = true;
            }
            else {
                // Se agregan los vecinos todavía no visitados
                for (Nodo n : grafo.ListaNodosAdyacentes(nodoEnCurso)) {
                    if (nodosNoVisitados.contains(n)) {
                        nodosNoVisitados.remove(n);
                        n.precursor = nodoEnCurso;
                        n.distanciaDelInicio =
nodoEnCurso.distanciaDelInicio + grafo.Coste(nodoEnCurso, n);
                       nodosAVisitar.add(n);
                    }
                }
            }
       }
    }
}
```

c. Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford (clase **`BellmanFord`**) es el primero que garantiza el camino más corto. Consiste en aplicar todos los arcos, y en actualizar los nodos si se encuentra algún camino más corto, tantas veces como número de nodos (menos uno). Se detiene, no obstante, si no es posible mejorar las distancias encontradas.

Se trata, por tanto, de un gran bucle "mientras". Con cada arco, se consulta si el nodo se ha alcanzado de una manera más corta que hasta el momento. En caso afirmativo, se actualiza el predecesor y la distancia desde el inicio.

Se agrega una última iteración al final que permite comprobar si existe un camino óptimo; en caso contrario se mostrará un error.

```
import java.util.ArrayList;

// Algoritmo de Bellman-Ford
public class BellmanFord extends Algoritmo {
    // Constructor
    public BellmanFord(Grafo _grafo, IHM _ihm) {
        super(_grafo, _ihm);
    }

    // Método de resolución
    @Override
    protected void Run() {
        // Inicialización
        boolean distanciaCambiada = true;
        int i = 0;
        ArrayList<Arco> listaArcos = grafo.ListaArcos();

        // Bucle principal
        int numBucleMax = grafo.NumeroNodos() - 1;
        while (i < numBucleMax && distanciaCambiada) {
            distanciaCambiada = false;
            for (Arco arco : listaArcos) {
                  if (arco.origen.distanciaDelInicio + arco.coste
< arco.destino.distanciaDelInicio) {
                    // Se ha encontrado una ruta más corta
                    arco.destino.distanciaDelInicio =
arco.origen.distanciaDelInicio +  arco.coste;
                    arco.destino.precursor = arco.origen;
                    distanciaCambiada = true;
                }
            }
            i++;
        }

        // Comprueba si es negativa
        for (Arco arco : listaArcos) {
            if (arco.origen.distanciaDelInicio + arco.coste <
arco.destino.distanciaDelInicio) {
                System.err.println("Bucle negativo - no hay
ninguna ruta más corta");
            }
        }
    }
}
```

d. Algoritmo de Dijkstra

El algoritmo de Dijkstra (clase **`Dijkstra`**) se aplica no sobre los arcos, sino sobre los nodos. En cada iteración se selecciona un nodo y se aplican todos los arcos salientes, actualizando los nodos adyacentes para los que hemos

encontrado un camino más corto que el previamente anotado.

Para seleccionar el nodo utilizado, tomamos aquel cuya distancia al origen sea la menor, una vez recorrida nuestra lista.

```
import java.util.ArrayList;

// Algoritmo de Dijkstra
public class Dijkstra extends Algoritmo {

    // Constructor
    public Dijkstra(Grafo _grafo, IHM _ihm) {
        super(_grafo, _ihm);
    }

    // Método principal
    @Override
    protected void Run() {
        // Inicialización
        ArrayList<Nodo> listaNodos = grafo.ListaNodos();
        boolean salidaEncontrada = false;

        // Bucle principal
        while(listaNodos.size() != 0 && !salidaEncontrada) {
            // Búsqueda del nodo con la distancia menor
            Nodo nodoEnCurso = listaNodos.get(0);
            for (Nodo nodo : listaNodos) {
                 if (nodo.distanciaDelInicio <
nodoEnCurso.distanciaDelInicio) {
                     nodoEnCurso = nodo;
                }
            }

            if (nodoEnCurso.equals(grafo.NodoSalida())) {
                salidaEncontrada = true;
            }
            else {
                ArrayList<Arco> arcosSalientes =
grafo.ListaArcosSalientes(nodoEnCurso);

                for (Arco arco : arcosSalientes) {
                  if (arco.origen.distanciaDelInicio + arco.coste <
arco.destino.distanciaDelInicio) {
                      arco.destino.distanciaDelInicio =
arco.origen.distanciaDelInicio + arco.coste;
                      arco.destino.precursor = arco.origen;
                   }
                }

                listaNodos.remove(nodoEnCurso);
            }
        }
    }
}
```

e. Algoritmo A*

El algoritmo A* (clase **AStar**) es prácticamente idéntico al Dijkstra. La diferencia es que se ordenan los nodos no solo en función de las distancias al origen, sino también basándose en la distancia al origen sumada a la distancia estimada hasta el destino.

En primer lugar, en la inicialización se invoca al método que permite estimar esta distancia. Se cambia, a continuación, la condición de nuestro recorrido para encontrar el elemento que tenga una distancia total menor. El resto del código es idéntico. A continuación se muestra la clase completa y se indican en negrita las líneas modificadas respecto a Dijkstra.

```
import java.util.ArrayList;
// Algoritmo A*
public class AStar extends Algoritmo {

    // Constructor
    public AStar(Grafo _grafo, IHM _ihm) {
        super(_grafo, _ihm);
    }

    // Método principal
    @Override
    protected void Run() {
        // Inicialización
        grafo.CalcularDistanciasEstimadas();
        ArrayList<Nodo> listaNodos = grafo.ListaNodos();
        boolean salidaEncontrada = false;

        // Bucle principal
        while(listaNodos.size() != 0 && !salidaEncontrada) {
            // Búsqueda del nodo con la distancia menor
            Nodo nodoEnCurso = listaNodos.get(0);
            for (Nodo nodo : listaNodos) {
                 if (nodo.distanciaDelInicio +
nodo.distanciaEstimada < nodoEnCurso.distanciaDelInicio +
nodoEnCurso.distanciaEstimada) {
                     nodoEnCurso = nodo;
                }
            }

            if (nodoEnCurso.equals(grafo.NodoSalida())) {
                // Se ha encontrado la salida
                salidaEncontrada = true;
            }
            else {
                // Se aplican los arcos salientes de este nodo
                ArrayList<Arco> arcosSalientes =
grafo.ListaArcosSalientes(nodoEnCurso);

                for (Arco arco : arcosSalientes) {
                   if (arco.origen.distanciaDelInicio + arco.coste <
arco.destino.distanciaDelInicio) {
```

```
                        arco.destino.distanciaDelInicio =
arco.origen.distanciaDelInicio + arco.coste;
                        arco.destino.precursor = arco.origen;
                    }
                }

                listaNodos.remove(nodoEnCurso);
            }
        }
    }
}
```

4. Aplicación al mapa

Vamos a aplicar, a continuación, nuestros algoritmos para resolver el problema de búsqueda de caminos en nuestro mapa. Debemos, por tanto, definir las clases particulares que representan los nodos de nuestro grafo y el propio grafo.

a. Gestión de las baldosas

Cada casilla de nuestro mapa es una baldosa. Pueden ser de distintos tipos: hierba, camino, árbol, agua o puente.

Definimos, en primer lugar, los distintos tipos **`TipoBaldosa`** mediante una enumeración:

```
// Enumeración de los tipos de terreno
public enum TipoBaldosa {Hierba, Arbol, Agua, Puente, Camino};
```

Agregamos una pequeña clase complementaria **`ConvertidorTipoBaldosa`** que permite transformar un carácter en un tipo. Esto permitirá introducir un nuevo mapa de manera sencilla.

Seleccionamos las siguientes convenciones para cada tipo: los caminos se representan por "." (para representar las piedras), el árbol por "*" (el árbol visto desde arriba), la hierba por " " (espacio, es decir, vacío), el agua con una "X" (infranqueable) y los puentes con "=" (representando el puente).

```
class ConvertidorTipoBaldosa {
    public static TipoBaldosa CharToType(char c) {
        switch (c) {
            case ' ' :
                return TipoBaldosa.Hierba;
            case '*' :
                return TipoBaldosa.Arbol;
            case '=' :
                return TipoBaldosa.Puente;
            case 'X' :
                return TipoBaldosa.Agua;
            case '.' :
                return TipoBaldosa.Camino;
        }
        return null;
    }
```

```
}
```

A continuación debemos codificar las clases. Tenemos también una clase **Baldosa** que hereda de la clase **Nodo.**

Nuestras baldosas poseen, además de los atributos propios de los nodos, tres nuevos atributos:

- El tipo de firme.
- La fila en el mapa.
- La columna.

Esta clase comienza con un constructor.

```
public class Baldosa extends Nodo {
    protected TipoBaldosa tipo;
    protected int fila;
    protected int columna;

    // Constructor
    public Baldosa(TipoBaldosa _tipo, int _fila, int _columna) {
        tipo = _tipo;
        fila = _fila;
        columna = _columna;
    }
}
```

Además, crearemos un método `Accesible` que indicará si es posible ir hasta una casilla o no. Para ello, basta con observar el tipo de casilla: solamente los caminos, la hierba y los puentes son accesibles.

```
    boolean Accesible() {
        return (tipo.equals(TipoBaldosa.Camino) ||
tipo.equals(TipoBaldosa.Hierba) ||
tipo.equals(TipoBaldosa.Puente));
    }
```

Agregaremos un método que nos indique el coste de la casilla. Los caminos tienen un coste en puntos de acción igual a 1, la hierba y los puentes igual a 2. Devolvemos una distancia infinita para las casillas que no son alcanzables (árboles y agua).

```
    double Coste() {
        switch (tipo) {
            case Camino :
                return 1;
            case Puente :
            case Hierba :
                return 2;
        default :
                return Double.POSITIVE_INFINITY;
        }
```

```
    }
```

Terminamos esta clase sobrecargando el método `toString()`, que muestra las coordenadas de la casilla, así como su tipo:

```
    @Override
    public String toString() {
        return "[" + fila + ";" + columna + ";" +
tipo.toString() + "]";
    }
```

b. Implementación del mapa

Una vez definidas las casillas, podemos pasar al mapa, representado por la clase **Mapa** que implementa la interfaz `Grafo`. Se trata de la clase más larga en términos de líneas de código.

Definimos en primer lugar los nuevos atributos: el mapa se representa mediante un array de baldosas de dos dimensiones (`baldosas`). Guardamos también el número de filas y de columnas y las baldosas inicial y final.

Por motivos de optimización, conservamos dos listas, vacías al principio: la lista de nodos y la lista de arcos.

```
import java.util.ArrayList;
import java.util.Arrays;
public class Mapa implements Grafo {
    Baldosa[][] baldosas;
    int numFilas;
    int numColumnas;
    Baldosa nodoPartida;
    Baldosa nodoLlegada;
    ArrayList<Nodo> listaNodos = null;
    ArrayList<Arco> listaArcos = null;
}
```

El constructor recibe como parámetro una cadena de caracteres que contiene el diseño del mapa del tipo "ASCII Art". Vamos a inicializar el tablero de baldosas, separar cada fila y cada carácter y completarlo. Se registra, a continuación, el inicio y el destino y terminamos inicializando las listas de nodos y arcos.

```
    public Mapa(String _mapa, int _filaSalida, int
_columnaSalida, int _filaLlegada, int _columnaLlegada) {
        // Creación del array de baldosas
        String[] filas = _mapa.split("\n");
        numFilas = filas.length;
        numColumnas = filas[0].length();
        baldosas = new Baldosa[numFilas][];

        // Relleno
        for (int i = 0; i < numFilas; i++) {
            baldosas[i] = new Baldosa[numColumnas];
            for (int j = 0; j < numColumnas; j++) {
```

```
                TipoBaldosa tipo =
ConvertidorTipoBaldosa.CharToType(filas[i].charAt(j));
                baldosas[i][j] = new Baldosa(tipo, i, j);
            }
        }

        // Partida y llegada
        nodoPartida = baldosas[_filaSalida][_columnaSalida];
        nodoPartida.distanciaDelInicio = nodoPartida.Coste();
        nodoLlegada = baldosas[_filaLlegada][_columnaLlegada];

        // Lista de nodos y de arcos
        ListaNodos();
        ListaArcos();
    }
```

A continuación tenemos que implementar todos los métodos de la interfaz. Empezamos con los métodos que devuelven las baldosas de partida y de destino, que no hacen más que devolver el atributo correspondiente.

```
    @Override
    public Nodo NodoPartida() {
        return nodoPartida;
    }

    @Override
    public Nodo NodoSalida() {
        return nodoLlegada;
    }
```

El siguiente método debe devolver la lista de todos los nodos. Si no se ha creado la lista en una llamada anterior, la crearemos. Para ello, recorremos todas las filas de la tabla y agregamos todas las columnas de una vez a la lista (gracias al método `addAll` de `Collections`).

```
    @Override
    public ArrayList<Nodo> ListaNodos() {
        if (listaNodos == null) {
            listaNodos = new ArrayList();
            for (int i = 0; i < numFilas; i++) {
                listaNodos.addAll(Arrays.asList(baldosas[i]));
            }
        }
        return listaNodos;
    }
```

Hace falta también un método que devuelva los nodos adyacentes al nodo que se pasa como parámetro. En este caso, comprobaremos los cuatro vecinos y si son alcanzables los agregaremos a nuestra lista, que luego devolveremos. Debemos prestar atención a los bordes del mapa.

```
    @Override
    public ArrayList<Nodo> ListaNodosAdyacentes(Nodo origen) {
```

```
        // Inicialización
        ArrayList<Nodo> listaNodosSalientes = new ArrayList();
        int fila = ((Baldosa)origen).fila;
        int columna = ((Baldosa)origen).columna;

        // Vecino de la derecha
        if (columna - 1 >= 0 && baldosas[fila][columna-1]
.Accesible()) {
            listaNodosSalientes.add(baldosas[fila][columna-1]);
        }

        // Vecino de la izquierda
        if (columna + 1 < numColumnas && baldosas[fila][columna+1]
.Accesible()) {
            listaNodosSalientes.add(baldosas[fila][columna+1]);
        }

        // Vecino de arriba
        if (fila - 1 >= 0 && baldosas[fila-1][columna]
.Accesible()) {
            listaNodosSalientes.add(baldosas[fila-1][columna]);
        }

        // Vecino de abajo
        if (fila + 1 < numFilas && baldosas[fila+1][columna]
.Accesible()) {
            listaNodosSalientes.add(baldosas[fila+1][columna]);
        }

        return listaNodosSalientes;
    }
```

Se completa la manipulación de los nodos con un método que debe devolver el número de nodos. Se devuelve simplemente el número de casillas del tablero.

```
    @Override
    public int NumeroNodos() {
        return numFilas * numColumnas;
    }
```

Para ciertos algoritmos, como Bellman-Ford, hace falta también devolver no la lista de los nodos, sino la lista de los arcos. Vamos a empezar por la lista de los arcos salientes de un nodo en particular. Como en el caso de los nodos, vamos a probar cada dirección y si es posible, crear el arco.

Por lo tanto, el código de `ListaArcosSalientes ()`, es el siguiente:

```
    @Override
    public ArrayList<Arco> ListaArcosSalientes(Nodo origen) {
        ArrayList<Arco> listaArcosSalientes = new ArrayList();
        int fila = ((Baldosa)origen).fila;
```

```
        int columna = ((Baldosa)origen).columna;

        if (baldosas[fila][columna].Accesible()) {
            // Derecha
            if (columna - 1 >= 0 && baldosas[fila][columna-1]
.Accesible()) {
                listaArcosSalientes.add(new
Arco(baldosas[fila][columna], baldosas[fila][columna-1],
baldosas[fila][columna-1].Coste()));
            }

            // Izquierda
            if (columna + 1 < numColumnas &&
baldosas[fila][columna+1].Accesible()) {
                listaArcosSalientes.add(new
Arco(baldosas[fila][columna], baldosas[fila][columna+1],
baldosas[fila][columna+1].Coste()));
            }

            // Arriba
            if (fila - 1 >= 0 && baldosas[fila-1]
[columna].Accesible()) {
                listaArcosSalientes.add(new
Arco(baldosas[fila][columna], baldosas[fila-1][columna],
baldosas[fila-1][columna].Coste()));
            }

            // Abajo
            if (fila + 1 < numFilas &&
baldosas[fila+1][columna].Accesible()) {
                listaArcosSalientes.add(new
Arco(baldosas[fila][columna], baldosas[fila+1][columna],
baldosas[fila+1][columna].Coste()));
            }
        }
        return listaArcosSalientes;
    }
```

También debemos proporcionar la lista de todos los arcos del mapa. Para esto, nos basaremos en el método que acabamos de crear, porque para cada nodo se creará la lista de los arcos salientes que se van a añadir a la lista completa.

```
   @Override
   public ArrayList<Arco> ListaArcos() {
       if(listaArcos == null) {
           listaArcos = new ArrayList();

           // Recorrido de los nodos
           for (int linea = 0; linea < numLineas; linea++) {
               for (int columna = 0; columna < numColumnas;
columna++) {
                   ArrayList<Arco> arcos =
ListaArcosSalientes(baldosas[linea][columna]);
```

```
                    listaArcos.addAll(arcos);
                }
            }
        }
        return listaArcos;
    }
```

El siguiente método devuelve el coste para ir desde una casilla a otra. En nuestro caso, se trata simplemente del coste de la casilla de llegada.

```
    @Override
    public double Coste(Nodo salida, Nodo llegada) {
        return ((Baldosa)llegada).Coste();
    }
```

El método `ReconstruirRuta` debe crear una cadena que contenga los distintos nodos recorridos para ir desde el origen hasta el destino. Es necesario recorrer los predecesores conforme sea necesario, desde el inicio hasta el destino (que no tiene predecesor).

```
    @Override
    public String RecostruirRuta() {
        // Inicialización
        String ruta = "";
        Baldosa nodoEnCurso = nodoLlegada;
        Baldosa nodoAnterior = (Baldosa) nodoLlegada.precursor;


        // Bucle sobre los nodos de la ruta
        while (nodoAnterior != null) {
            ruta = "-" + nodoEnCurso.toString() + ruta;
            nodoEnCurso = nodoAnterior;
            nodoAnterior = (Baldosa) nodoEnCurso.precursor;
        }
        ruta = nodoEnCurso.toString() + ruta;
        return ruta;
    }
```

Para ciertos algoritmos como A*, es preciso conocer la distancia estimada hasta el destino. Se utiliza la distancia de Manhattan: se trata del número de casillas horizontales más el número de casillas verticales para ir desde la casilla en curso hasta la casilla de destino. Como se sigue subestimando la distancia, se trata de un buen heurístico para A*.

```
    @Override
    public void CalcularDistanciasEstimadas() {
        for (int fila = 0; fila < numFilas; fila++) {
            for (int columna = 0; columna < numColumnas;
columna++) {
                baldosas[fila][columna].distanciaEstimada =
Math.abs(nodoLlegada.fila - fila) +
Math.abs(nodoLlegada.columna - columna);
```

```
            }
        }
    }
```

El último método permite poner a `null` las listas de arcos y de nodos y reinicia las distancias y los precursores.

```
    @Override
    public void Borrar() {
        // Borrar las listas
        listaNodos = null;
        listaArcos = null;

        // Borrar las distancias y precursores
        for (int fila = 0; fila < numFilas; fila++) {
             for (int columna = 0; columna < numColumnas;
columna++) {
                baldosas[fila][columna].distanciaDelInicio =
Double.POSITIVE_INFINITY;
                baldosas[fila][columna].precursor = null;
           }
        }

        // Nodo inicial
        nodoPartida.distanciaDelInicio = nodoPartida.Coste();
    }
```

Nuestro código está ahora terminado: hemos codificado el problema completo.

c. Programa principal

La última etapa es la creación del problema principal. Empezamos creando una nueva clase **`Aplicacion`** que debe implementar la interfaz `IHM` y, por lo tanto, su método `MostrarResultado()`. Se muestra simplemente la distancia del camino encontrado y, a continuación, el recorrido en la consola.

```
import java.time.Duration;
import java.time.LocalDateTime;

// Uso de los algoritmos sobre ejemplos de mapa
public class Aplicacion implements IHM {
    // Programa main
    public static void main(String[] args) {
        // Completado más tarde
    }

    // Método que proviene de la interfaz, para visualizar el resultado
    @Override
    public void MostrarResultado(String ruta, double distancia) {
        System.out.println("Ruta (tamaño: " + distancia + "):
" + ruta);
    }
}
```

Necesitamos ejecutar el algoritmo escogido a partir de su nombre. Para ello, crearemos un primer método **`EjecutarAlgoritmo`**. Este recibe como parámetro el nombre del algoritmo y a continuación el grafo que representa el problema. Guardamos, para realizar un estudio comparativo, el tiempo de ejecución (calculado mediante `LocalDateTime` antes y después de llamar al método `Resolver` del algoritmo). Mostramos el nombre del algoritmo y a continuación la duración en milisegundos.

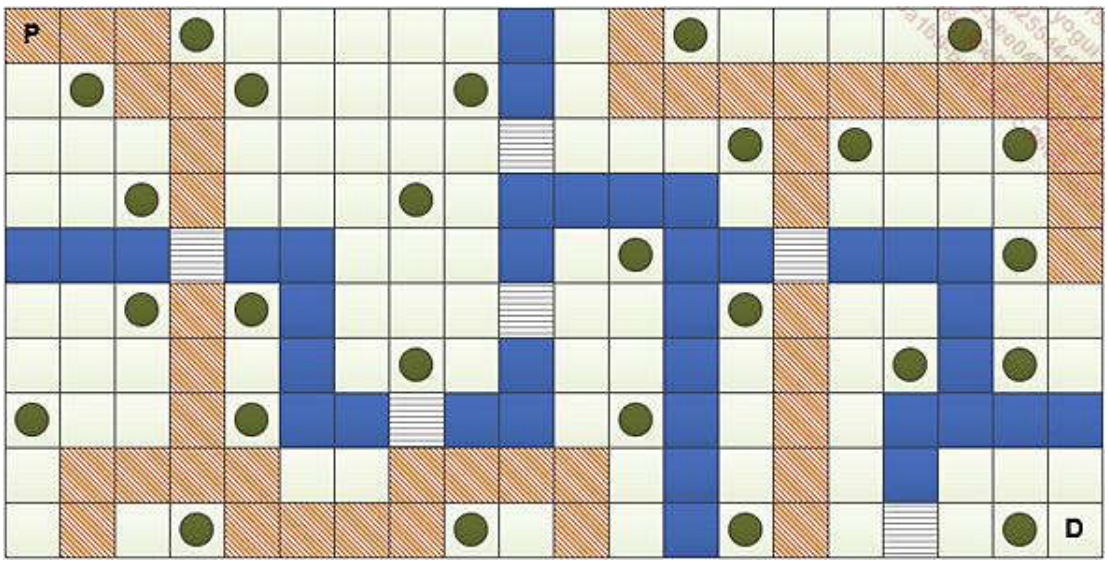
```
private void EjecutarAlgoritmo(String nombre, Grafo grafo) {
    // Inicialización
    LocalDateTime inicio;
    LocalDateTime fin;
    Duration duracion;
    Algoritmo algo = null;

    // Creación del algoritmo
    switch(nombre) {
        case "Profundidad" :
            algo = new BusquedaEnProfundidad(grafo, this);
            break;
        case "Anchura" :
            algo = new BusquedaEnAnchura(grafo, this);
            break;
        case "Bellman-Ford" :
            algo = new BellmanFord(grafo, this);
            break;
        case "Dijkstra" :
            algo = new Dijkstra(grafo, this);
            break;
        case "A*" :
            algo = new AStar(grafo, this);
            break;
    }

    // Resolución
    System.out.println("Algoritmo: " + nombre);
    inicio = LocalDateTime.now();
    algo.Resolver();
    fin = LocalDateTime.now();
    duracion = Duration.between(inicio, fin);
    System.out.println("Duración (ms): " + duracion.toMillis() + "\n");
}
```

Como vamos a querer comparar los cinco algoritmos, escribimos un método que permite ejecutarlos uno tras otro.

```
// Ejecución de todos los algoritmos seguidos
private void EjecutarAlgoritmos(Grafo grafo) {
    EjecutarAlgoritmo("Profundidad", grafo);
    EjecutarAlgoritmo("Anchura", grafo);
    EjecutarAlgoritmo("Bellman-Ford", grafo);
    EjecutarAlgoritmo("Dijkstra", grafo);
    EjecutarAlgoritmo("A*", grafo);
```

```
}
```

Podemos, ahora, implementar el método principal `Ejecutar()` que va a crear el mapa y, a continuación, ejecutar los distintos algoritmos. Para ello, crearemos el mapa correspondiente al problema presentado en este capítulo, y un segundo mapa algo más grande y complejo que mostramos a continuación:



Los mapas se representarán en ASCII, reemplazando cada casilla por el carácter correspondiente.

El código de este método es el siguiente:

```
// Ejecutar sobre ambos problemas
private void Ejecutar() {
    // Caso primer mapa
    String mapaStr = "..  XX   .\n"
                   + "*.  *X  *.\n"
                   + " .  XX ...\n"
                   + " .* X *.* \n"
                   + " ...=...  \n"
                   + " .* X     \n"
                   + " .  XXX*  \n"
                   + " .  * =   \n"
                   + " .... XX  \n"
                   + "   *.  X* ";
    Mapa mapa1 = new Mapa(mapaStr, 0, 0, 9, 9);
    EjecutarAlgoritmos(mapa1);

    // Caso segundo mapa
    mapaStr = "...*     X .*    *  \n"
            + " *..*   *X .........\n"
            + "   .     =   *.*  *.\n"
            + "  *.   * XXXX .    .\n"
            + "XXX=XX   X *XX=XXX*.\n"
            + "  *.*X   =  X*.  X  \n"
            + "   . X * X  X . *X* \n"
            + "*  .*XX=XX *X . XXXX\n"
            + " ....  .... X . X   \n"
```

```
                + " . *....* . X*. = * ";
        Mapa mapa2 = new Mapa(mapaStr, 0, 0, 9, 19);
        EjecutarAlgoritmos(mapa2);
    }
```

Terminamos con el contenido de `main`, que crea un objeto `Aplicacion` y ejecuta el método `Ejecutar`:

```
    public static void main(String[] args) {
        System.out.println("Búsqueda de rutas");
        Aplicacion app = new Aplicacion();
        app.Ejecutar();
    }
```

El programa está completamente operacional, es fácil probarlo sobre nuevos mapas. Podríamos imaginar la posibilidad de cargarlos desde un archivo de texto, por ejemplo.

5. Comparación del rendimiento

A continuación vamos a comparar el rendimiento de nuestros distintos algoritmos.

El tiempo de ejecución depende en gran medida de la máquina en la que se ejecuten los algoritmos. No son más que una indicación orientativa, puesto que el orden de los algoritmos, desde el más rápido hasta el más lento, es el mismo en una máquina u otra. Además, los tiempos indicados son valores medios obtenidos a partir de 100 ejecuciones en un ordenador portátil con un procesador Intel Core i5-4210U y 8 GB de RAM.

En primer lugar, los algoritmos de búsqueda en profundidad y en anchura encuentran caminos, pero no son óptimos. En efecto, encuentran respectivamente caminos de 54 y 32 puntos de acción para el primer mapa (valor óptimo 27) y 88 y 54 para el segundo mapa (valor óptimo 49).

Por el contrario, en ambos casos, el algoritmo A* encuentra la ruta óptima. En efecto, nuestro heurístico subestima (o se corresponde con) la distancia hasta el destino, lo que garantiza encontrar la ruta más corta. Sin embargo, este no es necesariamente el caso según el problema en curso.

En lo relativo a los tiempos de ejecución, obtenemos los siguientes valores medios:

	Primer mapa	Segundo mapa
Profundidad	0,78 ms	1,41 ms
Anchura	0,94 ms	1,09 ms
Bellman-Ford	0,94 ms	0,78 ms
Dijkstra	1,09 ms	0,94 ms
A*	0,78 ms	1,56 ms

A pesar de las 100 o 200 casillas y los cientos de arcos, observamos que todos nuestros algoritmos terminan en menos de 2 ms, lo cual resulta un tiempo de resolución muy rápido.

El orden de visita de los nodos para el recorrido en profundidad no se ha definido tras una reflexión, sino al azar desde el código. Vistos los resultados, comprobamos que este orden no está adaptado al problema. En

contraposición, el algoritmo resulta rápido en el primer mapa (y mucho menos en el segundo, pues se nota más la falta de adaptación).

La búsqueda en anchura encuentra caminos globalmente más cortos que la de profundidad, puesto que todos los arcos tienen prácticamente el mismo coste, lo que hace que sea menos sensible frente al orden de recorrido. Además, es más rápida en el segundo mapa, también gracias a esta particularidad que le permite encontrar la salida más rápido.

De los tres algoritmos de búsqueda de rutas óptimas, vemos que Bellman-Ford es el más eficaz sobre el mapa grande. En efecto, este algoritmo no ordena los nodos para encontrar el más próximo, a diferencia de Dijkstra y A*, lo que le permite ganar tiempo en mapas grandes.

Dijkstra y A* funcionan con el mismo principio, buscando el mejor nodo (y esto es lo que consume tiempo). Sin embargo, según el problema, uno u otro será el más eficaz. En el primer mapa, no hay trampas especiales: la ruta más corta sigue, globalmente, una línea recta, de modo que A* es el más rápido. En el segundo mapa, por el contrario, la ruta más próxima de la línea derecha se encuentra bloqueada justo antes de llegar al río. El camino más corto supone dar un rodeo al principio, y por este motivo el algoritmo A* es menos eficaz que Dijkstra (y resulta el más lento). Podemos destacar, también, que el punto de partida se encuentra en un ángulo.

De este modo, los algoritmos utilizan únicamente un cuarto de la zona alrededor del punto de partida, y es el cuarto correcto (el que contiene el destino). Si el punto de partida estuviera en mitad de una gran zona, Dijkstra no sería tan eficaz comparado con A* (que justamente solo procesa una parte del mapa).

Como conclusión:

- La búsqueda de una ruta en anchura es globalmente mejor que la búsqueda en profundidad, puesto que no es sensible al orden en que se recorren los nodos.
- Bellman-Ford es el algoritmo mejor adaptado para problemas sencillos.
- Dijkstra y A* son equivalentes en su conjunto: si existen trampas, como en el caso de un laberinto, por ejemplo, entonces Dijkstra está mejor adaptado; por el contrario, en una gran zona libre de obstáculos, debería darse preferencia a A*. Además, si el punto de partida está en el centro de la zona, A* permite limitar en gran medida la zona de recorrido.

Resumen

La búsqueda de rutas, o pathfinding, permite vincular los nodos de un grafo utilizando arcos predefinidos. Estos están asociados a una longitud (o coste). Es posible, también, buscar la ruta con el menor coste, siendo el coste un kilometraje, un tiempo o incluso un precio (por ejemplo, la gasolina consumida).

Existen varios algoritmos, cada uno con sus particularidades.

Cuando se intenta saber si existe un camino, sin buscar el más corto, podemos utilizar algoritmos de búsqueda exhaustiva en profundidad o en anchura. Si se sabe, de manera global, en qué dirección ir, la búsqueda en profundidad puede resultar interesante (siempre que se le precise bien el orden para recorrer los nodos vecinos).

La búsqueda en anchura produce generalmente mejores resultados y es, sobre todo, más genérica. En ambos casos, se avanza de nodo en nodo y se memorizan los nodos adyacentes descubiertos, que visitaremos posteriormente. Se diferencian en la estructura utilizada para almacenar los nodos vecinos: una pila para la búsqueda en profundidad y una cola para la búsqueda en anchura.

El algoritmo de Bellman-Ford permite encontrar el camino más corto, sea cual sea el grafo. Consiste en aplicar los arcos para calcular las distancias óptimas, mediante numerosas iteraciones. No es rápido en términos de cálculos, pero es fácil de implementar y puede resultar eficaz, puesto que no necesita ordenar ni buscar los elementos en una lista (a diferencia de los algoritmos siguientes).

El algoritmo de Dijkstra es algo más "inteligente" que el de Bellman-Ford, puesto que se aplica una única vez sobre cada arco, seleccionando cada vez el nodo más cercano al inicio que todavía no se haya utilizado más próximo al destino. De este modo, los cálculos se realizan una única vez. Sin embargo, según la manera en la que se codifique el algoritmo, este puede resultar menos eficaz, puesto que necesita encontrar el nodo más cercano entre una colección de nodos.

Por último, el algoritmo A*, con muy buena reputación principalmente en los videojuegos, utiliza un heurístico que permite estimar la distancia desde un nodo hasta el destino. Funciona igual que el algoritmo de Dijkstra, pero se utiliza la distancia total del camino (es decir, la distancia calculada desde el origen más la distancia estimada hasta el destino) que indica el nodo que debe tenerse en cuenta en la siguiente iteración. De manera global, en un mapa despejado, A* resulta más eficaz, mientras que en un laberinto Dijkstra presentará un mejor rendimiento.

Presentación del capítulo

La naturaleza ha encontrado la manera de resolver algunos problemas aparentemente sin solución. De esta manera, la vida está presente prácticamente en todos los lugares de la Tierra, desde lugares congelados hasta las fosas submarinas (que tienen las temperaturas más bajas y las presiones más elevadas) pasando por el aire.

Este éxito se explica por la potencia de la evolución biológica. Este hecho permite adaptar constantemente las diferentes especies de la mejor manera posible, para poder colonizar.

Los informáticos han imaginado cómo esta evolución se podría utilizar para responder a problemas complejos. De esta manera aparecieron los algoritmos genéticos.

En una primera parte, se explican los principios subyacentes a la evolución biológica. Son necesarios para entender el funcionamiento global y la inspiración de los algoritmos genéticos.

A continuación se presenta el funcionamiento de estos, al principio de manera global con un ejemplo y después volviendo a las principales etapas, con las correctas prácticas y las trampas que hay que evitar. También se explicará la co-evolución (la evolución conjunta de dos especies).

Los algoritmos genéticos se pueden utilizar en muchos dominios de aplicación, que se presentarán. A continuación, se ofrecen dos ejemplos de implementación en lenguaje Java.

Este capítulo termina con un resumen.

Evolución biológica

Los algoritmos genéticos se basan en la **evolución biológica**. Si bien no es necesario entender todos sus detalles, es importante entender la fuente de inspiración de estos algoritmos.

1. El concepto de evolución

La evolución biológica se estudió a partir de finales del siglo 18. En efecto, las pruebas de esta evolución se acumulan y los científicos quieren entender los fenómenos subyacentes.

A comienzos de siglo 19 aparece la **paleontología** (el término se utiliza desde 1822), ciencia que estudia los fósiles y las formas de vida que han desaparecido en la actualidad. Los científicos encuentran muchos esqueletos y los clasifican. Estos presentan muchas características comunes entre ellos o con formas de vida actuales. Por lo tanto, parece evidente que ha habido una continuidad y que las especies han sido progresivamente modificadas a lo largo de la historia.

Además, las sociedades basadas en la ganadería eran numerosas. Después de mucho tiempo, sabemos seleccionar los mejores individuos para mejorar la producción ganadera de una especie (como la leche de las vacas) o simplemente por puro placer. De esta manera, las razas de perros, gatos o caballos son más numerosas y con mayores diferencias. Es evidente que un animal es parecido a sus padres, aunque no es totalmente idéntico a ellos, y que seleccionando los mejores padres podemos obtener nuevas razas. Los caniches, los pastores alemanes, los cockers y los labradores descienden todos del lobo gris (Canis Lupus), domesticado durante la prehistoria. Es la intervención humana la que ha modelado todas estas razas.

Para terminar, los grandes descubridores iban de isla en isla y descubrían nuevas especies. Parecía que los individuos que vivían en las islas más cercanas también eran más parecidos físicamente. Por el contrario, eran mucho más diferentes a los individuos de otros continentes. Por lo tanto, las especies habían evolucionado de manera diferente, pero gradual.

Por lo tanto, la evolución biológica ya no era un tabú durante el siglo 19, sino una realidad científica. Sin embargo, faltaba por saber cómo había tenido lugar esta evolución.

2. Los causas de las mutaciones

Darwin (1809-1882) y Lamarck (1744-1829) pensaban de manera diferente respecto a las razones de las modificaciones entre los padres y los descendientes, llamadas **mutaciones**.

Para Darwin, un descendiente era una mezcla de sus padres, pero algunas veces, de manera aleatoria, parecían diferencias. Solo podían sobrevivir los individuos más adaptados y por lo tanto reproducirse, con lo que solo sus modificaciones se transmitían a sus descendientes. Por lo tanto, las mutaciones que no eran interesantes para la supervivencia no se propagaban y se extinguían. Por el contrario, aquellas que aportaban una ventaja selectiva, se conservaban y se propagaban.

Por su parte, Lamarck había propuesto otra teoría hacía varias décadas antes: la transmisión de caracteres adquiridos durante la vida del individuo. Por lo tanto, pensaba que estas variaciones eran una respuesta a una necesidad fisiológica interna. De esta manera, como los ancestros de las jirafas (del mismo tamaño que las gacelas) habían necesitado extender cada vez más su cuello para alcanzar las hojas de las árboles, su cuello se había alargado algunos centímetros o milímetros durante la vida del animal. A continuación, se había transmitido este alargamiento a sus descendientes, hasta alcanzar la longitud actual.

Han tenido lugar numerosos experimentos para entender las causas de estas mutaciones y anular o confirmar las hipótesis de Darwin y Lamarck. Cortando la cola del ratón durante de numerosas generaciones, Weismann observó

en 1888 que ninguno terminaba naciendo sin él, lo que entraba en contradicción con la transmisión de los caracteres adquiridos.

La tesis de Darwin parecía confirmarse (aunque oficialmente, su teoría se reconoció en el año 1930): durante la reproducción, se producían **mutaciones aleatorias** de vez en cuando. Solo los individuos para los que estas mutaciones eran beneficiosas, se convertían en más fuertes, resistentes o atractivos y de esta manera, se podían reproducir para transmitir esta nueva característica. Habitualmente y de manera incorrecta, esta teoría se simplifica diciendo "la supervivencia del más fuerte", cuando sería más correcto decir "la supervivencia del mejor adaptado".

Sin embargo, la manera en la que aparecían estas mutaciones era desconocida.

3. El soporte de esta información: los factores

Una vez reconocida la evolución y las mutaciones aleatorias, faltaba descubrir cómo la información genética se almacenaba en un individuo y cómo podía transmitirse.

Fue Gregor Mendel quién realizó el primer estudio complejo. Para esto, empezó estudiando la hibridación del ratón y después se centró en los guisantes a partir de mediados del siglo 19.

Seleccionó plantas de guisante con siete características bien distintas (como la forma o el color del grano), cada una con dos posibilidades. A continuación cruzó estas plantas durante varias generaciones y observó las plantas obtenidas. Dedujo lo que hoy llamamos las **leyes de Mendel** (en 1866).

Estas refutan la teoría de la mezcla propuesta por Darwin, complementado su teoría de la evolución (un hijo será la mezcla de sus padres). En efecto, durante la primera generación, todas las plantas eran "puras": no había ninguna mezcla. De esta manera, un cruce entre plantas de flores blancas y moradas, no provocaban el nacimiento de flores rosas, sino moradas.

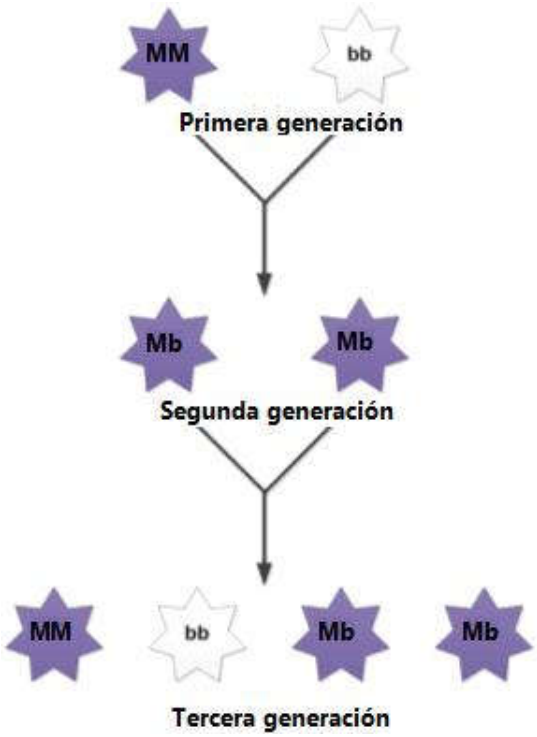
Las siguientes leyes indicaban que cada individuo tenía **factores** (dos de ellos) que solo tenían un valor posible, para un rasgo dado. Por lo tanto, su forma biológica exacta era desconocida. Durante de la reproducción, un único factor, elegido de manera aleatoria, se transmitía a los descendientes. De esta manera, los individuos que tenían dos factores morados cruzados con individuos de dos factores blancos, solo provocaban el nacimiento de individuos que contenían un factor morado y un blanco. El factor morado, **dominante**, daba el color a las flores. El factor blanco se llamaba **recesivo**.

Por lo tanto, su experiencia se puede resumir con la siguiente imagen. La primera generación está formada por plantas "puras": se trate de flores moradas después de varias generaciones, que por lo tanto tienen dos factores M (para Morado) o de flores blancas, que tienen dos factores b (para blanco). La mayúscula en Morado es una convención que indica que se trata de un factor dominante.

En biología, indicamos el factor blanco con m y no con b. En efecto, la convención consiste en elegir la inicial del factor dominante (aquí M para Morado), y ponerla en mayúsculas para el valor (llamado alelo) correspondiente y en minúscula, para el alelo recesivo (blanco para nuestras flores).

Por lo tanto, la segunda generación, resultado de un cruce de las plantas puras, solo tiene individuos que han heredado un factor M de un progenitor y b del otro. Por lo tanto, todos son Mb, siendo el morado dominante, todos parecen morados.

Durante la tercera generación, tenemos cuatro posibilidades. En efecto, el primer padre puede dar un factor M o b, como la segunda. Entonces tenemos un 25% de MM que son morados, un 25% de bb que son blancas y un 50% de Mb que son moradas (de hecho, el 25% de Mb y el 25% de bM, pero indicamos el factor dominante en primer lugar por convención).



Aquí vemos también por qué decimos que algunas características "saltan una generación" (se obtiene una flor blanca en la tercera generación aunque no la tengamos en la segunda). De hecho se trata de trazos recesivos.

Con las leyes de Mendel, se fijan las bases de la genética.

4. Los factores del código genético

Desafortunadamente, la comunidad científica no conoció rápidamente los trabajos de Mendel. Otros científicos continuaron sus trabajos sobre el almacenamiento de la información genética y se encadenaron los descubrimientos con gran rapidez.

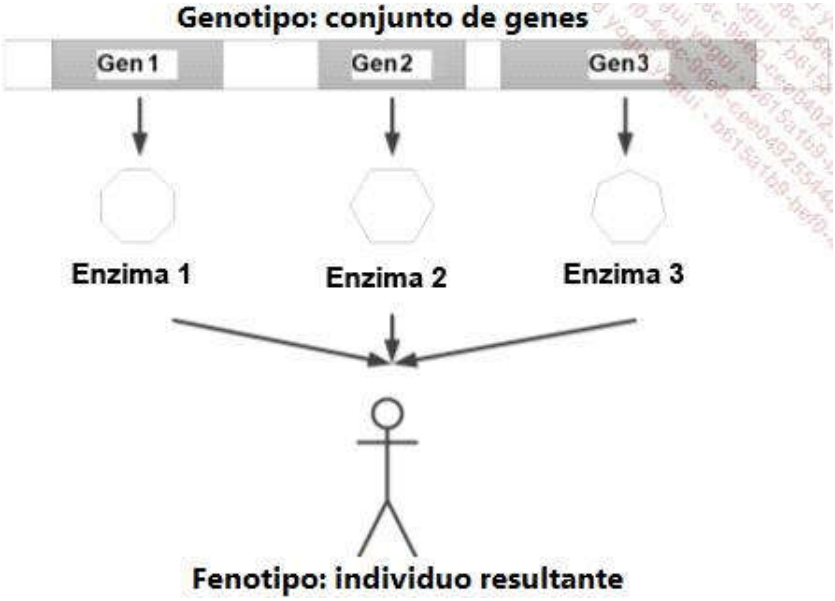
De esta manera Flemming aisló el **ADN** en 1869 y después los **cromosomas** en 1879. En 1900, las leyes de Mendel fueron re-descubiertas por parte de varios descubridores de manera independiente. Por tanto, parecía evidente que era en el ADN de los cromosomas donde se situaban los factores de Mendel.

Desde 1909 hablamos de **genes** en lugar de factores, y se propuso un primer mapa de los genes de la drosófila (su cromosoma X) en el año 1913. La estructura del ADN en doble hélice se descubrió en 1952 por Watson, Crick y Wilkins. El **código genético** fue admitido en los años 1960.

Así entendemos mejor el paso de los genes a las encimas. Se hace en dos momentos: la **transcripción**, que transforma el ADN en ARN (una especie de negativo del ADN que se puede mover hasta la producción de la encima) y después la **traducción**, que permite de pasar del ARN a los aminoácidos que forman la proteína.

Ya se han presentado los principios básicos de la **genética**: un individuo tiene cromosomas, que contienen genes. Cada gen se corresponde con una enzima, gracias a un código que indica la composición de la misma. El conjunto de los genes de un ser vivo se llama su **genotipo**. Las interacciones entre todas las enzimas crean el individuo, llamado **fenotipo**.

A continuación se muestra un esquema resumen:



Los trabajos no se detienen en los años 1960. Después, el proceso se ha entendido mejor, porque de hecho es mucho más complejo. Por ejemplo, ahora sabemos que un único gen, en función de las características del entorno, permite obtener diferentes enzimas, conservando o aumentando de manera selectiva algunas zonas del código de la enzima antes de la creación de esta. Sin embargo, los algoritmos genéticos no utilizan estos procesos, y por lo tanto, solo es necesario entender los principios básicos.

5. El "ciclo de la vida"

En resumen, un individuo tiene un conjunto de genes (el genotipo), presentes al mismo tiempo en doble copia. La transcripción y después la traducción permiten transformar estos genes en enzimas, que van a interactuar entre ellas para crear al ser vivo (el fenotipo).

Durante la reproducción, va a dar a su descendiente la mitad de su capital genético, que se mezclará con el capital genético del segundo padre. Además, durante este proceso, se pueden producir mutaciones aleatorias.

El individuo creado de esta manera, se va parecer a sus padres, siendo ligeramente diferente a ellos. Según las mutaciones que se hayan producido, podrá estar más o menos adaptado que sus padres para sobrevivir en su entorno.

Si está más adaptado, tendrá más oportunidades de sobrevivir, será más resistente o más atractivo y por lo tanto, a continuación se podrá reproducir. Al contrario, si las mutaciones sufridas le hacen menos adaptado, tendrá más dificultades para sobrevivir. Las causas son numerosas: muerte prematura del individuo, debilidad, mala resistencia a las enfermedades, dificultades para alimentarse o desplazarse, falta de atractivo para el sexo opuesto, etc...

Por lo tanto, la **selección natural** va a beneficiar a las mutaciones y cruces de individuos interesantes para la supervivencia de la especie. Estos se van a diseminar en la población y la especie continuamente va a mejorar y adaptarse a su entorno.

Podemos resumir este "círculo de la vida" con la siguiente imagen:


Padres
Reproducción
Hijos
Supervivencia
Adultos
Selección

Evolución artificial

La evolución biológica que hemos visto anteriormente se llama "incorpóreo": en efecto, los principios de reproducción, supervivencia o selección no indican cómo se debe almacenar o transmitir la información, ni siquiera lo que debe evolucionar.

Por lo tanto, despierta el interés de dominios muy diversos, ya sea del ámbito de la economía, la sociología, la música, etc... La informática no es una excepción y esta evolución biológica se puede utilizar para crear una evolución artificial, que permita resolver problemas irresolubles con los métodos más clásicos.

1. Principios

Por lo tanto, los algoritmos evolutivos van a partir de una población de soluciones potenciales para un problema. Cada uno se evalúa, para asignarle una nota, llamada Fitness. Cuando más fuerte sea la Fitness de una solución, más prometedora es.

A continuación, se seleccionan los mejores individuos y se reproducen. Se utilizan dos operadores artificiales: el cruce entre dos individuos, llamado **crossover** y de las **mutaciones** aleatorias.

A continuación se aplica una etapa de **supervivencia** para crear la nueva generación de individuos.

Por lo tanto, el proceso es el siguiente:

Aparecieron diferentes variantes en los años 60 de manera independiente. Holland puso a punto los **algoritmos genéticos**, pero también hablamos de **programación evolutiva** (Fogel) o de **estrategias de evolución** (Rechenbert y Bäck). Algunos años más tarde aparecieron la **evolución gramatical** (Ryan, Collins y O'Neill) y la **programación genética** (Koza).

Todos estos algoritmos, normalmente recogidos bajo el nombre genérico de algoritmos genéticos, se basan en este principio de evolución y este bucle generacional. Las diferencias se centran principalmente a nivel de las representaciones y de los operadores.

2. Convergencia

La convergencia hacia la solución óptima se demuestra de manera teórica. Sin embargo, nada indica el tiempo necesario para converger hacia esta solución, que por lo tanto, puede ser superior a lo que es aceptable.

Por lo tanto, es importante elegir correctamente los diferentes operadores (selección, mutación, crossover y supervivencia) y las tres representaciones existentes: los genes, los individuos y la población. En efecto, estas opciones pueden tener una gran influencia en la velocidad de convergencia hacia una solución óptima o casi óptima.

3. Ejemplo

Antes de entrar en los detalles de las diferentes fases, nos centramos en un ejemplo. Se utilizará aquí un mini-algoritmo genético para descubrir una combinación en Mastermind.

a. Juego del Mastermind

El juego del Mastermind está fechado en el año 1972. Se enfrentan dos jugadores:

- El primero va a definir una combinación de 4 colores entre 8 colores disponibles, sin utilizar dos veces el mismo. Por lo tanto, ¡hay 8! combinaciones (leer "factorial 8"), es decir 40.320 posibilidades.
- El segundo tiene un número de intentos definido para encontrar esta combinación. Para esto, a cada combinación propuesta el adversario le dará los índices en forma de peones negros y blancos. Un peón negro indica que uno de los peones de la combinación es del color correcto y están en el lugar adecuado, un peón blanco significa que uno de los peones es del color correcto, pero en mala posición.

Poco a poco, el segundo jugador va a afinar su combinación hasta encontrar la correcta. Si no lo consigue en el número de intentos estipulado, pierde (en caso contrario, gana).

Aquí, para el ejemplo, nos importa el número de intentos; supondremos que es infinito. Para los colores, en cada peón se añade la inicial de su color según la siguiente nomenclatura:

- B: Blanco
- C: Cián
- A: Amarillo
- M: Malva
- N: Negro
- O: Naranja
- R: Rojo
- V: Verde

Además, la combinación que trataremos de encontrar será la siguiente:

Los círculos grises a la derecha servirán para indicar los índices blancos y negros, que indican los peones del color correcto.

b. Creación de la población inicial

La primera etapa de un algoritmo genético es crear una población inicial, la mayor parte de las veces aleatoria. Aquí sacaremos cinco combinaciones al azar, lo que formará nuestra primera generación. Los individuos se numerarán desde I1 hasta I5.

Por lo tanto, la población inicial es la siguiente:

c. Función de evaluación

A lo largo de las generaciones va a ser necesario numerar a los individuos. Aquí, haremos una evaluación en dos momentos:

- Establecimiento de los índices (peones negros y blancos).
- Transformación de los índices en nota. Para esto, un peón negro (color correcto en la posición correcta) valdrá 2 puntos y un peón blanco (color correcto en la posición incorrecta), valdrá 1 punto.

Por lo tanto, los individuos tendrán todos notas entre 0 (ningún peón del color correcto) y 8 (combinación ganadora).

Por lo tanto, a continuación se muestra la evaluación de nuestra población inicial:

Vemos que nuestros individuos tienen notas que van de 1 a 4 puntos, y una media de 2,6 puntos.

d. Fase de reproducción

Esta fase es la más compleja. En efecto, será necesario:

- Seleccionar los padres.

- Hacer el crossover si es necesario, es decir, mezclar la información de los padres seleccionados para hacer uno nuevo.
- Crear mutaciones.

Por lo tanto, vamos a crear cinco descendientes. Para seleccionar los padres, vamos a hacer tres descendientes con dos padres y dos descendientes con un único padre. Cuando un individuo (de I1 a I5) tiene una nota correcta, más grande es su probabilidad de ser seleccionado como padre.

A continuación se muestra lo que resulta de una tirada como padres para nuestra nueva población:

- 6: I5 (un único padre)
- 7: I2 x I3 (dos padres)
- I8: I4 x I5 (dos padres)
- I9: I2 x I1 (dos padres)
- I10: I5 (un único padre)

Cuando un individuo tiene un único padre, no hacemos crossover. En caso contrario, se tomará la primera parte del primer padre y la segunda parte del segunda padre (dos peones por padre). Por ejemplo, para I7 guardamos sus dos padres. I2 contienen "A N M V" y I3 "V C O B". Tomando los dos primeros peones de I2 y los dos siguientes de I3, tenemos "A N O B", que será I7.

Por lo tanto, se obtienen los siguientes individuos antes de la mutación:

La última etapa es la de las mutaciones. Para esto, se cambiará de manera aleatoria algunos colores. Aquí, solo dos individuos sufrirán mutaciones: I8 (B se transforma en V) e I10 (O se tansforma en R).

Terminamos por evaluar los individuos, y por lo tanto se obtienen los siguientes hijos:

Nuestra población de hijos está lista. Observe que los descendientes tienen notas que van desde 2 hasta 5 puntos, con una media de 3,4 puntos. Por lo tanto, nuestros descendientes son mejores que los padres.

e. Supervivencia y encadenamiento de las generaciones

Una vez que se crean los descendientes, nos encontramos con una población de diez individuos: cinco padres (I1 a I5) y cinco descendientes (I6 a I10). La fase de supervivencia va a permitir encontrar una población de cinco individuos, para conservar un tamaño constante.

Aquí, simplemente se podría conservar los cinco hijos, que se convertirían en los cinco adultos. Por lo tanto, a continuación se seleccionaría entre ellos los padres de los cinco futuros descendientes (de I11 a I15).

A continuación se volvería a comenzar por la creación de los descendientes, sus evaluaciones, la fase de supervivencia y así sucesivamente.

f. Terminación del algoritmo

Para cualquier algoritmo genético, hay que seleccionar un punto de parada. Aquí, simplemente se podría parar cuando un individuo obtenga la nota de 8 (se ha encontrado la combinación correcta).

Cuando no se conoce la situación óptima, podemos parar al cabo de un número de generaciones fijado inicialmente o cuando se obtengan más mejoras de una generación a otra.

Globalmente, si los argumentos y operadores se eligen correctamente, sabemos que se encontrará la situación óptima con muchas menos evaluaciones que si tenemos que recorrer todo el espacio. De esta manera, aquí se podría encontrar la combinación correcta con un centenar de evaluaciones, sobre las 40.320 combinaciones posibles.

Nuestro ejemplo ha terminado. Cada una de las fases y de las opciones que hay que realizar, se van a detallar a continuación.

Primeras fases del algoritmo

Antes de pasar al cálculo de las generaciones, hay que empezar crear la primera. Para esto, será necesario hacer opciones sobre las representaciones y después, inicializar los individuos de la población inicial. Además, será necesario seleccionar su función de evaluación.

1. Elección de las representaciones

Como sucede con muchas de las técnicas de inteligencia artificial, la elección de las representaciones es primordial para limitar el espacio de búsqueda y para hacerlo lo más adaptado posible al algoritmo elegido.

a. Población e individuos

La **población** contiene una lista de individuos. El lenguaje informático utilizado impone algunas veces la representación de esta lista. Para facilitar la etapa de reproducción, es más fácil seleccionar una estructura de datos con un acceso directo a un individuo como una tabla, respecto a un acceso por recorrido como una lista.

Los **individuos** contienen una lista de genes. De nuevo aquí el formato exacto de esta lista se determina en parte por el lenguaje. Un acceso directo a uno de los elementos no es obligatorio.

b. Genes

La representación de los genes es aquella sobre la que hay que pasar la mayor parte del tiempo. Tradicionalmente, se trata de una **lista ordenada** de valores. Esto significa que para todos los individuos, el primer gen tiene el mismo significado (en nuestro ejemplo anterior, los peones se almacenaban en el mismo orden).

Sin embargo, en algunos casos puede ser más adecuado seleccionar una representación en la que la posición de los genes es variable, adaptándose a los operadores. Cada gen contiene el nombre de la variable asociada y el valor (por ejemplo, longitud y altura para un objeto).

Además, es importante reflexionar bien sobre las variables necesarias para resolver el problema. Se desaconseja tener demasiados valores a optimizar y por lo tanto, es necesario verificar que no hay variables redundantes, cuyos valores pudieran ser resultado de otros.

Para terminar, es más complejo para un algoritmo genético resolver un problema, en el que los diferentes genes están relacionados entre ellos. Una analogía puede ser la de la diferencia entre una batidora y un mezclador. En los dos casos, tenemos un grifo que permite de obtener agua a diferentes temperaturas y con diferentes caudales.

En el caso del mezclador, tenemos dos grifos: uno para el agua fría y otro para caliente. Es complicado modificar el caudal sin modificar la temperatura, porque girando uno de los grifos, se modifican las dos características del agua. Aumentar la cantidad sin modificar la temperatura es complejo. En el caso de la batidora, tenemos un único grifo. Si lo abrimos, aumentamos el caudal, pero si se cierra, cambiaríamos la temperatura. Las dos variables son independientes y de esta manera es más fácil de ajustar este tipo de grifo.

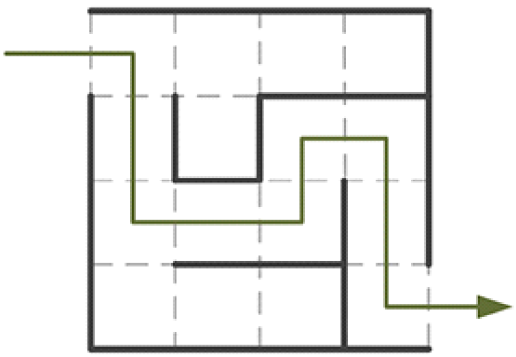
Para un algoritmo genético se observa lo mismo: cuanto más desacoplados estén los genes, más fácil es converger hacia la respuesta correcta.

c. Casos complejos

Hay casos donde las opciones de representación son más complejas. Tomemos el caso de un laberinto: queremos encontrar cómo salir. Por lo tanto, cada individuo representa una sucesión de instrucciones (arriba, abajo,

izquierda, derecha), y el objetivo inicial es llegar al final de este último.

A continuación se muestra un pequeño ejemplo de laberinto y su solución:



En primer lugar es necesario seleccionar si las direcciones son absolutas o relativas. En el caso de direcciones relativas, cada cambio depende de la dirección actual. De esta manera, si estamos yendo hacia la derecha y tenemos una dirección "izquierda", de repente estaremos yendo hacia arriba. El siguiente esquema ilustra este principio:

Al contrario, en caso de una dirección absoluta, se indica la dirección deseada, en el ejemplo anterior se trataría de "arriba".

A continuación es necesario seleccionar si las direcciones tienen un valor para una única casilla o hasta el próximo cruce. Si hay que ir tres veces a la derecha para alcanzar el próximo cruce, tendremos tres veces la misma instrucción en el primer caso, pero una única vez en el segundo.

De esta manera, el siguiente camino indica [D, D, D] en el primer caso y solo [D] en el segundo:

Por lo tanto, a continuación se muestran las representaciones del camino que permitiría salir del laberinto del ejemplo en los cuatro casos presentados (instrucciones relativas o absolutas, y para una casilla o hasta un cambio). Cada dirección se representa por su inicial (D para derecha, I para izquierda, A para arriba y B para abajo).

Ámbito → **Direcciones ↓**	**Una casilla**	**Varias casillas (hasta el próximo cruce)**
Absolutas	[D, B, B, D, D, A, D, B, B, D]	[D, B, D, A, D, B, D]
Relativas	[D, D, A, I, A, I, D, D, A, I]	[D, D, I, I, D, D, I]

En caso de un algoritmo genético, las direcciones absolutas son más interesantes. En efecto, esto desacopla el

significado de los diferentes genes, el sentido de uno ya no depende del de los anteriores. Una mutación del primero a la "derecha" que cambia a "arriba", no modificaría el resto del trayecto. Es más sencillo para un humano dar direcciones absolutas mirando el trayecto, que pensando cada vez en la dirección en la que vamos en un determinado momento.

Además, con el objetivo de limitar al máximo el número de genes, parece más oportuno conservar una dirección hasta un cruce (o un muro). En este pequeño ejemplo, de esta manera se pasa de 10 a 7 genes, es decir una ganancia del 30%, ganancia que se repercutirá en la velocidad de convergencia.

2. Inicialización de la población inicial

Durante la fase de inicialización, se crea una primera población. Para la mayor parte de los problemas, se parte de soluciones aleatorias que por lo tanto, de media están muy poco adaptadas al problema.

Si ya conocemos soluciones aceptables al problema, es posible inyectarlas directamente durante la inicialización. Entonces, el proceso completo es más rápido, y necesita menos generaciones. Sin embargo, hay que tener cuidado con conservar una diversidad en la población inicial (por lo tanto, no es necesario hacer una primera generación de clones).

Si conocemos menos soluciones aceptables que individuos en la población, de esta manera se puede completar con individuos aleatorios, en lugar de tener que duplicar las soluciones.

3. Evaluación de los individuos

La elección de la **función de evaluación** (o función de Fitness) es primordial, ya que es esta la que indica cuál es el objetivo que se tiene que alcanzar o al menos, en qué dirección hay que ir.

Esta función se puede calcular a partir de los datos contenidos en los genes con funciones matemáticas, pero este no siempre es el caso. En efecto, también puede:

- Darse por un sistema externo, que "comprobaría" la solución propuesta.
- Asignarse por un humano, que juzgaría la calidad de la solución.
- Obtenerse después de la simulación del individuo creado (que puede ser un programa informático, un comportamiento en un robot, etc...).
- O conocerse después una prueba real, por ejemplo durante la creación de piezas mecánicas.

La única restricción real es que la función de Fitness debe permitir diferenciar los individuos buenos de los malos.

Sin embargo es necesario prestar atención a que mida correctamente el objetivo buscado, porque el algoritmo busca maximizarlo y algunas veces puede dar resultados sorprendentes.

Por ejemplo para el laberinto, si calificamos a los individuos por el número de casillas recorridas, se corre el riesgo de favorecer a los individuos que van a hacer idas y vueltas entre dos casillas hasta el infinito o a los bucles. Si medimos la distancia a la salida, corremos el riesgo de llevar a nuestros individuos a un callejón sin salida cerca de la salida. Por lo tanto, seleccionar una función correcta algunas veces es complejo.

Para terminar, la función seleccionada debe ser lo más continua posible (en sentido matemático): no debe presentar niveles demasiado grandes. En efecto, hay que "guiar" progresivamente a los individuos hacia las soluciones más óptimas. Por lo tanto, debe ofrecer valores gradualmente crecientes, sin estancarse ni sufrir diferencias bruscas.

De esta manera, seleccionar una función de Fitness para nuestro laberinto que asignaría 0 a los individuos que

permanezcan en el laberinto y 50 a aquellos que puedan salir, no podría llevar a la convergencia y por lo tanto, sería necesario contar con el puro azar para encontrar una solución válida. En efecto, nada indicaría al algoritmo que está mejorando y se acerca al objetivo.

Se puede hacer una analogía con el juego del "frío o caliente ", en el que se indica a una persona dónde se encuentra un objeto. Gracias a una escala gradual que va desde "congelado" hasta "te quemas ", pasando por todas las temperaturas, la persona puede encontrar el objeto. Si solo decimos sí o no, solo podría contar con la suerte o con un recorrido exhaustivo de todos los lugares, para poder encontrar el objeto.

Creación de las generaciones siguientes

Una vez que se crea la primera población y se evalúa, hay que seleccionar los diferentes operadores para pasar a la generación siguiente:

- Selección de los padres entre los adultos.
- Operadores de crossover y de mutación para crear los descendientes.
- Supervivencia para crear la nueva población a partir de los adultos y de los descendientes creados.

1. Selección de las padres

La selección consiste en determinar cuáles son los individuos que merecen ser elegidos como padres para la generación siguiente. Es necesario que en proporción, los mejores padres se reproduzcan más que los padres con un Fitness más bajo, pero cada uno debe tener una probabilidad no nula de reproducirse.

En efecto, haciendo mutar o crecer las soluciones "malas" en apariencia, podemos encontrar una correcta solución a un problema.

Los padres se pueden seleccionar de diversas maneras, deterministas o estocásticas. Las selecciones estocásticas (que hacen intervenir a una parte de azar), son las más utilizadas.

Una de las soluciones más habituales es utilizar una **ruleta sesgada**: cuanto más adaptado es un individuo, más sección de la ruleta. Por lo tanto, los individuos siguientes tendrán una parte cada vez más pequeña. Entonces un sorteo indica cuál es el individuo elegido.

Estadísticamente, los que tienen Fitness más elevadas tendrán más hijos, pero todos tendrán al menos una oportunidad de reproducirse, aunque esta sea baja.

La parte de la ruleta de cada individuo se puede determinar por el rango de este, teniendo el primero siempre la misma parte respecto a la segunda o su Fitness. En este último caso, las proporciones cambian en cada generación y un individuo mucho más adaptado que el resto de su población, tendrá muchos más descendientes para transmitir rápidamente sus genes. Al contrario, en una población uniforme donde las diferencias de Fitness son bajas, la ruleta dará casi a cada individuo la misma oportunidad de reproducirse.

La segunda solución, después de la ruleta, es utilizar el **torneo**: se elijen al azar dos individuos y el más adaptado es el que se reproducirá. Por lo tanto, los individuos más adaptados ganarán más habitualmente los torneos y se reproducirán más que los individuos poco adaptados, pero incluso aquí todos tienen oportunidades de reproducirse (excepto el individuo menos adaptado, que perderá todos sus torneos).

La tercera solución es utilizar un **método determinista**: se calcula el número de descendientes de cada individuo sin sorteo, sino mediante una fórmula matemática o con reglas elegidas con anterioridad. Por ejemplo, se puede decidir para una población de 15 individuos que el mejor individuo siempre tendrá 5 hijos, que el segundo tendrá 4, hasta el quinto que tendrá 1 hijo; los siguientes no se reproducirán.

Para terminar, se puede añadir una característica **elitista** a esta selección, que permita conservar al mejor individuo que por lo tanto, se clona para crear su propio descendiente, sin mutación. De esta manera, nos aseguramos de que jamás se pierda una solución correcta.

Sin embargo, no existe ninguna manera de conocer al operador de selección más adaptado a un caso dado. Por lo tanto, algunas veces será necesario hacer pruebas empíricas cambiando la solución obtenida. Sin embargo, puede observar que el elitismo, a parte de los problemas críticos y particulares, normalmente no es útil incluso es nefasto, para el algoritmo de problemas complejos. En efecto, conservando sistemáticamente una respuesta correcta puede

dejar a un lado una respuesta mejor.

2. Reproducción

Durante la reproducción, se elige para cada hijo desde 1 hasta N padres. La información genética de los diferentes padres se mezclan con el operador de crossover.

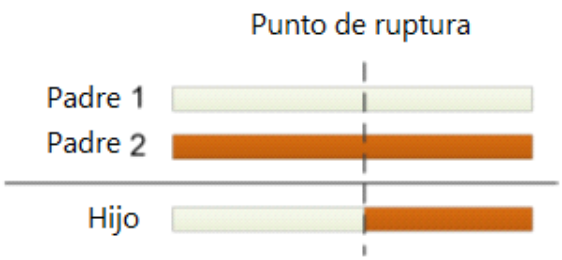
Una vez se haya realizado la mezcla de los padres, se aplican al resultado las mutaciones elegidas de manera aleatoria, y cuyo número depende de la tasa de mutación del algoritmo.

a. Crossover

El **crossover**, algunas veces llamado "operador de cruce", permite crear un nuevo descendiente a partir de sus dos ascendentes, mezclando la información genética.

Un descendiente puede tener solo un único padre (por lo que no hay crossover), tener dos (es el caso más clásico) o más. De nuevo aquí, es el diseñador el que tiene que seleccionar. Aquí hablaremos solo de los crossovers entre dos padres, pero los principios enunciados se pueden generalizar fácilmente para tres padres o más.

El crossover más habitual consiste en tomar un punto de corte en el genoma. Todos los genes situados antes de este punto vienen del primer padre, y los situados después, del segundo. También es el operador más cercano a la realidad biológica. Se llama **discreto** porque conserva los valores sin modificaciones, tal cual.



Sin embargo se pueden imaginar variantes, por ejemplo haciendo la media de los padres para cada gen. Entonces, el crossover se llama continuo. Matemáticamente consiste en tomar la mitad del segmento, representado por los dos padres.

Este operador no se utiliza forzosamente para cada reproducción: es posible crear descendientes con un único padre, sin necesidad de crossover. Por lo tanto, es necesario determinar la **tasa de crossover** del algoritmo, normalmente superior al 50%. De nuevo aquí es la experiencia y el problema los que guían las elecciones.

Sin embargo, el crossover no es válido y produce casi siempre malos descendientes en dos casos:

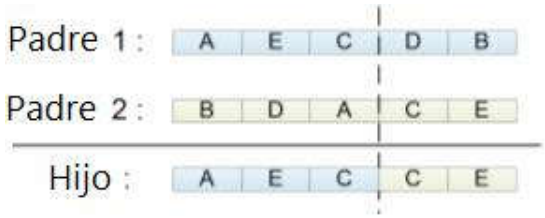
- Si los genes relacionados semánticamente están muy separados.
- Si los genes están forzados por el resto del genoma.

En efecto, en el primer caso, si los genes relacionados están separados, hay mucho riesgo de que un crossover los separe y dos individuos de calidad correcta, podrán dar lugar a descendientes de mala calidad, porque solo tengan una parte de la solución.

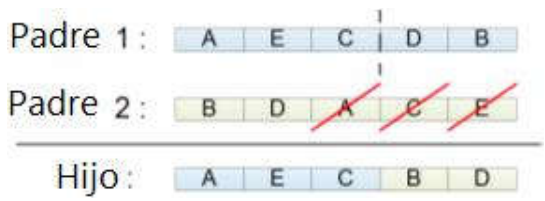
En el segundo caso, el de los genes forzados por el resto del genoma, imaginemos el problema del viajante de comercio. Se trata de un problema combinatorio clásico en el que un individuo debe visitar varias ciudades,

haciendo los menos kilómetros posibles. Por lo tanto, es necesario optimizar su camino.

El genoma de un individuo como este, se podría corresponder con la sucesión de las ciudades a visitar. Si tenemos cinco ciudades nombradas desde la A a la E, vemos que el cruce de los dos individuos siguientes no tendría sentido: algunas ciudades se visitarían dos veces (C y E) y otras ninguna (B y D).



En este caso, hay que adaptar el operador. Por ejemplo, puede tomar las N primeros ciudades del primer individuo y después, tomar las ciudades ausentes en el orden del segundo individuo.



Por lo tanto, el crossover está relacionado con la representación elegida para los genes. Por lo tanto, si la version discreta es la más habitual, no se puede aplicar sin comprobar que se corresponda con nuestro problema.

b. Mutación

El segundo operador local es el operador de **mutación**. Su objetivo es introducir novedades dentro de la población, para permitir descubrir nuevas soluciones potenciales.

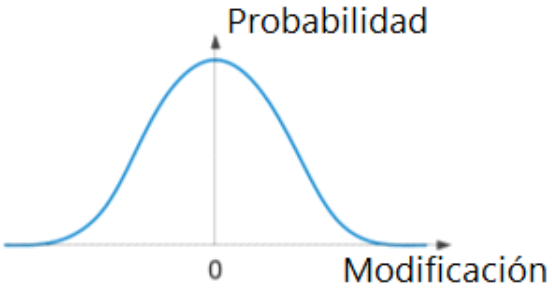
Por lo tanto, consiste en seleccionar de manera aleatoria algunos genes. La probabilidad de que un gen se modifique por una mutación, se llama la **tasa de mutación**. Si es muy elevada, las soluciones correctas corren el riesgo de desparecer. Si es muy baja, no permite encontrar nuevas soluciones rápidamente. Por lo tanto, es necesario encontrar el equilibrio correcto.

De nuevo aquí, en función del tamaño de la población, del número de genes o del problema, se seleccionarán tasas diferentes. Sin embargo, un buen comienzo consiste en partir de una tasa del 5% y adaptarla a continuación según las necesidades.

Respecto a su efecto, la mutación puede seguir una **distribución uniforme**: en este caso, el nuevo valor del gen se seleccionará al azar en todo el espacio posible. De esta manera, para nuestro problema Mastermind, nuestro nuevo color de peón se elige entre la lista de colores disponibles.

El operador también puede, y esto normalmente es más eficaz, modificar el valor actual por un valor cercano. Entonces sigue una **distribución normal** (también llamada "campana de Gauss"), centrada en 0 y que permita obtener la modificación a aplicar a la version actual.

A continuación, se muestra un ejemplo de distribución normal:



Este tipo de mutación permite, sobre todo si el espacio de búsqueda es grande, desplazarse gradualmente por él.

Además, en algunos casos las mutaciones pueden consistir en:

- Agregar o eliminar un gen al final o en medio del cromosoma.
- Duplicar un gen (en particular en el caso de un número variable de genes).
- Intercambiar la posición de dos genes (por ejemplo, para el viajante de comercio, cambiando el orden de dos ciudades).
- Etc.

Por lo tanto, es necesario adaptarse al problema que se tiene que resolver, como para el resto de operadores.

3. Supervivencia

Cuando se crean los descendientes, nos encontramos con la antigua población formada entre otros por los padres y la nueva generación. Sin embargo, solo es necesario conservar una única población.

La solución más simple consiste en una **sustitución** total de los adultos por los hijos. Por lo tanto, todos los individuos sobreviven durante una única generación.

Sin embargo hay otras soluciones posibles, como los **torneos** entre los individuos de las dos poblaciones, enfrentándose sistemáticamente un individuo de cada generación o eligiendo al azar en la población completa. Todos los individuos que se han seleccionado para formar parte de la nueva población, no pueden participar más en otros torneos.

De nuevo aquí, también se pueden utilizar **métodos deterministas**, que consisten en seleccionar los mejores individuos de las dos generaciones.

Sin embargo, aconsejamos seleccionar un método estocástico y uno determinista, para los operadores de selección y de supervivencia. Se esta manera, se puede seleccionar una ruleta sesgada (estocástica) para la selección y una sustitución (determinista) para la supervivencia. Dos métodos estocásticos o dos métodos deterministas, dan normalmente menos resultados correctos.

4. Terminación

Para terminar, es necesario definir un criterio de parada. Esto permite saber en qué momento detenerse, para dar al usuario las mejores soluciones encontradas. Este criterio puede extenderse a un número de generaciones o un Fitness mínimo a obtener por los individuos (un "marcador" a alcanzar). También puede extenderse al descubrimiento o no de mejores soluciones (siendo por ejemplo del tipo "si no se encuentra mejor solución durante X generaciones").

Es importante seleccionar el criterio de parada antes de lanzar las simulaciones y para comparar diferentes algoritmos. Además, trazando la evolución del Fitness a lo largo del tiempo, a continuación se puede ver si el criterio de parada termina la simulación demasiado pronto (se continúa mejorando las soluciones durante la parada) o demasiado tarde (hace varias generaciones que no hay ninguna mejora).

Coevolución

La **coevolución** es un fenómeno biológico formalizado en 1964, como consecuencia de un estudio de dos biólogos sobre los cruces entre plantas y algunas mariposas. En efecto, demostraron que las dos especies habían evolucionado conjuntamente: las mariposas comían de la planta, que ha desarrollado mecanismos para defenderse (veneno o protecciones físicas). Las mariposas desarrollaron medios para resistir a estos venenos o defensas.

Por lo tanto, se trata de un curso sin fin entre las dos especies, intentando superponer periodos de avances para asegurar su propia supervivencia y no pueden parar de evolucionar. Las dos especies en competición se han desarrollado y han evolucionado más rápido, que si cada una hubiera evolucionado de manera independiente.

Esta coevolución es la que se observa en todos los sistemas "presa - depredador". De esta manera, hay un flujo evolutivo entre los hackers, que intentan romper la seguridad de los sistemas informáticos, y los responsables de la seguridad, que intentan repeler e impedir los ataques. Cada campo debe evolucionar constante y rápidamente, para intentar conservar una ventaja sobre el otro campo y contrarrestar cualquier nueva amenaza/defensa establecida.

Esta presión evolutiva se puede utilizar en nuestro beneficio en un algoritmo genético. De esta manera, se puede hacer evolucionar no a una población sino a dos o más poblaciones de las que compiten (o casos raros que cooperan). De esta manera, se podría hacer evolucionar a los laberintos y a los algoritmos para recorrerlos de manera paralela.

De esta manera, en robótica se puede utilizar un algoritmo genético para aprender a andar, empezando por los escenarios más llanos (nuestras "presas" sencillas para empezar). Cuando el robot es capaz de desplazarse en superficies sencillas, entonces se podría hacer evolucionar el suelo para agregar obstáculos, formas de caminar, agujeros, etc. y de esta manera mejorar el proceso de marcha.

Esta coevolución implica la modificación de la función de evaluación: entonces, es necesario que un individuo de una especie sea evaluado por su interacción con los individuos de la otra especie. En el ejemplo de nuestro robot, los mejores suelos serían aquellos que consigan derrotar al robot y los mejores robots aquellos capaces de desplazarse sobre más tipos de terreno. A medida que avanzan las generaciones, la calidad de las dos poblaciones aumentaría.

En la práctica, esto sucede raramente, pero los resultados son mejores cuando el algoritmo empieza por resolver problemas sencillos y después cada vez más complejos, a medida que los problemas y las soluciones evolucionan.

Dominios de aplicación

El equipo de John Holland de la universidad del Michigan, empezó a trabajar en los algoritmos genéticos en los años 60. Sin embargo, esta tecnología solo empezó a conocerse a partir de año 1975, con la publicación de su libro.

Entonces, los algoritmos evolutivos en general empezaron a tocar muchos dominios. Para que sean eficaces, es suficiente con responder a algunas restricciones:

- El número de soluciones potenciales debe ser muy grande.
- No hay método exacto que permita obtener una solución.
- Una solución casi óptima es aceptable.
- Se puede evaluar la calidad de una solución potencial.

Si se cumplen estas cuatro restricciones, entonces un algoritmo genético puede ser una solución correcta para encontrar una respuesta al problema que, aunque no pueda ser garantía de ser la mejor, en todo caso será aceptable y esto en un tiempo razonable.

Los encontramos en dominios de la **ingeniería** y el **diseño**. En efecto, en la actualidad cada vez es más difícil crear piezas que respondan a las restricciones y que minimicen o maximicen algunas características (menos materia prima, consumo más bajo, potencia más importante, mejor resistencia, etc...). De esta manera, las carrocerías de los coches se pueden crear por medio de un algoritmo genético para hacerlas más aerodinámicas.

Su segundo gran dominio de aplicación es la **logística**. Aquí nuestros encontramos con problemas de tipo "viajante de comercio", pero también problemas de optimización con restricciones. Pueden servir para organizar mejor un almacén para limitar los desplazamientos, seleccionar los horarios de autobús, tren o avión más adaptados, mejorar una red informática organizando los diferentes nodos para limitar los cuellos de botella, crear horarios, etc... De esta manera los trenes suizos (Swiss Federal Railways o SBB), tienen horarios creados por algoritmos genéticos.

Los laboratorios de **biología** y **bioquímica**, también son grandes consumidores de algoritmos genéticos. En efecto, estos permiten ayudar a la secuenciación del ADN, el descubrimiento de nuevas proteínas, el cálculo de la forma replicada de una molécula, etc... Los ejemplos son muchos, en particular en los laboratorios de investigación, como el laboratorio de bio-informática comparada de Nuestra Señora en España.

La **finanza** es un dominio muy complejo. De nuevo aquí, esta técnica puede ayudar para mejorar las previsiones bursátiles, gestionar portfolios u optimizar sus inversiones. Hay libros enteros dedicados a la utilización de los algoritmos genéticos en este dominio, como "Genetic Algorithms and Investment Strategies" de ediciones Wiley.

También ayudan a la **creación**. Una utilización comercial es la creación de nuevos paquetes para productos diversos y variados, como Staples para sus embalajes de emboltorios de papel o Danone para el lanzamiento del producto Activia en los Estados Unidos. Fuera del marketing, también sirven para crear música o imágenes. Un equipo del INRIA (Instituto Nacional de Investigación Informática y automatización francés), creó una gama de bufandas cuyos diseños se obtuvieron por un algoritmo genético. Hay edificios que también fueron diseñados por algoritmos genéticos, como la escalera-biblioteca del "Rainbow Building" en Corea del Sur.

Para terminar, se pueden encontrar en dominios como los **videojuegos** (para crear comportamientos para los adversarios, niveles o personajes), o en dominios más originales como el **forense**. De esta manera, hay un programa que permite ayudar a un testigo a crear un retrato robot de un criminal. Este propone varias caras y el testigo elige en cada generación aquellas más parecidas según su recuerdos, lo que es más sencillo que seleccionar cada parte del cuerpo de manera separada.

Implementación

Ahora nos vamos a interesar por la implementación en Java de un algoritmo genético genérico, que se utiliza aquí para resolver dos problemas abordados anteriormente:

- El viajante de comercio, que consiste en encontrar la ruta más corta para unir un conjunto de ciudades.
- El laberinto, dando la sucesión de instrucciones que hay que seguir para ir desde la entrada hasta la salida.

El código que se propone aquí está disponible para su descarga. El programa que contiene la función `main` es una aplicación de consola.

1. Implementación genérica de un algoritmo

a. Especificaciones

Queremos codificar un motor genérico para un algoritmo genético, que a continuación se aplica a dos problemas diferentes, escribiendo el menor código posible para pasar de uno al otro.

Por lo tanto, es importante fijar correctamente las necesidades. El proceso evolutivo en sí mismo, el corazón del sistema, se ocupa de inicializar la población y después, lanza la evaluación y selección de los padres y la creación de los descendientes y para terminar, la supervivencia. A continuación nos centramos en la evaluación, hasta que se alcance un criterio de parada.

Por lo tanto, se va a definir dos criterios de parada posibles: se alcanza la Fitness que queríamos o se alcanza el número máximo de generaciones. En los dos problemas, se trata de minimizar la función de evaluación: el número de kilómetros para el viajante de comercio o la distancia a la salida para el laberinto. Por lo tanto, se fija un Fitness mínimo a alcanzar.

Se podría adaptar el algoritmo para permitir maximizar el valor de adaptación, pero como nuestros dos problemas buscan minimizar el Fitness, no lo haremos aquí.

Los diferentes argumentos del algoritmo se definen en una clase aparte. A continuación, tendremos interfaces o clases abstractas para los individuos y los genes. En efecto, se trata de las únicas dos clases que hay que redefinir para cada caso. Para el problema de salida del laberinto, necesitamos tener genomas de tamaño variable (la lista de las instrucciones), por lo que el algoritmo debe permitir gestionarlo.

Para aligerar el corazón del sistema de la gestión del caso a resolver, llevamos a una fábrica la creación de los individuos y la inicialización de su entorno.

Para terminar, definimos una interfaz para el programa principal. En efecto, en nuestro caso vamos a hacer salidas por la consola, pero fácilmente podríamos adaptar nuestro programa para agregar líneas en una tabla o mostrar gráficamente los mejores individuos.

b. Argumentos

Es necesario empezar definiendo una clase estática **`Argumentos`** que contiene todos los argumentos. Esta se inicializa con un valor por defecto, que es el valor normalmente aconsejado como punto de partida, y son accesibles desde las otras clases.

En primer lugar definimos los argumentos relativos a toda la población y al algoritmo en general (entre ellos, los criterios de parada):

- El número de individuos por generación, llamado `numIndividuos` y se inicializa a 20.
- El número máximo de generaciones, llamado `numMaxGeneraciones` y se inicializa a 50.
- El número inicial de genes si el genoma es de tamaño variable, llamado `numGenes` y se inicializa a 10.
- El Fitness que se debe alcanzar, llamado `minFitness` y se inicializa a 0.

A continuación se definen las diferentes tasas utilizadas durante la reproducción: la tasa de mutaciones (`tasaMutacion`) a 0.1, la tasa de adición de genes (`tasaAdicionGenes`) a 0.2, la tasa de eliminación de genes (`tasaElimGen`) a 0.1 y la tasa de crossover (`tasaCrossover`) a 0.6.

Se termina esta clase creando un generador aleatorio `random` que se podrá utilizar a continuación en todo el código, sin tener que volver a crearse. En caso de que queramos poder reproducir los resultados, es suficiente con indicar una semilla al generador aleatorio.

Por lo tanto, el código completo de esta clase es el siguiente:

```
import java.util.Random;

public class Argumentos {
   // Argumentos de la población y los individuos
   public static int numIndividuos = 20;
   public static int numGenes = 10;

   // Criterios de parada
   public static int numMaxGeneraciones = 50;
   public static doble minFitness = 0.0;

   // Tasa de las operadores
   public static doble tasaMutacion = 0.1;
   public static doble tasaAdicionGenes = 0.2;
   public static doble tasaElimGen = 0.1;
   public static doble tasaCrossover = 0.6;

   // Generador aleatorio
   public static Random random = new Random();
}
```

c. Individuos y genes

A continuación es necesario definir una interfaz para nuestros genes, llamada **`IGen`**. Solo contiene una función de mutación. No se indica nada sobre la manera de gestionar los genes, que depende del problema a resolver.

```
public interfaz IGen {
   void Mutar();
}
```

Los individuos se implementan con la clase abstracta **`Individuo`**. Aquí encontramos el código necesario para recuperar el Fitness de un individuo o su genoma.

A continuación se añaden dos funciones abstractas puras (por lo tanto, es necesario obligatoriamente redefinirlas en los descendientes): una para evaluar al individuo y otra para hacerlo mutar.

Para terminar, se utiliza una función `toString` para las visualizaciones. Nos basta con visualizar el valor del Fitness, seguido del genoma. Para esto, se utiliza la clase `StringJoiner` que permite transformar una lista en una cadena con el delimitador elegido.

Por lo tanto, se obtiene el siguiente código:

```
import java.util.ArrayList;
import java.util.StringJoiner;

public abstract class Individuo {
   protected doble Fitness;
   protected ArrayList<IGen> genoma;

   public doble getFitness() {
       return Fitness;
   }

   public abstract void Mutar();
   public abstract doble Evaluar();

   @Override
   public String toString() {
       String gen = "(" + Fitness + ")";
       StringJoiner sj = new StringJoiner(" - ");
       sj.add(gen);
       for(IGen g: genoma) {
           sj.add(g.toString());
       }
       return sj.toString();
   }
}
```

Para cada problema, una clase hija hereda de esta clase **`Individuo`**. La función de otra clase es seleccionar qué clase hija instanciar en función de las necesidades. Si dejamos esta opción en el núcleo del algoritmo, se pierde la generalidad.

Por lo tanto, se utiliza una fábrica de individuos, que es un singleton. De esta manera, es accesible desde cualquier parte del código, pero una única instancia se crea durante la ejecución del código.

Por lo tanto, el código básico de la fábrica **`FabricaIndividuos`** es el siguiente:

```
class FabricaIndividuos {
   private static FabricaIndividuos instance;

   private FabricaIndividuos() {}

   public static FabricaIndividuos getInstance() {
       if (instance == null) {
           instance = new FabricaIndividuos();
```

```
        }
        return instance;
    }
}
```

Esto permite inicializar el entorno de vida de los individuos con un método `Init()`, obtener el individuo deseado creado de manera aleatoria o a partir de uno o dos padres gracias a tres métodos `CrearIndividuo()`, que toman como argumento el problema a resolver en forma de una cadena y después, los padres potenciales.

```
    void Init(String tipo) {
        // ...
    }

    public Individuo CrearIndividuo(String tipo) {
        Individuo ind = null;
        // ...
        return ind;
    }

    public Individuo CrearIndividuo(String tipo, Individuo padre) {
        Individuo ind = null;
        // ...
        return ind;
    }

    public Individuo CrearIndividuo(String tipo, Individuo padre1,
Individuo padre2) {
        Individuo ind = null;
        // ...
        return ind;
    }
```

Su código se completará para cada problema particular.

d. IHM

Para separar el algoritmo genético de su utilización (aplicación de escritorio, móvil, en línea de comandos, en un sitio web, etc...), se define una interfaz para la IHM (llamada **IHM**). Se deberá implementar por todos los programas. Solo hay un único método, el que permite visualizar el mejor individuo de cada generación.

Por lo tanto, el método debe tomar como argumento al individuo y la generación.

```
public interfaz IHM {
    void MostrarMejorIndividuo(Individuo ind, int generacion);
}
```

e. Proceso evolutivo

La clase principal, **ProcesoEvolutivo**, es la última. Es la que controla todo el proceso evolutivo.

Su código básico es el siguiente:

```
import java.util.ArrayList;

public class ProcesoEvolutivo {
}
```

Esta clase tiene cuatro atributos:

- La población activa, que es una lista de individuos.
- El número de la generación activa.
- Una referencia a la clase que sirve de IHM.
- El mejor Fitness encontrado hasta el momento.
- El nombre del problema a resolver.

Por lo tanto, añadimos las siguientes definiciones:

```
   protected ArrayList<Individuo> poblacion;
   protected int numGeneracion = 0;
   protected IHM ihm = null;
   protected doble mejorFitness;
   protected String problema;
```

El primer método es el constructor. Recibe dos argumentos: la cadena que representa el problema a resolver, y la referencia hacia la IHM. En primer lugar, se inicializa el entorno de los individuos con una llamada a la fábrica, y después se crea la primera población (cuyo tamaño se define en los argumentos), también con **`FabricaIndividuos`**.

```
   public ProcesoEvolutivo(IHM _ihm, String _problema) {
       ihm = _ihm;
       problema = _problema;
       FabricaIndividuos.getInstance().Init(problema);
       población = new ArrayList();
       for (int i = 0; i < Argumentos.numIndividuos; i++) {
poblacion.add(FabricaIndividuos.getInstance().CrearIndividuo
(problema));
       }
   }
```

El siguiente método es el que gestiona la supervivencia de una generación a la siguiente. Se elige una simple sustitución: en cada generación, todos los hijos se convierten en los adultos que van desapareciendo.

Por lo tanto, su código es muy sencillo:

```
   private void Supervivencia(ArrayList<Individuo> nelleGeneracion) {
       poblacion = nuevaGeneracion;
```

```
    }
```

El siguiente método es el de selección de los individuos para convertirlos en padres. Elegimos un torneo. Para esto en primer lugar hay que seleccionar dos individuos, de manera aleatoria en la población. A continuación se comparan y se guarda el que tenga el Fitness más pequeño (es decir, el más adaptado).

Esta selección permite conservar una correcta diversidad en la población, favoreciendo los individuos adecuados.

```
   private Individuo Seleccion() {
       int indice1 =
Argumentos.random.nextInt(Argumentos.numIndividuos);
       int indice2 =
Argumentos.random.nextInt(Argumentos.numIndividuos);
       if (poblacion.get(indice1).Fitness <=
poblacion.get(indice2).Fitness) {
           return poblacion.get(indice1);
       }
       else {
           return poblacion.get(indice2);
       }
   }
```

El último método es el método `Run()`, que es el bucle principal. Este debe actuar en bucle hasta alcanzar el número máximo de generaciones o el Fitness destino fijado en los argumentos.

En cada iteración, es necesario:

- Lanzar la evaluación de cada individuo.
- Recuperar el mejor individuo y lanzar su visualización.
- Crear una nueva población (utilizando el elitismo para conservar la mejor solución hasta entonces y después la selección del o de los padres, si se aplica el crossover).
- Aplicar la supervivencia de los descendientes (que se convierten en la población en proceso).

El código utiliza dos métodos privados (uno para la evaluación y otro para la reproducción) de manera que se haga al método `Run()` más legible:

```
   // Bucle principal
   public void Run() {
       mejorFitness = Argumentos.minFitness + 1;
       while(numGeneracion < Argumentos.numMaxGeneraciones &&
mejorFitness > Argumentos.minFitness) {
           Individuo mejorInd =
EvaluarYRecuperarMejorInd(poblacion);
           mejorFitness = mejorInd.Fitness;
           ArrayList<Individuo> nuevaPoblacion =
Reproduccion(mejorInd);
           Supervivencia(nuevaPoblacion);
           numGeneracion++;
       }
   }
```

```
   // Evalúa toda la población y devuelve el mejor individuo
   private Individuo
EvaluarYRecuperarMejorInd(ArrayList<Individuo> poblacion) {
       Individuo mejorInd = poblacion.get(0);
       for(Individuo ind: poblacion) {
           ind.Evaluar();
           if (ind.Fitness < mejorInd.Fitness) {
               mejorInd = ind;
           }
       }
       ihm.MostrarMejorIndividuo(mejorInd, numGeneracion);
       return mejorInd;
   }

   // Selección y reproducción con elitismo, crossover y mutación
   private ArrayList<Individuo> Reproduccion(Individuo mejorInd) {
       ArrayList<Individuo> nuevaPoblacion = new ArrayList();
       nuevaPoblacion.add(mejorInd); // elitismo
       for (int i = 0; i < Argumentos.numIndividuos - 1; i++) {
           // Con o sin crossover ?
           if (Argumentos.random.nextDouble() <
Argumentos.tasaCrossover) {
               // Con crossover,por lo tanto dos padres
               Individuo padre1 = Seleccion();
               Individuo padre2 = Seleccion();
               nuevaPoblacion.add(FabricaIndividuos.getInstance().
               CrearIndividuo(problema, padre1, padre2));
           }
           else {
               // Sin crossover, un único padre
               Individuo padre = Seleccion();
               nuevaPoblacion.add(FabricaIndividuos.getInstance().
               CrearIndividuo(problema, padre));
           }
       }
       return nuevaPoblacion;
   }
```

Nuestro algoritmo genético genérico está ahora completo. Solo falta codificar los problemas a resolver.

2. Utilización para el viajante de comercio

a. Presentación del problema

El primer problema es un gran clásico en informática: el problema del viajante de comercio (en inglés *Travelling Salesman Problem* o TSP).

Se busca minimizar el número de kilómetros que debe recorrer un vendedor, que debe pasar una única vez por cada ciudad y volver a su punto de inicio.

Si hay cinco ciudades, se puede seleccionar cualquier ciudad de inicio, por lo tanto, tenemos cinco opciones. A

continuación quedan cuatro ciudades a visitar para la segunda opción, a continuación tres y así sucesivamente. Por lo tanto, existen 5 * 4 * 3 * 2 * 1 rutas posibles, es decir 120. Para N ciudades, hay N * (N-1) * (N-2) * ... * 1, que lo denotamos con N! (N factorial).

Por lo tanto, para seis ciudades se pasa de 120 a 720 rutas posibles. Para siete ciudades, hay 5.040 posibilidades. Con diez ciudades, se llega a casi 4 millones de posibilidades.

El viajante de comercio forma parte de los problemas llamados "NP completos": que son aquellos que tienen un número de soluciones potenciales que aumenta de manera exponencial, sin manera matemática para determinar la mejor.

Cuando el número de ciudades permanece bajo, se pueden probar todas las posibilidades. Sin embargo, de manera muy rápida llegamos al bloqueo. Por lo tanto, es interesante pasar por soluciones heurísticas que no verifican todas las rutas, sino solo las más interesantes, y este es el caso de los algoritmos genéticos.

Aquí, trabajaremos con un problema con siete ciudades francesas, con las siguientes distancias en kilómetros:

París						
462	Lyon					
772	326	Marsella				
379	598	909	Nantes			
546	842	555	338	Burdeos		
678	506	407	540	250	Toulouse	
215	664	1005	584	792	926	Lille

Para este problema, la solución óptima es 2.579 kilómetros. Hay 7*2=14 recorridos de los 5.040 posibles con esta longitud. En efecto, se puede partir de cada una de las siete ciudades para hacer el bucle más corto, y se puede hacer en el sentido que queramos. Por lo tanto, la probabilidad de encontrar "al azar" una solución óptima es del 0.28% (es decir, 1 cada 360).

b. Entorno

Por lo tanto empezamos el código por un conjunto de clases que representa el entorno del individuo. Esas clases permiten definir los genes y la evaluación del individuo.

Empezamos definiendo la estructura **PVC** que representa el problema del viajante de comercio. Esta es estática y permite manipular las ciudades. Por lo tanto, contiene una lista de ciudades (`ciudades`) y una tabla de doble entrada que indica las distancias que separan estas ciudades (`distancias`).

```
import java.util.ArrayList;
import java.util.Arrays;

// Problema del Viajante de Comercio
public class PVC {
   static ArrayList<String> ciudades;
   static int[][] distancias;
```

```
    // Métodos aquí
}
```

El primer método es un método de inicialización, equivalente a un constructor. Inicializa la lista de las ciudades y la rellena con las siete ciudades citadas anteriormente. A continuación, crea la tabla de doble entrada e introduce los diferentes kilometrajes que separan las ciudades, en orden.

```
    public static void Init() {
        ciudades = new ArrayList(Arrays.asList("París", "Lyon", "Marsella",
"Nantes", "Burdeos", "Toulouse", "Lille"));

        distancias = new int[ciudades.size()][];
        distancias[0] = new int[] {0, 462, 772, 379, 546, 678, 215}; // París
        distancias[1] = new int[] { 462, 0, 326, 598, 842, 506, 664}; // Lyon
        distancias[2] = new int[] { 772, 326, 0, 909, 555, 407, 1005};
// Marsella
        distancias[3] = new int[] { 379, 598, 909, 0, 338, 540, 584}; // Nantes
        distancias[4] = new int[] { 546, 842, 555, 338, 0, 250, 792}; // Burdeos
        distancias[5] = new int[] { 678, 506, 407, 540, 250, 0, 926}; // Toulouse
        distancias[6] = new int[] { 215, 664, 1005, 584, 792, 926, 0}; // Lille
    }
```

A continuación se codifica un método `getDistancia` que permite saber cuántos kilómetros separan a dos ciudades que se pasan como argumentos (con su índice). Por lo tanto, es suficiente con leer la tabla de distancias.

```
    protected static int getDistancia(int ciudad1, int ciudad2) {
        return distancias[ciudad1][ciudad2];
    }
```

Para la visualización, se añade un método que devuelve el nombre de una ciudad a partir de su índice:

```
    protected static String getCiudad(int ciudadIndice) {
        return ciudades.get(ciudadIndice);
    }
```

Para terminar, se crea un último método: devuelve una copia de las ciudades existentes y se llama `getCiudadIndice`. Para esto, se crea una nueva lista de índices, a la que se añaden todos los índices.

```
    protected static ArrayList<Integer> getCiudadIndice() {
        int numCiudades = ciudades.size();
        ArrayList<Integer> ciudadesIndex = new ArrayList();
        for (int i = 0; i < numCiudades; i++) {
            ciudadesIndex.add(i);
        }
        return ciudadesIndex;
    }
```

El entorno está ahora completo y es posible codificar a los individuos.

c. Genes

Los individuos se componen de una sucesión de genes, cada uno de ellos representa una ciudad a visitar. Por lo tanto, empezamos definiendo los genes con una clase **PVCGen** que implementa la interfaz **IGen**.

Esta clase solo contiene un atributo, el índice de la ciudad que se corresponde con el gen:

```
class PVCGen implements IGen {
   protected int ciudadIndice;

   // Métodos aquí
}
```

Los dos primeros métodos son los constructores. En los dos casos, hay que indicar la ciudad correspondiente, es decir directamente como un entero (durante la inicialización), es decir, gracias a otro gen que será necesario copiar (para la reproducción).

Por lo tanto, los constructores son los siguientes:

```
   public PVCGen(int _ciudadIndice) {
       ciudadIndice = _ciudadIndice;
   }
   public PVCGen(PVCGen g) {
       ciudadIndice = g.ciudadIndice;
   }
```

Se añade un método `getDistancia`, que devuelve la distancia entre la ciudad de este gen y el contenido en otro gen. Para esto, se llama al método creado anteriormente en el entorno **PVC**.

```
   protected int getDistancia(PVCGen g) {
       return PVC.getDistancia(ciudadIndice, g.ciudadIndice);
   }
```

La interfaz **IGen** define un método `Mutar()`. En el caso del problema del viajante de comercio, la mutación de un gen único no tiene sentido: solo se puede cambiar el orden de los genes pero no su contenido, dando por hecho que obligatoriamente se debe pasar una y solo una vez por cada ciudad. Por lo tanto, este método devuelve una excepción si se llama.

```
   @Override
   public void Mutar() {
       throw new UnsupportedOperationException("Not supported yet.");
   }
```

Esta clase termina por un método `toString`, de manera que se puedan visualizar nuestros genes (y por lo tanto, nuestro mejor individuo):

```
   public String toString() {
       return PVC.getCiudad(ciudadIndice);
   }
```

Los genes se codifican ahora. Esta codificación es muy específica al problema a resolver, pero sin embargo permanece muy rápido.

d. Individuos

Por lo tanto, ahora podemos codificar nuestros individuos **PVCIndividuo**, que heredan de la clase abstracta **Individuo**. El genoma y el fitness ya estaban definidos y no tenemos atributos adicionales.

El primer método es el constructor por defecto. Cuando esto se llama durante la inicialización, demanda la lista de ciudades a recorrer y después elige una de manera aleatoria, transformándola en un gen y así de manera consecutiva, hasta que todas las ciudades se hayan visitado.

```
import java.util.ArrayList;

public class PVCIndividuo extends Individuo {

   public PVCIndividuo() {
       genoma = new ArrayList();
       ArrayList<Integer> indiceDispo = PVC.getCiudadIndice();
       while (!indiceDispo.isEmpty()) {
           int indices =
Argumentos.random.nextInt(indiceDispo.size());
           genoma.add(new PVCGen(indiceDispo.get(indice)));
           indiceDispo.remove(indice);
       }
   }}
```

El operador de mutación consiste en cambiar la posición de un gen: se elimina un gen de manera aleatoria y se sustituye en un índice elegido al azar. Esta mutación solo se hace si se selecciona un número inferior a la tasa de mutación.

```
   @Override
   public void Mutar() {
       if (Argumentos.random.nextDouble() <
Argumentos.tasaMutacion) {
           int indice1 = Argumentos.random.nextInt(genoma.size());
           PVCGen g = (PVCGen)genoma.get(indice1);
           genoma.remove(g);
           int indice2 = Argumentos.random.nextInt(genoma.size());
           genoma.add(indice2, g);
       }
   }
```

Ahora se pueden codificar los dos últimos constructores. El primero es un constructor utilizado cuando solo se tiene un único padre. En este caso, se reconstruye un genoma haciendo una copia de los genes uno a uno, y

después se llama a nuestro operador de mutación.

```
public PVCIndividuo(PVCIndividuo padre) {
    genoma = new ArrayList();
    for (IGen g: padre.genoma) {
        this.genoma.add(new PVCGen((PVCGen)g));
    }
    Mutar();
}
```

El segundo se llama en el caso de un crossover. Eligiendo un punto de corte de manera aleatoria y se copian las ciudades que tienen este punto desde el primer padre. A continuación recorremos el segundo padre para recuperar solo las ciudades todavía no visitadas, conservando su orden. Para terminar, se llama al operador de mutación.

```
public PVCIndividuo(PVCIndividuo padre1, PVCIndividuo padre2) {
    genoma = new ArrayList();
    int ptCorte = Argumentos.random.nextInt(padre1.genoma.size());
    for(int i = 0; i < ptCorte; i++) {
        genoma.add(new PVCGen((PVCGen) padre1.genoma.get(i)));
    }
    for (IGen g: padre2.genoma) {
        if (!genoma.contains((PVCGen)g)) {
            genoma.add(new PVCGen((PVCGen)g));
        }
    }
    Mutar();
}
```

El último método de esta clase es la evaluación de un individuo. Para esto, se debe recorrer la lista de las ciudades y pedir la distancia entre las ciudades dos a dos. Para terminar, no podemos olvidarnos de añadir la distancia de la última ciudad a la primera para volver de nuevo a nuestro recorrido.

```
@Override
public doble Evaluar() {
    int kmTotal = 0;
    PVCGen antiguoGen = null;
    for (IGen g: genoma) {
        if (antiguoGen != null) {
            kmTotal += ((PVCGen)g).getDistancia(antiguoGen);
        }
        antiguoGen = (PVCGen)g;
    }
    kmTotal += antiguoGen.getDistancia((PVCGen)genoma.get(0));
    Fitness = kmTotal;
    return Fitness;
}
```

La clase **PVCIndividuo** está terminada. Sin embargo, necesitamos modificar la fábrica de individuos **FabricaIndividuos** para que pueda llamar a constructores correctos y a la correcta inicialización en función de

las necesidades. Por lo tanto, es necesario crear cada vez un switch y si el problema vale "PVC", entonces llamaremos a nuestros diferentes métodos.

Por lo tanto, el método `Init` se convierte en:

```
   void Init(String tipo) {
       switch (tipo) {
           casilla "PVC":
               PVC.Init();
               break;
       }
   }
```

A continuación se modifica el método que permite inicializar un individuo:

```
   public Individuo CrearIndividuo(String tipo) {
       Individuo ind = null;
       switch (tipo) {
           casilla "PVC":
               ind = new PVCIndividuo();
               break;
       }
       return ind;
   }
```

Se hace lo mismo con el constructor a partir de un padre, y después este a partir de dos padres, que llaman a los constructores correctos de nuestra clase **`PVCIndividuo`**.

```
   public Individuo CrearIndividuo(String tipo, Individuo padre) {
       Individuo ind = null;
       switch (tipo) {
           casilla "PVC":
               ind = new PVCIndividuo((PVCIndividuo)padre);
               break;
       }
       return ind;
   }

   public Individuo CrearIndividuo(String tipo, Individuo padre1,
Individuo padre2) {
       Individuo ind = null;
       switch (tipo) {
           casilla "PVC":
               ind = new PVCIndividuo((PVCIndividuo)padre1,
(PVCIndividuo)padre2);
               break;
       }
       return ind;
   }
```

Nuestro programa es ahora completamente funcional.

e. Programa principal

Terminamos por el programa principal. Este contiene el método `main` e implementa la interfaz **`IHM`** para poder obtener y visualizar el mejor individuo.

Por lo tanto, el esqueleto de nuestra clase **`Aplicacion`** es el siguiente:

```
public class Aplicacion implements IHM {
   public static void main(String[] args) {
       Aplicacion app = new Application();
       app.Run();
   }

   public void Run()
   {
         // Código principal ICI
   }

   @Override
   public void MostrarMejorIndividuo(Individuo ind, int generacion) {
       System.out.println(generacion + " -> " + ind.toString());
   }
}
```

El código principal del programa que se debe introducir en el método `Run` es sencillo. En efecto, se comienza introduciendo los argumentos deseados. No queremos poder agregar o eliminar genes y el crossover no aporta nada sobre un problema tan pequeño, por lo que las tasas correspondientes son 0. También se fija una tasa de mutación de 0.3, lo que indica que el 30% de los individuos sufren un intercambio en su genoma. Para terminar, se fija el Fitness que se debe alcanzar al valor óptimo de 2.579.

Una vez que se han indicado los argumentos, se crea un nuevo proceso evolutivo, indicándole que el problema a resolver es de tipo "PVC". Terminamos lanzando el algoritmo.

```
   public void Run() {
       // Resolución del viajante de comercio
       // Argumentos
       Argumentos.tasaCrossover = 0.0;
       Argumentos.tasaMutacion = 0.3;
       Argumentos.tasaAdicionGenes = 0.0;
       Argumentos.tasaElimGen = 0.0;
       Argumentos.minFitness = 2579;
       // Ejecución
       ProcesoEvolutivo sist =
new ProcesoEvolutivo(this, "PVC");
       syst.Run();
  }
```

f. Resultados

Con un algoritmo genético, nada permite garantizar que todas las simulaciones encuentran la óptima, a menos que se deje que el algoritmo continúe indefinidamente. Aquí, hemos limitado el algoritmo a 500 generaciones.

Durante las pruebas realizadas con los argumentos indicados en 1.000 lanzamientos, una única simulación no encuentra la óptima. Un desarrollo clásico del algoritmo es el siguiente:

```
0 -> (3336.0) - Nantes - Lille - París - Burdeos - Toulouse - Lyon -
Marsella
1 -> (3156.0) - Marsella - Toulouse - Burdeos - París - Nantes -
Lille - Lyon
2 -> (2582.0) - Marsella - Toulouse - Burdeos - Nantes - Lille -
París - Lyon
3 -> (2582.0) - Marsella - Toulouse - Burdeos - Nantes - Lille -
París - Lyon
4 -> (2582.0) - Marsella - Toulouse - Burdeos - Nantes - Lille -
París - Lyon
5 -> (2582.0) - Marsella - Toulouse - Burdeos - Nantes - Lille -
París - Lyon
6 -> (2582.0) - Marsella - Toulouse - Burdeos - Nantes - Lille -
París - Lyon
7 -> (2582.0) - Marsella - Toulouse - Burdeos - Nantes - Lille -
París - Lyon
8 -> (2582.0) - Marsella - Toulouse - Burdeos - Nantes - Lille -
París - Lyon
9 -> (2582.0) - Marsella - Toulouse - Burdeos - Nantes - Lille -
París - Lyon
10 -> (2579.0) - Marsella - Toulouse - Burdeos - Nantes - París -
Lille - Lyon
```

Con la media de los 1.000 tests, el algoritmo converge en 11,4 generaciones. Por lo tanto, ha habido 11,4*20 individuos como máximo (porque puede haber repeticiones dentro de la población) y por lo tanto, 228 soluciones potenciales probadas. Cuando se comparan las 5.040 soluciones posibles (entre las que se encuentran 14 recorridos óptimos), vemos que el algoritmo permite no tener que probar todas las posibilidades, siendo dirigidos con la evolución al objetivo que se pretende alcanzar.

Recuerde que la probabilidad de encontrar "por azar " la respuesta correcta era de 1 sobre 360 y por lo tanto, vemos que el algoritmo genético ha permitido ir más rápido que el puro azar.

3. Utilización para la resolución de un laberinto

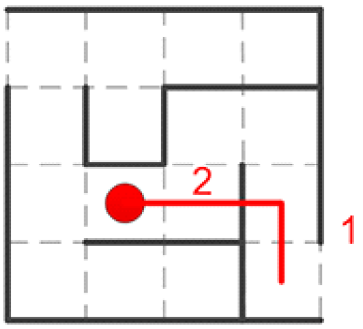
a. Presentación del problema

Queremos encontrar la salida de un laberinto. Para esto, cada individuo está formado por la sucesión de acciones que hay que tomar para ir desde la entrada hasta la salida.

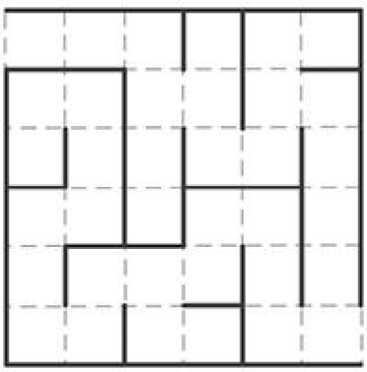
Nos detenemos en el momento de encontrar una solución que funcione, aunque no sea la óptima en términos de movimientos (por lo tanto, podemos hacer idas y vueltas).

El Fitness de un individuo es la distancia de Manhattan entre su última posición y la salida. Esta distancia es muy sencilla: se trata simplemente del número de casillas horizontales añadidas al número de casillas verticales, entre un individuo y la casilla de salida.

En el siguiente ejemplo, el individuo (círculo) está a una distancia de 3 de la salida: dos casillas horizontales y una casilla vertical.



Además de este primer laberinto, se propone un segundo que se muestra a continuación:



Hay algoritmos eficaces para salir de un laberinto. Por lo tanto, la utilización de un algoritmo genético en una aplicación real para resolver este problema, no sería siempre la mejor opción.

La dificultad aquí reside en el número de rutas posibles y el hecho de que no se conoce con antelación el tamaño de las rutas. Por lo tanto, se deben gestionar genomas de tamaños variables (cada gen se corresponde con un orden de desplazamiento).

Los órdenes de desplazamiento son absolutos y se siguen aplicando hasta que este ya no sea posible (muro) o encontremos un cruce.

b. Entorno

Como sucedía para el problema del viajante de comercio, vamos a empezar creando el entorno de nuestros individuos. Aquí, necesitamos principalmente generar los laberintos y calcular las distancias.

Empezamos definiendo una estructura **`Casilla`** que se corresponda con una casilla de nuestro laberinto y que contenga sus coordenadas, como un constructor. También se redefine el método `equals` para comprobar la igualdad entre dos casillas.

```
public class Casilla {
   public int i;
   public int j;
```

```
    public Casilla(int _i, int _j) {
        i = _i;
        j = _j;
    }

    @Override
    public boolean equals(Object o) {
        return (i == ((Casilla)o).i && j == ((Casilla)o).j);
    }
}
```

Se define una clase estática **`Laberinto`**. Esta contiene todos las "puertas" del laberinto, es decir, todas las rutas desde una casilla a otra. Cada una es una pareja de casillas que se representa por una tabla de `Casilla`.

Por lo tanto, la clase contiene tres atributos: la lista de las rutas llamada `rutas`, la casilla de entrada y la casilla de salida.

```
import java.util.ArrayList;

public class Laberinto {
    private static ArrayList<Casilla[]> rutas;
    private static Casilla entrada;
    private static Casilla salida;

    // Otros atributos

    // Métodos
}
```

También es necesario definir dos cadenas que son nuestros laberintos en "ASCII art". Esto permite introducirlos más fácilmente que creando la lista de las rutas a mano.

```
    public static String Lab1 = "*--*--*--*--*\n" +
                                "E           |\n" +
                                "*  *  *--*--*\n" +
                                "|  |  |     |\n" +
                                "*  *--*  *  *\n" +
                                "|        |  |\n" +
                                "*  *--*--*  *\n" +
                                "|        |  S\n" +
                                "*--*--*--*--*";
    public static String Lab2 = "*--*--*--*--*--*--*\n" +
                                "E        |  |     |\n" +
                                "*--*--*  *  *  *--*\n" +
                                "|     |     |     |\n" +
                                "*  *  *  *  *  *  *\n" +
                                "|  |  |  |     |  |\n" +
                                "*--*  *  *--*--*  *\n" +
                                "|     |  |     |  |\n" +
                                "*  *--*--*  *  *  *\n" +
                                "|  |        |  |  |\n" +
                                "*  *  *  *--*  *  *\n" +
```

```
                            "|     |     |     S\n" +
                            "*--*--*--*--*--*--*";
```

Para terminar, las direcciones se definen por una enumeración que puede tomar cuatro valores:

```
public enum Direccion { Arriba, Abajo, Izquierda, Derecha};
```

El primer método inicializa los atributos (rutas y entrada/salida) en función de una cadena que se pasa como argumento (el diseño del laberinto). Para esto, en primer lugar la cadena se descompone respecto al carácter '\n' (retorno de carro).

Las líneas pares (la primera línea numerada como 0), se corresponde con los muros. Los caracteres "*" que representan las esquinas de las casillas, una sucesión de "**--** " entre dos asteriscos representa un muro, mientras que los espacios indican la ausencia de muro. En este último caso, se añade una ruta vertical, desde la casilla superior hasta la que se encuentra más abajo.

Las líneas impares se corresponden con los pasillos. Por lo tanto, es necesario recorrer los caracteres tres a tres, la presencia de un muro se simboliza con "**|**". Si no existe, se puede crear una ruta horizontal entre la casilla y la anterior.

También se va a verificar la presencia de la entrada y de la salida cuando estemos en una línea impar (se corresponden con los pasillos). En este caso, tenderemos una 'E' o una 'S' en la línea.

Por lo tanto, el código es el siguiente:

```
public static void Init(String s) {
    rutas = new ArrayList();

    String[] lineas = s.split("\n");
    int numLineas = 0;
    for (String linea: lineas) {
        if (numLineas% 2 != 0) {
            // Número impar,por lo tanto línea de pasillo
            int indices = linea.indiceOf("E");
            if (indice != -1) {
                // Tenemos una entrada en este pasillo
                if (indice == linea.length() - 1) {
                    indices --;
                }
                entrada = new Casilla(numLineas/2, indices/3);
            }
            indices = linea.indiceOf("S");
            if (indice != -1) {
                // Tenemos una salida en el pasillo
                if (indice == linea.length() - 1) {
                    indices --;
                }
                salida = new Casilla(numLineas/2, indices/3);
            }
            // Recorremos el pasillo para crear las rutas
            // horizontales
```

```
                for (int columna = 0; columna < linea.length() / 3;
columna++) {
                    String casillaStr = linea.substring(columna*3,
columna*3 + 3);
                    if (!casillaStr.contains("|") &&
!casillaStr.contains("E") && !casillaStr.contains("S")) {
                        // Ruta abierta, se añade
                        rutas.add(new Casilla[]{new Casilla(numLineas/2,
columna-1), new Casilla(numLineas/2, columna)});
                    }
                }
            }
            else {
                // Línea par: estamos en los muros
                String [] casillas = linea.substring(1).split("\\*");
                int columna = 0;
                for (String bloque: casillas) {
                    if (bloque.equals("  ")) {
                        // Ruta abierta, se añade
                        rutas.add(new Casilla[] {new Casilla(numLineas/2 -
1, columna), new Casilla(numLineas/2, columna)});
                    }
                    columna++;
                }
            }
            numLineas++;
        }
    }
```

El siguiente método permite determinar si es posible ir desde una casilla hasta otra. Para esto, se buscan las rutas si existe una que vaya desde la casilla 1 hasta la casilla 2 o desde la casilla 2 hasta la casilla 1. En efecto, las rutas solo se guardan una vez, aunque se toman en los dos sentidos.

```
    private static boolean esPosible(Casilla pos1, Casilla pos2) {
        for (Casilla[] camino: rutas) {
            if ((camino[0].equals(pos1) && camino[1].equals(pos2)) ||
((camino[0].equals(pos2) && camino[1].equals(pos1)))) {
                return true;
            }
        }
        return false;
    }
```

También se escribe un método que permite saber si una casilla es un cruce. Para esto, simplemente se cuenta el número de rutas que llegan hasta la casilla. Si hay tres o más, entonces es un cruce. En caso contrario, se trata simplemente de un pasillo (dos rutas) o de un callejón sin salida (un camino).

```
    private static boolean esCruce(Casilla pos) {
        int numCaminos = 0;
        for (Casilla[] camino: rutas) {
            if (camino[0].equals(pos) || camino[1].equals(pos)) {
                numCaminos++;
```

```
            }
        }
        return numCaminos > 2;
    }
```

El entorno está listo y podemos pasar a la implementación de los individuos. Más adelante volveremos a esta clase para evaluar los individuos en los desplazamientos por el laberinto.

c. Genes

Empezamos creando los genes con una clase llamada **LabGen** y que implementa la interfaz **IGen**. Un gen contiene únicamente una dirección a seguir. Se añaden dos constructores: uno para crear un gen de manera aleatoria y otro para copiar un gen dado como argumento.

```
public class LabGen implements IGen {
    public Laberinto.Direccion direccion;

    public LabGen() {
        direccion =
Laberinto.Direccion.values()[Argumentos.random.nextInt(4)];
    }

    public LabGen(LabGen g) {
        direccion = g.direccion;
    }
}
```

Añadimos un método `toString`, que solo muestre la primera letra de la dirección para simplificar las visualizaciones (A para Arriba, B para aBajo, I para Izquierda o D para Derecha):

```
    @Override
    public String toString() {
        return direccion.name().substring(0, 1);
    }
```

Para terminar, hay que definir un método `Mutar()` para respetar la interfaz. Es suficiente con volver a hacer una selección al azar para una nueva dirección. Como hay cuatro direcciones posibles, en un 25% de los casos nos encontraremos con la dirección que teníamos antes de la mutación.

```
    @Override
    public void Mutar() {
        direccion =
Laberinto.Direccion.values()[Argumentos.random.nextInt(4)];
    }
```

d. Individuos

Ahora vamos a codificar los individuos en la clase **LabIndividuo**, que hereda de **Individuo**, la clase abstracta

definida anteriormente.

Esta clase no contiene atributo, el Fitness y el genoma ya están definidos en la clase padre.

Empezamos creando un primer constructor que no recibe argumentos, y que por lo tanto permite crear individuos de manera aleatoria. Estos tendrán tantos genes como se definen en la clase configurada.

Por lo tanto, el código básico de la clase es el siguiente:

```
import java.util.ArrayList;

public class LabIndividuo extends Individuo {
   public LabIndividuo() {
       genoma = new ArrayList();
       for (int i = 0; i < Argumentos.numGenes; i++) {
           genoma.add(new LabGen());
       }
   }
}
```

A continuación, necesitamos un método que permita mutar a nuestros individuos. Esta mutación puede tener tres formas diferentes:

- La eliminación de un gen, con una tasa definida por `tasaElimGen`. El gen se elige de manera aleatoria.
- La adición de un gen, con una tasa definida por `tasaAdicionGenes`. Se añade el nuevo a la sucesión del recorrido ya creado y la dirección se elige de manera aleatoria.
- La modificación de los genes con una tasa por gen de `tasaMutacion`. Por lo tanto, se recorre todo el genoma y se prueba si se deben cambiar las direcciones una a una.

El código es el siguiente:

```
   @Override
   public void Mutar() {
       // ¿Eliminación de un gen?
       if (Argumentos.random.nextDouble() <
Argumentos.tasaElimGen) {
           int indices = Argumentos.random.nextInt(genoma.size());
           genoma.remove(indice);
       }
       // ¿Añadir un gen al final?
       if (Argumentos.random.nextDouble() <
Argumentos.tasaAdicionGenes) {
           genoma.add(new LabGen());
       }
       // ¿Cambio de valores?
       for(IGen g: genoma) {
           if (Argumentos.random.nextDouble() <
Argumentos.tasaMutacion) {
               g.Mutar();
           }
       }
```

```
    }
```

A continuación se crea un segundo constructor que solo toma un padre. En este caso, se copian los genes uno a uno, y después se llama al método de mutación:

```
    public LabIndividuo(LabIndividuo padre) {
        genoma = new ArrayList();
        for (IGen g: padre.genoma) {
            genoma.add(new LabGen((LabGen) g));
        }
        Mutar();
    }
```

El último constructor recibe dos padres como argumentos. En primer lugar se elige un punto de crossover de manera aleatoria. Todos los genes antes de este punto, se copian desde el primer padre y después, es el turno de los genes situados después de este punto en el segundo padre (si quedan genes). Para esto, se utiliza ventajosamente el método `subList`, que permite tomar solo una parte de la colección.

```
    public LabIndividuo(LabIndividuo padre1, LabIndividuo padre2) {
        genoma = new ArrayList();
        // Crossover
        int indices = Argumentos.random.nextInt(padre1.genoma.size());
        for (IGen g: padre1.genoma.subList(0, indice)) {
            genoma.add(new LabGen((LabGen) g));
        }
        if (indice < padre2.genoma.size()) {
            for (IGen g: padre2.genoma.subList(indice,
padre2.genoma.size())) {
                genoma.add(new LabGen((LabGen) g));
            }
        }
        // Mutación
        Mutar();
    }
```

El último método de esta clase es la evaluación, que implica mover al individuo en el laberinto. Vamos a llevar esta función de la clase `Laberinto`. Por lo tanto, empezamos llamando al método de evaluación de la clase `Laberinto` de la clase **Individuo**.

```
    @Override
    public doble Evaluar() {
        Fitness = Laberinto.Evaluar(genoma);
        return Fitness;
    }
```

Ahora se completa la clase **Laberinto** con el método `Evaluar`. Para empezar, partimos de la entrada, que es la casilla de inicio. A continuación se aplican los genes uno a uno y se cambia en cada desplazamiento la casilla sobre la que estamos. La dirección solicitada se guarda hasta que ya no sea posible avanzar o lleguemos a un cruce. En este momento, se pasa al gen siguiente.

Se para cuando se llegue a la casilla de llegada o cuando no queden más genes que aplicar. Al final, se calcula la distancia de Manhattan a la salida, que se devuelve.

Para simplificar el código, se va a implementar el desplazamiento a una dirección dada, en un método `Mover`.

A continuación se muestra el código de estos dos métodos:

```
   static void Mover(Casilla inicio, int movI, int movJ) {
       boolean finMovimiento = false;
       while(esPosible(inicio, new Casilla(inicio.i + movI,
inicio.j + movJ)) && !finMovimiento) {
           inicio.i += movI;
           inicio.j += movJ;
           finMovimiento = esCruce(inicio) ||
inicio.equals(salida);
       }
       finMovimiento = false;
   }

   static doble Evaluar(ArrayList<IGen> genoma) {
       Casilla casillaActual = new Casilla(entrada.i, entrada.j);
       boolean finMovimiento = false;
       for(IGen g: genoma) {
           switch (((LabGen)g).direccion) {
               casilla Abajo:
                   Mover(casillaActual, 1, 0);
                   break;
               casilla Arriba:
                   Mover(casillaActual, -1, 0);
                   break;
               casilla Derecha:
                   Mover(casillaActual, 0, 1);
                   break;
               casilla Izquierda:
                   Mover(casillaActual, 0, -1);
                   break;
           }
           if (casillaActual.equals(salida)) {
               break;
           }
       }
       // Cálculo del Fitness: distancia de Manhattan
       int distancia = Math.abs(salida.i - casillaActual.i) +
Math.abs(salida.j - casillaActual.j);
       return distancia;
   }
```

e. Modificación de la fábrica

Una vez que se han codificado los individuos, hay que modificar la fábrica de individuos **`FabricaIndividuos`**, para agregar a cada método un nuevo caso, si el problema dado se llama "Lab". Las líneas añadidas están en negrita en el siguiente código:

```
   void Init(String tipo) {
       switch (tipo) {
           casilla "PVC":
               PVC.Init();
               break;
           casilla "Lab":
               Laberinto.Init(Laberinto.Lab2);
               break;
       }
   }

   public Individuo CrearIndividuo(String tipo) {
       Individuo ind = null;
       switch (tipo) {
           casilla "PVC":
               ind = new PVCIndividuo();
               break;
           casilla "Lab":
               ind = new LabIndividuo();
               break;
       }
       return ind;
   }

   public Individuo CrearIndividuo(String tipo, Individuo padre) {
       Individuo ind = null;
       switch (tipo) {
           casilla "PVC":
               ind = new PVCIndividuo((PVCIndividuo)padre);
               break;
           casilla "Lab":
               ind = new LabIndividuo((LabIndividuo)padre);
               break;
       }
       return ind;
   }

   public Individuo CrearIndividuo(String tipo, Individuo padre1,
Individuo padre2) {
       Individuo ind = null;
       switch (tipo) {
           casilla "PVC":
               ind = new PVCIndividuo((PVCIndividuo)padre1,
(PVCIndividuo)padre2);
               break;
           casilla "Lab":
               ind = new LabIndividuo((LabIndividuo)padre1,
(LabIndividuo)padre2);
               break;
       }
       return ind;
   }
```

Observe que el laberinto elegido aquí es el segundo, más complejo que el primero. Es suficiente con cambiar el laberinto de la función `Init` para cambiar el problema.

Se podría agregar un argumento al método `Init` para determinar la cadena que se debe utilizar o crear un método específico, pero conservamos las opciones "en duro" para simplificar el código y centrarnos en el algoritmo genético.

f. Programa principal

Ahora terminamos por el programa principal **`Aplicacion`**. Retoma la misma estructura que la del problema del viajante de comercio. La única diferencia se sitúa en el método `Run`.

De esta manera encontramos los argumentos más adaptados a este problema:

- Una tasa de crossover del 60%, es decir 0.6.
- Mutaciones de tasa de 0.1 (un gen de cada 10 de media), la adición de un gen en el 80% de los casos (0.8) y la eliminación en el 10% (de esta manera, hay tendencia a alargar las rutas en lugar de acortarlas).
- El Fitness mínimo previsto es nulo, es decir que se llega a la casilla de salida.

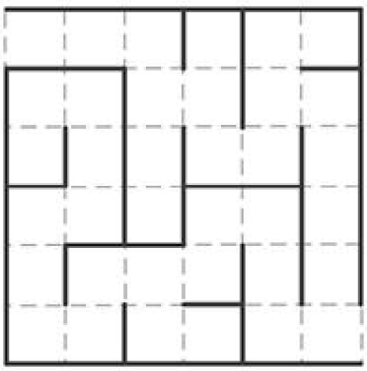
A continuación el programa se lanza con el método `Run` del proceso.

```
public class Aplicacion implements IHM {
   public static void main(String[] args) {
       Aplicacion app = new Application();
       app.Run();
   }

   public void Run() {
       // Resolución del laberinto
       // Argumentos
       Argumentos.tasaCrossover = 0.6;
       Argumentos.tasaMutacion = 0.1;
       Argumentos.tasaAdicionGenes = 0.8;
       Argumentos.tasaElimGen = 0.1;
       Argumentos.minFitness = 0;
       Argumentos.numMaxGeneraciones = 300;
       // Ejecución
       ProcesoEvolutivo sist =
new ProcesoEvolutivo(this, "Lab");
       syst.Run();
   }

   @Override
   public void MostrarMejorIndividuo(Individuo ind, int
generacion) {
       System.out.println(generacion + " -> " + ind.toString());
   }
}
```

Nuestro programa ha terminado.

g. Resultados

En primer lugar, recordemos el diseño del laberinto utilizado:

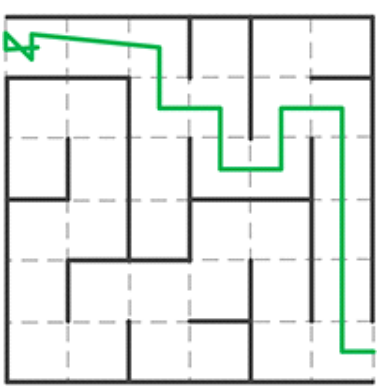


A continuación se muestra un caso típico de solución obtenida:

```
0 -> (5.0) - D - B - I - I - B - B - B - I - B - D
1 -> (5.0) - D - B - I - I - B - B - B - I - B - D
2 -> (4.0) - B - D - D - D - A - B - D - B - B - I - D
3 -> (4.0) - B - D - D - D - A - B - D - B - B - I - D
4 -> (4.0) - B - D - D - D - A - B - D - B - B - I - D
5 -> (4.0) - B - D - D - D - A - B - D - B - B - I - D
6 -> (4.0) - B - D - D - D - A - B - D - B - B - I - D
7 -> (4.0) - B - D - D - D - A - B - D - B - B - I - D
8 -> (4.0) - B - D - D - D - A - B - D - B - B - I - D
9 -> (4.0) - B - D - D - D - A - B - D - B - B - I - D
10 -> (0.0) - I - I - A - B - B - A - D - B - D - B - D - A - D - B
```

Se puede ver que en la primera generación, el algoritmo se detiene a 5 casillas de la salida. A continuación encuentran soluciones deteniéndose a 4 casillas de la salida y después la salida, gracias a la adición de genes. En efecto, es muy difícil resolver el segundo laberinto con 10 genes, sobre todo porque el algoritmo hace idas y vueltas algunas veces.

Por lo tanto, se obtiene el siguiente camino y se puede observar que tropieza contra los muros muchas veces (indicadas por los pequeños trazos en el recorrido):



Con 20 individuos y los argumentos elegidos, no hay nada que garantice encontrar la solución óptima (y por lo tanto, la salida). En efecto, puede suceder que el algoritmo quede bloqueado en soluciones óptimas locales. Sin embargo, sobre 10.000 pruebas realizadas, la convergencia se ha producido como media en 11 generaciones (10,92), y como máximo al cabo de 42 generaciones.

Este es el motivo por el que cuando se utiliza un algoritmo genético, tenemos tendencia a ejecutarlo sobre numerosas generaciones y con muchos individuos, incluso a volver a lanzar el algoritmo numerosas veces, para determinar varias respuestas: nada puede garantizar la convergencia hacia la solución en un tiempo finito.

Resumen

Los algoritmos genéticos (o más normalmente los algoritmos evolutivos), están inspirados en las diferentes búsquedas realizadas en biología de la evolución.

Entonces tenemos una población de individuos cada una compuesta por un genoma, que es una lista de genes. Estos individuos se evalúan respecto a la calidad de la solución de un problema dado, que representan (es lo que se llama su fenotipo).

Los mejores individuos se seleccionan para ser reproductores. Se crean nuevas soluciones a partir de uno o varios padres. En caso de que intervengan varios padres se realiza un crossover, es decir, un cruce entre la información genética de los diferentes padres.

A continuación los genomas de los descendientes sufren mutaciones aleatorias, que representan los errores de copiado que tiene lugar durante la reproducción. Cada descendiente, aunque sean parecidos a sus padres, es potencialmente diferente.

A continuación, esta nueva generación debe sobrevivir para formar parte de la población de la generación siguiente. Volvemos a ejecutar el proceso hasta alcanzar las soluciones óptimas o casi óptimas.

De esta manera, los algoritmos genéticos permiten resolver muchos problemas para los que no existe solución matemática conocida y cuyo espacio de búsqueda es demasiado vasto para una búsqueda exhaustiva.

Presentación del capítulo

Este capítulo presenta diversas técnicas (o metaheurísticos) de búsqueda de mínimos locales. Por ejemplo, podemos querer minimizar el coste de producción, o la cantidad de materia prima necesaria para elaborar una pieza, respetando numerosas restricciones. Estos problemas resultan muy comunes en la vida cotidiana, y sin embargo son difíciles de resolver por un ordenador (y todavía más por un humano), puesto que el número de soluciones posibles es muy importante.

La primera sección de este capítulo presenta con más detalle este problema y las restricciones asociadas, así como algunos ejemplos.

Las siguientes secciones presentan los principales algoritmos: algoritmo voraz, descenso por gradiente, búsqueda tabú, recocido simulado y optimización por enjambre de partículas.

A continuación, se presentan los principales dominios de aplicación de estas técnicas.

Los distintos algoritmos se implementan en la última sección, en Java. El código correspondiente está disponible para su descarga.

Por último, un pequeño resumen cierra este capítulo.

Optimización y mínimos

Los problemas de optimización y de búsqueda de mínimos son comunes, y su resolución exacta es complicada o incluso imposible. La inteligencia artificial ha desarrollado algoritmos específicos para estos problemas.

1. Ejemplos

Los ingenieros tienen que resolver numerosos problemas de **optimización**, como minimizar el coste de un objeto conservando ciertas propiedades, o bien optimizar la fórmula de un metal para hacerlo más resistente.

En la vida cotidiana, existen también problemas de este tipo. Pagar utilizando la menor cantidad de monedas posible (o, por el contrario, tratar de deshacerse de la mayor cantidad de calderilla posible) es un ejemplo clásico. Para aquellos que tengan tickets restaurante, pedir en un restaurante o comprar en una tienda lo suficiente como para cubrir el precio del ticket (puesto que no se devuelve la cantidad sobrante) pero sin superar el importe total es otro.

Cargar un coche, organizar un almacén, modificar una composición, determinar un dibujo, crear un circuito impreso, limitar los costes de embalaje... son otros ejemplos de problemas de optimización.

2. El problema de la mochila

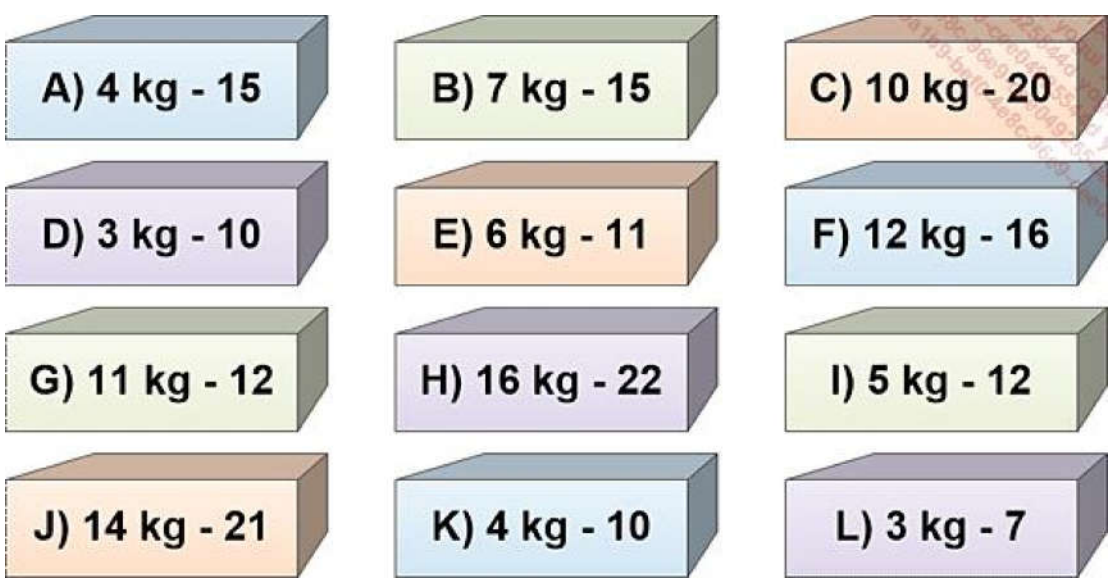
El **problema de la mochila** (o *Knapsack Problem* en inglés, abreviado KP) es sencillo de entender, pero muy difícil de resolver.

Una mochila tiene una capacidad máxima (si no, podría romperse). Tenemos varios objetos disponibles, cada uno con un peso y un valor. El objetivo consiste en maximizar el valor de los objetos cargados.

Evidentemente, es imposible cargar el total de los objetos (debido a que pesan demasiado). Es preciso, por tanto, escoger inteligentemente.

Comprobar todas las posibilidades se vuelve rápidamente imposible cuando el número de objetos aumenta. En efecto, hay que probar todas las combinaciones de 1 objeto, de 2, 3... hasta considerarlas todas, y eliminar aquellas soluciones imposibles porque son demasiado pesadas para, a continuación, seleccionar la mejor.

Imaginemos una mochila con una capacidad de 20 kg. Los objetos disponibles son los siguientes (peso y valor):



Es posible, por ejemplo, cargar los objetos A y H, lo que suma 4+16 = 20 kg y un total de 15+22 = 37 puntos de valor. Sin embargo, esta no es la mejor elección. Existe otra solución (que tampoco es óptima, aunque sí es mejor) que consiste en cargar C, I y K. Tenemos un peso igual a 10+5+4 = 19 kg y un valor total de 20+12+10 = 42.

La carga óptima es A, D, I, K y L. Se tiene un peso de 4+3+5+4+3 = 19 kg, y un valor total igual a 15+10+12+10+7 = 54.

3. Formulación de los problemas

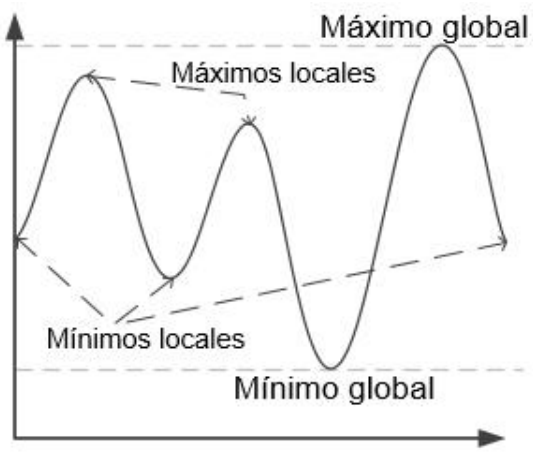
Todos los problemas de optimización pueden expresarse de la misma manera: existe una función f que asocia un valor con una solución x, denominada f(x). Se conoce una forma rápida de calcularla.

Existen, sin embargo, restricciones sobre x, y las soluciones deben pertenecer al conjunto X de las soluciones aceptables. En el caso de la mochila, se trata del conjunto de cargas cuyos pesos suman una cantidad inferior o igual a 20 kg.

La optimización consiste en encontrar x de cara a minimizar o a maximizar f(x).

En la práctica, nos interesamos únicamente en los mínimos. En efecto, buscar maximizar f(x) equivale a minimizar -f (x). Los algoritmos se presentarán únicamente para minimizaciones.

Los máximos y mínimos de una función se denominan **óptimos**. Los óptimos que se aplican a la función sobre todo el conjunto de definición se denominan **óptimos globales**. Aquellos que no son óptimos más que respecto a una pequeña vecindad se denominan **óptimos locales**.



4. Resolución matemática

La primera solución que nos viene a la mente consiste en estudiar la función f matemáticamente y encontrar el mínimo global de la función. Esto es posible en funciones que tienen una formulación matemática sencilla, y se estudia por otro lado en ciertos programas de instituto. Por ejemplo, la función f(x) = 3 + 1/x, con x perteneciente al conjunto [1,2] tiene un mínimo en 2 (vale entonces 3.5).

Sin embargo, en la realidad, las funciones pueden ser mucho más complejas de escribir o bien difíciles de estudiar para encontrar este mínimo matemático. Es el ejemplo del caso de la mochila.

En efecto, cada solución puede escribirse como un vector de dimensión N (siendo N el número de objetos posibles),

con cada valor que vale 0 o 1 (respectivamente para un objeto descartado o cargado). El peso de un objeto será p y su valor v.

En nuestro problema, tenemos 12 objetos y, por lo tanto, un vector de dimensión 12. La segunda solución (C, I y K) se expresa:

x = (0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0)

Las soluciones aceptables son aquellas que respetan un peso máximo de 20 kg, que matemáticamente se indica mediante la expresión:

$$x, \sum_{i=1}^{N} p(x_i) \leq 20$$

Esto se lee: el conjunto de las x, tales que la suma de los pesos de los componentes sea inferior a 20 kg.

La función a maximizar es:

$$f(x) = \sum_{i=1}^{n} x_i * v(x_i)$$

Se busca, por lo tanto, maximizar la suma de los valores de los objetos cargados (que valen 1).

La derivada de esta función no puede expresarse, y su estudio no es posible mediante técnicas matemáticas clásicas. Encontrar un óptimo matemático no es la solución.

5. Búsqueda exhaustiva

La segunda solución, tras la resolución matemática, es la **búsqueda exhaustiva**. Comprobando todas las posibilidades, necesariamente se encontrará la solución óptima.

Sin embargo, esta búsqueda es a menudo demasiado larga. En el problema de la mochila, existen demasiadas soluciones posibles, y su número aumenta de manera exponencial con el número de objetos posibles.

Para funciones cuyo espacio de búsqueda sea el conjunto de los números reales, el problema es todavía más complejo: en efecto, el número de valores entre otros dos valores concretos es siempre infinito. Comprobar todas las posibilidades resulta, por lo tanto, imposible.

Por ejemplo, si se intenta minimizar f(x) con x entre 1 y 2, existe una cantidad infinita de valores potenciales de x.

Podríamos tomar x = 0.5 y x = 0.6. Podríamos seleccionar 0.55, 0.578, 0.5896...

La búsqueda exhaustiva es, por lo tanto, en el mejor de los casos, demasiado larga para ejecutarse y, en el peor de los casos, completamente imposible.

6. Metaheurísticos

Al no poder resolver de forma determinista estos problemas complejos de optimización, es necesario utilizar otros métodos.

Existe una familia de métodos llamados **metaheurísticos**. Estos poseen diversas características:

- Son genéricos y pueden adaptarse a un gran número de problemas.
- Son iterativos, es decir, tratan de mejorar los resultados conforme avanzan.
- Son, a menudo, estocásticos, es decir, utilizan una parte más o menos importante de azar.
- No garantizan encontrar el óptimo global (salvo si se les deja ejecutarse durante un tiempo infinito), pero al menos sí un óptimo local de bastante calidad.

Existen diferentes metaheurísticos simples para la optimización y la búsqueda de óptimos. Lo que los diferencia es la forma en que cambian las soluciones y mejoran conforme avanza el tiempo.

Algoritmos voraces

Los **algoritmos voraces** son los más sencillos. No construyen más que una única solución, pero de manera iterativa. Así, conforme avanza el tiempo, se agrega un elemento, el más prometedor.

Este algoritmo debe adaptarse a cada problema. Solo se mantiene el principio general.

De este modo, en el caso de la mochila, agregaremos conforme avancemos los objetos más interesantes hasta alcanzar la capacidad de la mochila.

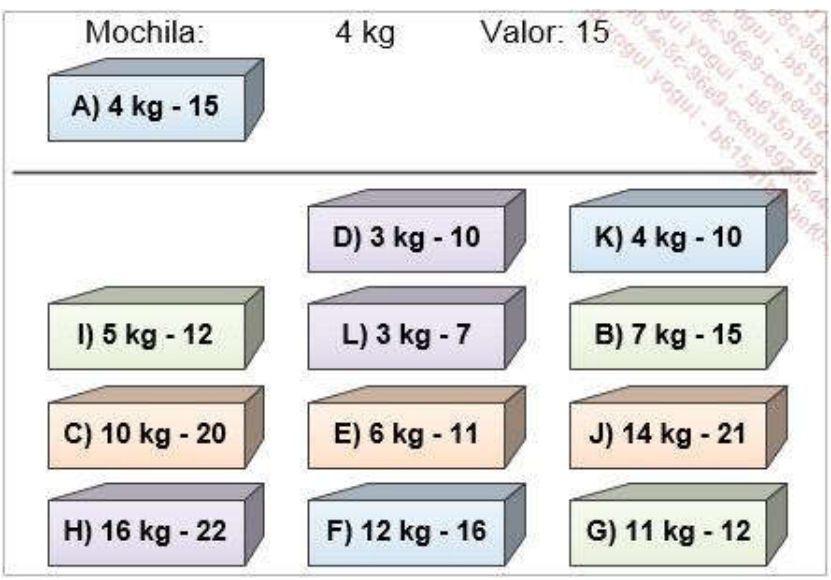
Para ello, empezamos calculando el valor por kilo de cada objeto:

A) 4kg - 15: 3.75	B) 7 kg - 15: 2.14	C) 10 kg - 20: 2	D) 3 kg - 10: 3.33
E) 6 kg - 11: 1.83	F) 12 kg - 16: 1.33	G) 11 kg - 12: 1.09	H) 16 kg - 22: 1.38
I) 5 kg - 12: 2.4	J) 14 kg - 21: 1.5	K) 4 kg - 10: 2.5	L) 3 kg - 7: 2.33

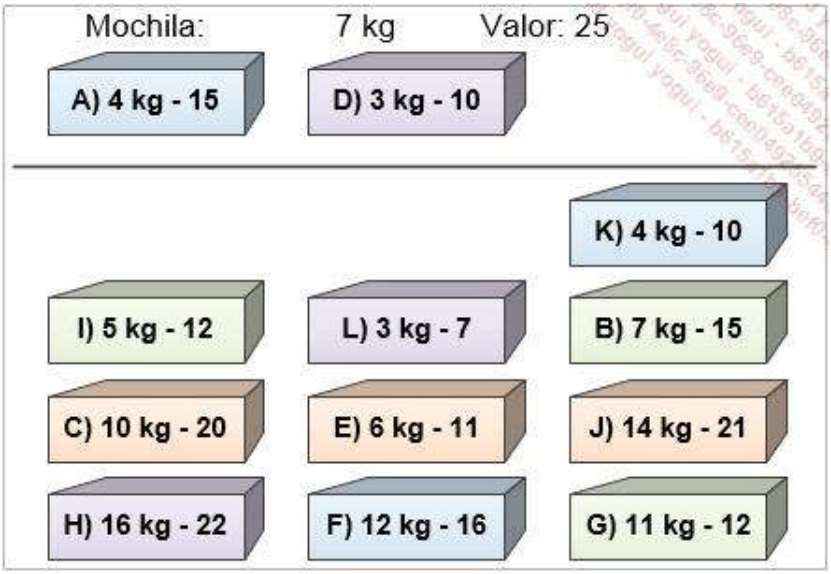
Ordenamos, a continuación, cada objeto desde el más interesante (el valor por kilo mayor) hasta el menos interesante. Se obtiene el orden siguiente:

A - D - K - I - L - B - C - E - J - H - F - G

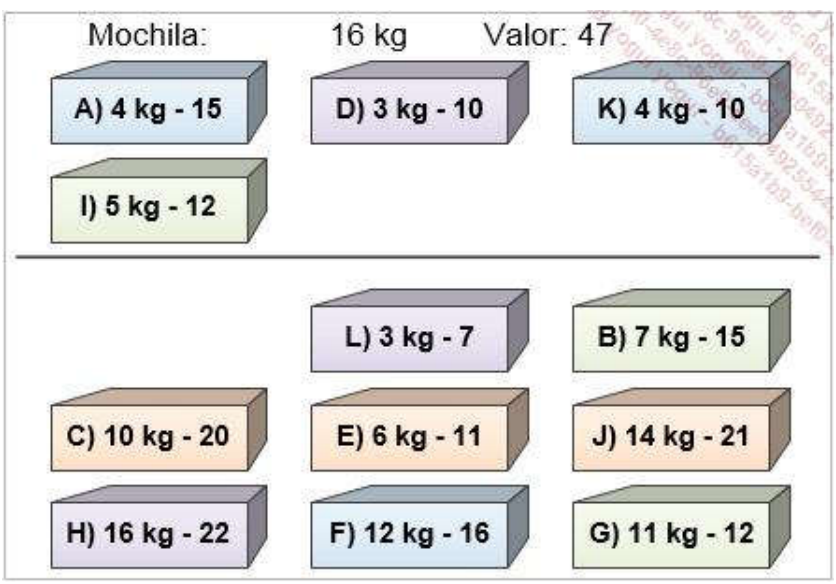
Se parte de una mochila vacía. Se agrega el primer elemento tras la ordenación, en este caso el objeto A. La mochila contiene, ahora, 4 kg y posee un valor total de 15.



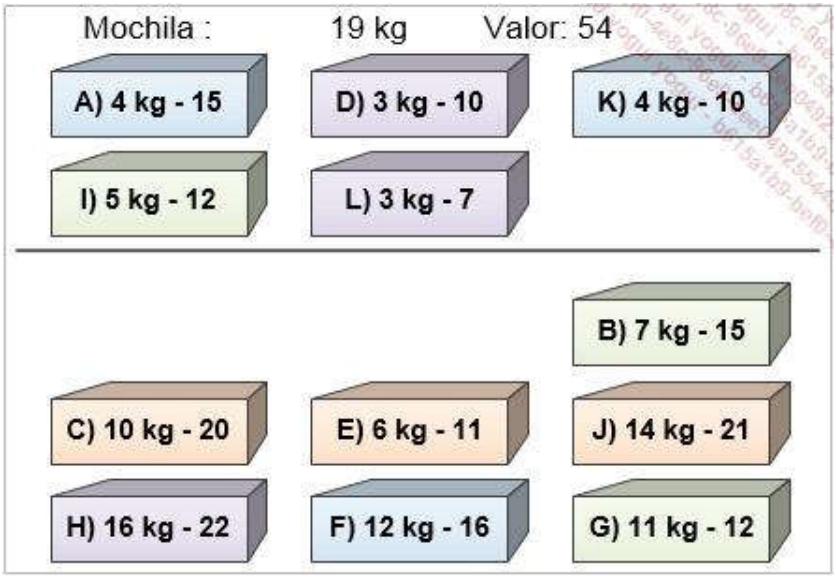
Se agrega, a continuación, el primer elemento de la lista ordenada restante. El objeto D no pesa más que 3 kg, de modo que es posible meterlo en la mochila.



Esta contiene, ahora, 7 kg y posee un valor de 25. El siguiente elemento es K. Una vez se agrega, la mochila contiene 11 kg y posee un valor igual a 35. El cuarto elemento es I.



Tenemos 16 kg y un valor total de 47. El siguiente elemento es L. La mochila contiene, ahora 19 kg y posee un valor de 54.



Los siguientes elementos de la lista tienen todos un peso demasiado grande como para entrar en ella (le queda solamente un kilo de carga autorizada). El algoritmo se detiene: la solución propuesta consiste en meter en la mochila los objetos A, D, I, K y L, para un valor total de 54. Se trata por otro lado del óptimo global para este ejemplo.

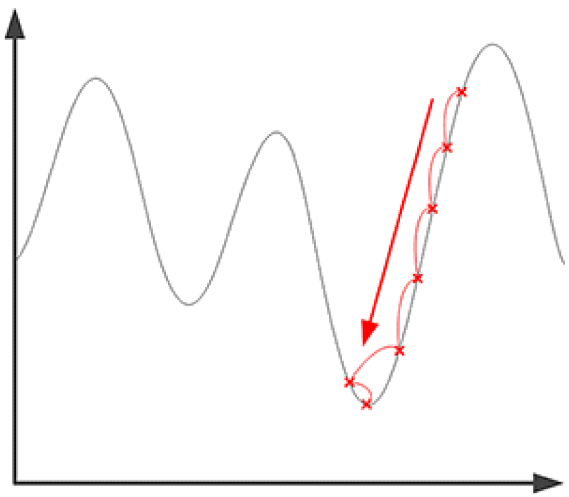
Descenso por gradiente

El **descenso por gradiente** es un metaheurístico incremental. A partir de una primera solución, escogida aleatoriamente o definida como base de partida (por ejemplo, la mejor solución conocida por los expertos), el algoritmo buscará una optimización sin modificar la solución más que en una unidad.

Cuando el problema es una función matemática, se calcula la derivada en el punto que representa la solución actual, y se sigue la dirección de la derivada más fuerte negativamente.

La derivada de una función representa su pendiente: si es positiva, entonces la curva es creciente, en caso contrario es decreciente. Además, cuanto mayor es la derivada, más acusada es la pendiente.

En el siguiente esquema, se indican las distintas soluciones obtenidas iterativamente: se parte de la solución más alta y se avanza en el sentido de la gradiente, hasta alcanzar el mínimo.



De manera intuitiva, este es el algoritmo utilizado por un senderista: si quiere alcanzar la cima de una montaña, pero no sabe dónde se encuentra, mira a su alrededor en qué dirección el camino sube más y sigue dicha dirección. A fuerza de subir, se verá, necesariamente, en lo alto del macizo sobre el que se encuentre. El procedimiento es el mismo si quiere alcanzar el valle, siguiendo rutas descendentes.

Por lo general, la derivada matemática no es accesible. No es posible, por lo tanto, seguirla directamente.

En su lugar, se calcularán las soluciones vecinas, a una distancia de una unidad (por definir). A continuación, se evalúa cada solución. Si se encuentra una solución mejor, entonces se parte de nuevo desde esta solución para una nueva iteración. En ausencia de mejora, nos detendremos.

Como con el algoritmo voraz, la elección de la unidad de modificación depende del problema que se quiera resolver. No es posible crear un algoritmo realmente muy genérico.

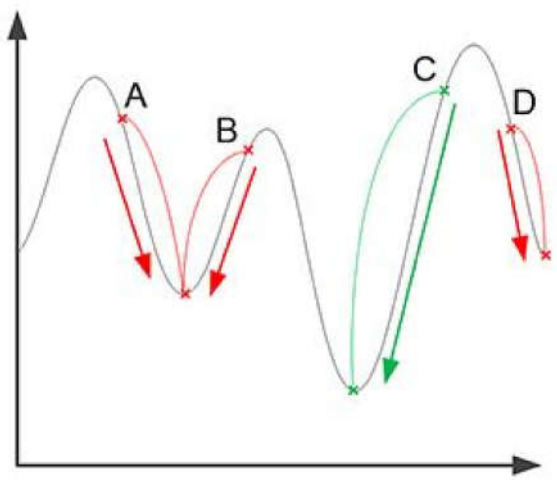
En el caso del problema de la mochila, podemos partir de una solución aleatoria. Se comprueban a continuación todas las variaciones, agregando un objeto que todavía no se había seleccionado o eliminando un objeto incluido en la mochila. Sin embargo, extraer un objeto va a disminuir necesariamente el valor de la mochila, por lo que será conveniente intercambiar objetos (se extrae uno para agregar otro(s) en su lugar).

Si la solución es aceptable (siempre que respete el peso máximo), entonces se evalúa. Solo se conserva la mejor solución entre todas las variaciones.

El descenso por gradiente tiene, sin embargo, algunos defectos:

- Es un algoritmo bastante lento, puesto que debe buscar todas las soluciones vecinas y evaluarlas.
- El algoritmo no encuentra más que un óptimo local. En efecto, solo se estudia el valle sobre el que se encuentra la solución inicial, y cuantos más óptimos locales tenga el problema, más difícil será encontrar el óptimo global.
- Incluso aunque la solución de partida se encuentre en el vecindario adecuado, si algún óptimo local tiene una derivada más fuerte que el óptimo global, va a atraer al algoritmo.

He aquí, por ejemplo, varias soluciones iniciales A a D. Solo la posición de partida C permite encontrar el óptimo global. Las soluciones A, B y D solo permiten encontrar un óptimo local.



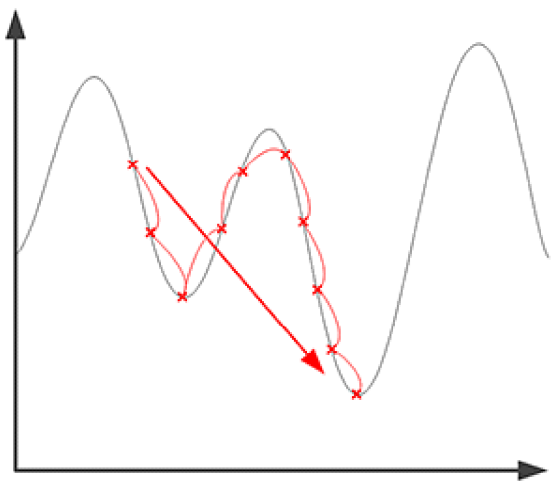
Para tratar de superar estos problemas, se utilizan a menudo varias inicializaciones (y, por lo tanto, varias soluciones de partida) para aumentar el número de óptimos descubiertos y, de este modo, aumentar la probabilidad de encontrar el óptimo global.

Búsqueda tabú

La **búsqueda tabú** es una mejora de la búsqueda mediante descenso por gradiente. En efecto, esta última se bloquea en el primer óptimo encontrado.

En el caso de la búsqueda tabú, con cada iteración, nos desplazamos hacia el mejor vecino, incluso aunque sea menos bueno que la solución actual. Además, se guarda una lista de las posiciones ya visitadas, que no se pueden seleccionar (de ahí el nombre, las anteriores soluciones se convierten en tabú).

De este modo, el algoritmo se "pasea" por el espacio de la solución y no se detiene en el primer óptimo descubierto. Se detendrá cuando todos los vecinos hayan sido visitados, tras un número de iteraciones máximo prefijado o cuando no se detecte ninguna mejora sustancial tras x pasos.



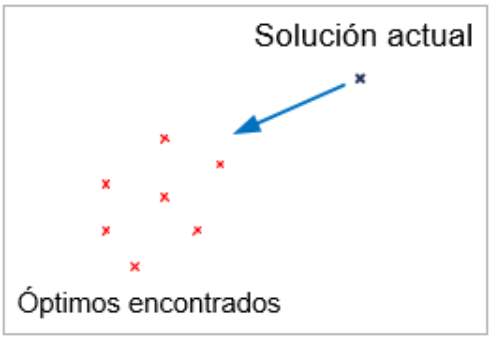
La principal dificultad de esta búsqueda es la elección de la longitud de la lista de posiciones tabú. En efecto, si esta lista es demasiado corta, corremos el riesgo de entrar en un bucle alrededor de las mismas posiciones. Por el contrario, una lista demasiado larga podría impedir probar otros caminos que partan de una misma solución potencial. No existe, sin embargo, ninguna manera de conocer la longitud de la lista ideal; debe seleccionarse de forma puramente empírica.

Esta lista se implementa, a menudo, como una lista (FIFO, del inglés *First In First Out*). De este modo, una vez que la lista ha alcanzado el tamaño máximo seleccionado, las posiciones registradas más antiguas salen de las posiciones tabú.

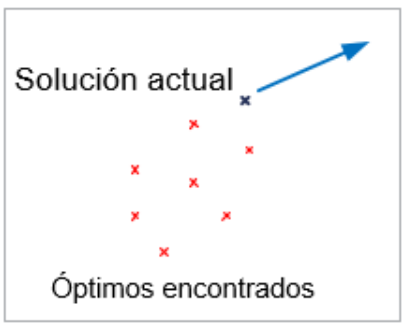
La elección del vecindario es idéntica a la utilizada en el descenso por gradiente, y la aplicación al problema de la mochila es idéntica (a saber, agregar un objeto o realizar un intercambio).

Es posible integrar otros dos procesos a la búsqueda tabú: la intensificación y la diversificación.

La **intensificación** consiste en favorecer ciertas zonas del espacio que parecen más prometedoras. En efecto, se registran todas las soluciones óptimas (locales o globales) encontradas hasta el momento. Se intenta comprobar de forma prioritaria las soluciones cercanas a la actual o que posean características similares. El desplazamiento está, por lo tanto, sesgado para aproximarse a estas soluciones:



A la inversa, la **diversificación** tiene como objetivo favorecer el hallazgo de nuevas zonas del espacio de búsqueda. De esta forma, se almacenan las posiciones probadas hasta el momento y se favorecen las soluciones diferentes. De este modo, es posible encontrar nuevas soluciones óptimas. El desplazamiento está, entonces, sesgado para alejarse de los antiguos óptimos:



Ambos procesos deben adaptarse a cada problema. Si no están equilibrados, se corre el riesgo de no hacer más que una intensificación (y, por lo tanto, quedarnos en el mismo lugar) o por el contrario una diversificación (y, por lo tanto, pasar junto a óptimos cercanos a los que ya se han descubierto).

En este caso, la adaptación resulta bastante empírica.

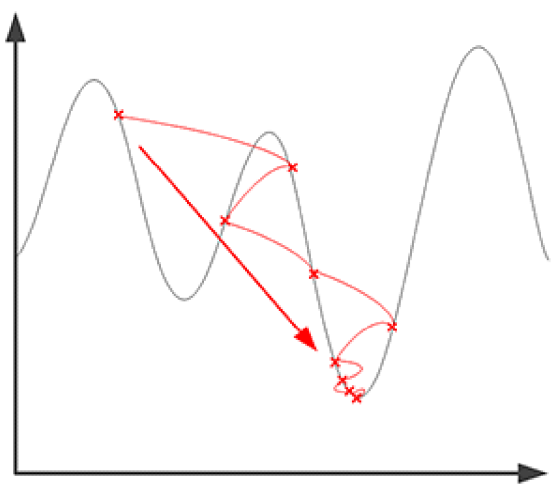
Recocido simulado

El **recocido simulado** mejora el descenso por gradiente y se inspira en el recocido utilizado en metalurgia. En efecto, cuando se están forjando o colando metales, estos sufren importantes restricciones. Es el caso de las hojas de las espadas, por ejemplo.

Para mejorar la dureza del filo, se calienta la hoja (de ahí el nombre de recocido). De este modo, los átomos pueden cristalizarse en estructuras más resistentes, y disminuyen las restricciones mecánicas y térmicas. Las hojas de buena calidad sufren, de este modo, varios ciclos de calor y formado.

En informática, se va a utilizar este principio para mejorar las soluciones y salir de los óptimos locales. Se va a fijar una **temperatura** numérica, que disminuirá conforme pase el tiempo. Cuanto mayor sea esta temperatura, más grandes podrán ser los saltos en el espacio de búsqueda. Además se acepta, a diferencia de lo que ocurre con el descenso por gradiente, transitar por soluciones menos óptimas que la solución actual.

El algoritmo empieza con una búsqueda global y va a encontrar zonas más interesantes. Cuando la temperatura decrece, se va a concentrar en una zona concreta, y termina como una búsqueda por gradiente clásica. La probabilidad de encontrar el óptimo global y no un óptimo local es, por lo tanto, mayor.



Conforme avanza el tiempo, se comprueba una solución vecina a la solución actual. Si mejora los resultados, se guarda. Si, por el contrario, es menos buena, se guarda con una probabilidad que depende de la temperatura. Escogemos para esta probabilidad un cálculo generalmente basado en una función exponencial, llamada "**regla de Metropolis**", aunque es posible construir variantes.

Esta regla define que la probabilidad de aceptar una solución menos óptima que la solución actual es:

$$P = e^{-\frac{\Delta E}{T}}$$

En esta ecuación, $\Delta \mathbf{E}$ representa la pérdida de calidad de la solución (considerada como una pérdida de energía, según la analogía con la física). T es la temperatura actual del sistema. Esta exponencial está siempre comprendida entre 0 y 1 y, conforme desciende la temperatura, mayor es la fracción y, por lo tanto, menor es la probabilidad.

La dificultad de este algoritmo radica en la elección de los parámetros. En efecto, es importante seleccionar la temperatura inicial y la ley de decrecimiento de esta: si la temperatura decrece de manera rápida, puede que el algoritmo no tenga tiempo suficiente para converger. Por el contrario, si la temperatura no decrece con la suficiente rapidez, el algoritmo puede salir permanentemente de las zonas que explora para explorar otras, sin encontrar jamás un óptimo.

La única forma de configurar estos parámetros es, de nuevo, el método empírico.

Optimización por enjambre de partículas

En la mayoría de los metaheurísticos, los resultados son mejores si se realizan varias ejecuciones a partir de soluciones iniciales diferentes. En efecto, esto permite recorrer una zona de búsqueda más amplia.

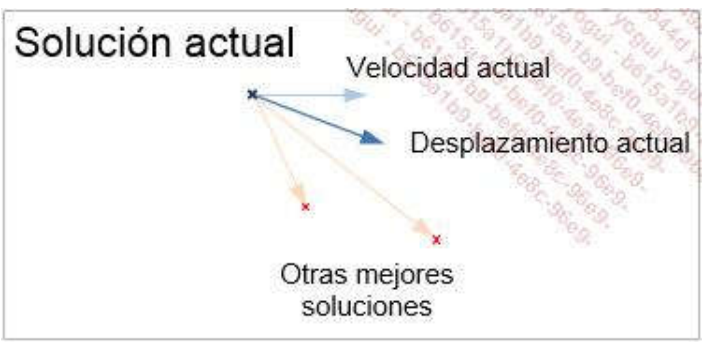
Sin embargo, es posible obtener varias veces la misma solución, y pasar al lado de un óptimo global (o de un mejor óptimo local).

La **optimización por enjambre de partículas** se inspira en la biología. En efecto, tanto en el comportamiento de los pájaros como de los peces podemos observar grandes grupos de animales que se desplazan en conjunto en tres dimensiones. Los pájaros (o los peces) no avanzan, sin embargo, dentro del grupo: la dirección de cada uno se adapta permanentemente en función de la dirección actual y la posición de los demás. Su velocidad también se adapta.

En este algoritmo, varias soluciones potenciales cohabitan en el espacio de búsqueda, y cada una se desplaza en una dirección determinada. Con cada iteración, las soluciones se van a desplazar como en una nube, avanzando hacia zonas que parezcan más interesantes.

Cada solución debe conocer su velocidad actual, según un vector (que permite indicar la dirección del desplazamiento) y las mejores posiciones descubiertas hasta el momento. Además, todas las soluciones del enjambre conocen la mejor solución actual (y su ubicación).

Hay ecuaciones sencillas (que se definen según el problema) que permiten actualizar la velocidad de desplazamiento y la posición de la solución en función de los distintos atributos (velocidad y mejores soluciones).



Podemos observar que este metaheurístico, a diferencia de los anteriores, no utiliza la derivada entre la solución actual y su vecindario. Esto permite a la optimización por enjambre de partículas aplicarse a más problemas.

Sin embargo, una vez más, la elección de parámetros es primordial. Si se escogen mal, la convergencia se realiza basándose en primer óptimo local descubierto. Por el contrario, es posible observar a los individuos desplazándose permanentemente desde una zona a otra sin converger jamás. Es preciso, por tanto, encontrar el equilibrio adecuado entre la **exploración** (para dejar que el enjambre descubra otras zonas) y la **explotación** (para buscar el óptimo local de la zona en curso).

Como con los demás algoritmos, esta elección se realiza de manera empírica.

Además, existe un sesgo en la búsqueda: los óptimos situados en el centro del espacio de búsqueda son más fáciles de alcanzar. En efecto, la velocidad se actualiza dimensión a dimensión. Puede ser adecuado transformar el espacio para evitar este sesgo o, por el contrario, aprovecharlo para mejorar los resultados.

Metaoptimización

Como la búsqueda de parámetros de los metaheurísticos es un problema complejo en sí mismo, podemos imaginar utilizar un algoritmo de optimización para realizar esta búsqueda.

Cuando se descubren parámetros mediante una búsqueda de óptimo, se habla de **metaoptimización**: la optimización del propio proceso de optimización.

Es posible utilizar los diferentes metaheurísticos expuestos en este capítulo, aunque también es posible utilizar otras técnicas como un sistema experto (que contendría las reglas propias de la experiencia de los investigadores) o algoritmos genéticos, o incluso redes neuronales.

Las distintas técnicas no son independientes y pueden utilizarse para complementarse entre sí y mejorar.

Dominios de aplicación

Estos algoritmos resultan muy útiles en muchos dominios, en particular aquellos para los que no exista ninguna manera de calcular el óptimo de forma matemática o cuando esta actividad llevaría demasiado tiempo.

Obtienen un óptimo, local o global. Se espera, entonces, tener un resultado; si no es global, al menos el más cercano a su nivel de calidad.

Los encontramos, de este modo, en todos los dominios que realicen el **diseño** de piezas o de sistemas. En efecto, permiten encontrar fácilmente formas o materiales adecuados, limitando su coste (o, en función del problema, la superficie de rozamiento, las turbulencias...). Se utilizan en **construcción**, por ejemplo, para optimizar las estructuras de carga.

Se han realizado estudios para optimizar el coste de las estructuras de hierro en construcciones que tenían que respetar las normas antisísmicas.

En **electrónica**, sirven para mejorar el diseño de circuitos impresos, limitando la cantidad de "cable" necesario o minimizando el espacio requerido por los diversos componentes.

En **finanzas**, los metaheurísticos permiten optimizar una cartera de acciones, limitando los riesgos y buscando maximizar las ganancias para una suma determinada.

Se utilizan en problemas de **planificación**, como, por ejemplo, crear horarios de autobús/avión/tren. Se busca minimizar el coste para la empresa y maximizar los beneficios. Para ello, se busca tener los vehículos en el garaje (o en el aeropuerto) el menor tiempo posible.

La programación y la **planificación** no solo afectan a los horarios. También pueden abordar la producción de bienes en función de las materias primas y de los recursos necesarios, o por el contrario saber cuándo y qué pedidos realizar en los plazos de producción previstos.

Los **militares** lo utilizan para asignar medios de defensa a un conjunto de ataques. El algoritmo de optimización permite ayudar al operador humano, que es el único que puede dar luz verde en caso de crisis. De este modo, es más fácil gestionar las tropas y los medios armados para hacer frente a ataques específicos en varios frentes.

En **telecomunicaciones**, pueden servir para mejorar el enrutado o la creación de redes, optimizando el tiempo de transferencia o la cantidad de cables necesarios para enlazar las distintas máquinas, respetando las restricciones de seguridad.

Las aplicaciones son, por tanto, numerosas y muy variadas, puesto que los metaheurísticos tienen una gran capacidad de adaptación y generan buenos resultados, con un coste de implementación asumible. Por lo tanto, se pueden utilizar en el dominio mucho más amplio de la **Machine Learning**, que permite mejorar los modelos.

Implementación

En primer lugar, se implementarán los algoritmos genéricos, y a continuación se crearán clases que heredan de las clases madres que permiten resolver el problema de la mochila.

Se utilizan dos versiones del problema de la mochila: la primera es la que se presenta como ejemplo en el algoritmo voraz (con 16 objetos) y la segunda es una versión más compleja y aleatoria, que permite comparar mejor los distintos algoritmos.

Se realiza un análisis de los resultados obtenidos al final de esta sección.

1. Clases genéricas

En primer lugar, hay que definir algunas clases e interfaces muy genéricas. Nos permitirán crear, a continuación, los distintos algoritmos.

ISolucion es una interfaz que representa una solución potencial para un problema determinado. La única obligación para esta solución es tener una propiedad que permita conocer su valor.

```
public interface ISolucion {
    double getValor();
}
```

Es posible, ahora, definir un problema gracias a una interfaz **IProblema**. Debemos poder obtener una solución aleatoria (`SolucionAleatoria`), el vecindario de una solución (`Vecindario`) y por último la mejor solución de entre una lista pasada como parámetro (`MejorSolucion`). Todos estos métodos se utilizarán en los diversos algoritmos.

```
import java.util.ArrayList;
public interface IProblema {
    ArrayList<ISolucion> Vecindario(ISolucion solucionActual);
    ISolucion SolucionAleatoria();
    ISolucion MejorSolucion(ArrayList<ISolucion> soluciones);
}
```

De cara a tener un código lo más genérico posible, vamos a separar también la interfaz **IHM** del resto del programa. De este modo, el código propuesto está disponible para distintos tipos de aplicación sin necesidad de modificación. El programa principal, en sí, es una aplicación en modo consola. El único método necesario permite mostrar un mensaje que se pasa como parámetro.

```
public interface IHM {
    void MostrarMensaje(String msg);
}
```

Algoritmo es la última clase genérica. Posee únicamente dos métodos: uno que pide resolver un problema y otro que permite mostrar el resultado del algoritmo. Además, tiene dos atributos: uno que permite tener un enlace hacia el problema que se ha de resolver y otro hacia la clase que sirve como interfaz con el usuario.

```
public abstract class Algoritmo {
    protected IProblema problema;
    protected IHM ihm;

    public void Resolver(IProblema _pb, IHM _ihm) {
        problema = _pb;
        ihm = _ihm;
    }

    protected abstract void DevolverResultado();
}
```

2. Implementación de los distintos algoritmos

Una vez codificadas las clases genéricas, podemos pasar a los distintos algoritmos. Estos son genéricos, y se trata de clases abstractas. Habrá que crear las clases hijas para cada algoritmo y para cada problema: los metaheurísticos son generalizables y es preciso adaptarlos.

Además, cada clase posee un método `Resolver` que es un patrón de método: este método está sellado (palabra clave `final`), lo que quiere decir que las clases derivadas no podrán redefinirlo. Por el contrario, utiliza métodos abstractos, que deben redefinirse. De este modo, el algoritmo global permanece fijo, pero los detalles de implementación se dejan a cargo de cada clase hija.

a. Algoritmo voraz

El algoritmo voraz **`AlgoritmoVoraz`** es el más sencillo: construiremos una solución, parte a parte, y mostraremos esta solución. Necesitamos únicamente un método `ConstruirSolucion`.

```
public abstract class AlgoritmoVoraz extends Algoritmo {
    @Override
    public final void Resolver(IProblema pb, IHM ihm) {
        super.Resolver(pb, ihm);
        ConstruirSolucion();
        DevolverResultado();
    }

    protected abstract void ConstruirSolucion();
}
```

b. Descenso por gradiente

El siguiente algoritmo es **`DescensoGradiente`**, para el descenso por gradiente. El algoritmo general consiste en crear una primera solución aleatoria y, a continuación, mientras no se alcance un criterio de parada, se pide el vecindario de una solución; si el vecindario existe, se escoge la mejor solución en su interior. Se realiza entonces la actualización de la solución en curso mediante `Actualizar` (que cambia o no la solución en función de si se produce o no una mejora).

El bucle termina incrementando las distintas variables que permiten alcanzar el criterio de parada. A continuación, se muestran los resultados.

Necesitaremos los siguientes métodos:

- `CriterioParada`: indica si se han alcanzado o no los criterios de parada.
- `Actualizar`: actualiza (o no) la solución actual, en función de la nueva solución propuesta como parámetro.
- `Incrementar`: actualiza los criterios de parada en cada bucle.

```
import java.util.ArrayList;

public abstract class DescensoGradiente extends Algoritmo {
    protected ISolucion solucionActual;

    @Override
    public final void Resolver(IProblema _pb, IHM ihm) {
        super.Resolver(_pb, ihm);

        solucionActual = problema.SolucionAleatoria();
        while(!CriterioParada()) {
            ArrayList<ISolucion> vecindario =
problema.Vecindario(solucionActual);
            if (vecindario != null) {
                ISolucion mejorSolucion =
problema.MejorSolucion(vecindario);
                Actualizar(mejorSolucion);
            }
            Incrementar();
        }
        DevolverResultado();
    }

    protected abstract boolean CriterioParada();
    protected abstract void Actualizar(ISolucion solucion);
    protected abstract void Incrementar();
}
```

c. Búsqueda tabú

La búsqueda tabú **BusquedaTabu** es más compleja: se va a mantener la mejor solución encontrada hasta el momento y la solución en curso.

Empezamos inicializando la solución de partida aleatoriamente. A continuación, mientras no se alcancen los criterios de parada (método `CriterioParada`), se crea el vecindario de la solución actual. En él, las soluciones presentes en la lista de posiciones tabús se eliminan (`EliminarSolucionesTabus`). Se guarda, a continuación, la mejor solución con su actualización (o no) gracias al método `Actualizar`.

Solo queda incrementar los contadores (`Incrementar`) y después, al final del bucle, mostrar el resultado.

```
import java.util.ArrayList;

public abstract class BusquedaTabu extends Algoritmo {
    protected ISolucion solucionActual;
    protected ISolucion mejorSolucion;
```

```
    @Override
    public final void Resolver(IProblema pb, IHM ihm) {
        super.Resolver(pb, ihm);

        solucionActual = problema.SolucionAleatoria();
        mejorSolucion = solucionActual;
        AgregarListaTabu(solucionActual);

        while (!CriterioParada()) {
            ArrayList<ISolucion> vecindario =
problema.Vecindario(solucionActual);
            if (vecindario != null) {
                vecindario = EliminarSolucionesTabus(vecindario);
                ISolucion mejorVecino =
problema.MejorSolucion(vecindario);
                if (mejorVecino != null) {
                    Actualizar(mejorVecino);
                }
            }
            Incrementar();
        }
        DevolverResultado();
    }

    protected abstract void AgregarListaTabu(ISolucion solucion);
    protected abstract ArrayList<ISolucion>
EliminarSolucionesTabus(ArrayList<ISolucion> vecindario);
    protected abstract boolean CriterioParada();
    protected abstract void Actualizar(ISolucion solucion);
    protected abstract void Incrementar();
}
```

d. Recocido simulado

En el caso del recocido simulado **RecocidoSimulado**, se parte del descenso por gradiente. Guardamos, sin embargo, la mejor solución encontrada hasta el momento, puesto que podemos aceptar una solución menos óptima.

Empezamos inicializando la temperatura (`InicializarTemperatura`). Repetimos el bucle, a continuación, hasta alcanzar los criterios de parada (`CriterioParada`). En cada iteración, se recupera el vecindario de la solución en curso, y en particular la mejor solución en su interior. A continuación se actualiza (o no) esta última (`Actualizar`). En este método se utiliza la temperatura.

El bucle termina incrementando las variables internas (`Incrementar`) y modificando la temperatura (`ActualizarTemperatura`). El algoritmo termina con la visualización de los resultados.

```
import java.util.ArrayList;

public abstract class RecocidoSimulado extends Algoritmo {
    protected ISolucion solucionActual;
    protected ISolucion mejorSolucion;
    protected double temperatura;
```

```
    @Override
    public final void Resolver(IProblema pb, IHM ihm) {
        super.Resolver(pb, ihm);

        solucionActual = problema.SolucionAleatoria();
        mejorSolucion = solucionActual;
        InicializarTemperatura();

        while (!CriterioParada()) {
            ArrayList<ISolucion> vecindario =
problema.Vecindario(solucionActual);
            if (vecindario != null) {
                ISolucion mejorVecino =
problema.MejorSolucion(vecindario);
                Actualizar(mejorVecino);
            }
            Incrementar();
            ActualizarTemperatura();
        }
        DevolverResultado();
    }

    protected abstract void ActualizarTemperatura();
    protected abstract void InicializarTemperatura();
    protected abstract boolean CriterioParada();
    protected abstract void Actualizar(ISolucion solucion);
    protected abstract void Incrementar();
}
```

e. Optimización por enjambre de partículas

El último algoritmo es la optimización por enjambre de partículas **`EnjambreParticulas`**. Este es muy diferente de los otros: en efecto, en lugar de utilizar un vecindario compuesto de varias soluciones y mantener una sola, se tiene una población de soluciones (`soluciones`) que se desplazarán en el entorno. Además de la mejor solución encontrada hasta el momento, se guardará también la mejor solución en el seno de la población.

La constante `NUM_INDIVIDUOS` indica el número de individuos utilizados en nuestra población (en este caso, 30). Se trata de uno de los parámetros que es preciso configurar.

Empezamos inicializando nuestra población aleatoriamente y actualizando nuestras mejores soluciones iniciales. A continuación, se repite el bucle hasta alcanzar un criterio de parada (`CriterioParada`).

En cada iteración, se actualizan las mejores soluciones globales (`ActualizarVariables`), a continuación las posiciones de las soluciones (`ActualizarSoluciones`) y por último se incrementan las variables utilizadas para el criterio de parada (`Incrementar`). Una vez terminado el bucle, se muestran los resultados.

```
import java.util.ArrayList;

public abstract class EnjambreParticulas extends Algoritmo {
    protected ArrayList<ISolucion> soluciones;
    protected ISolucion mejorSolucion;
    protected ISolucion mejorActual;
```

```
    protected final static int NUM_INDIVIDUOS = 30;

    @Override
    public final void Resolver(IProblema pb, IHM ihm) {

        super.Resolver(pb, ihm);
        soluciones = new ArrayList();
        for (int i = 0; i < NUM_INDIVIDUOS; i++) {
            ISolucion nuevaSol = problema.SolucionAleatoria();
            soluciones.add(nuevaSol);
        }
        mejorSolucion = problema.MejorSolucion(soluciones);
        mejorActual = mejorSolucion;

        while (!CriterioParada()) {
            ActualizarVariables();
            ActualizarSoluciones();
            Incrementar();
        }

        DevolverResultado();
    }

    protected abstract void ActualizarVariables();
    protected abstract void ActualizarSoluciones();
    protected abstract boolean CriterioParada();
    protected abstract void Incrementar();
}
```

3. Resolución del problema de la mochila

A continuación vamos a aplicar los metaheurísticos al problema de la mochila. Para ello, derivamos en primer lugar las interfaces de base (`ISolucion` e `IProblema`), y a continuación codificaremos los distintos métodos necesarios para los distintos algoritmos.

a. Implementación del problema

Antes de poder codificar el problema, es necesario definir el contenido de la mochila: las cajas (**`Caja`**). Cada caja posee un peso y un valor, así como un nombre que servirá, principalmente, para la representación.

Para facilitar la creación de las cajas y su manejo, se agrega un constructor, que permite actualizar los tres atributos. El método `toString` también se sobrecarga para poder mostrar la información de la caja.

```
public class Caja {
    public double peso;
    public double valor;
    protected String nombre;

    public Caja(String _nombre, double _peso, double _valor) {
        nombre = _nombre;
        peso = _peso;
```

```
        valor = _valor;
    }

    @Override
    public String toString() {
        return nombre + " (" + peso + ", " + valor + ")";
    }
}
```

Una vez definidas las cajas, es posible crear las soluciones **SolucionMochila**, que implementan la interfaz `ISolucion`.

```
import java.util.ArrayList;
import java.util.StringJoiner;

public class SolucionMochila implements ISolucion {
    // Aquí el código
}
```

En primer lugar, definiremos un atributo correspondiente al contenido de la mochila (una lista de cajas). Agregamos también un constructor que permite inicializar esta lista y otro que copia el contenido de una solución que se pasa como parámetro.

```
    public ArrayList<Caja> contenido;

    public SolucionMochila() {
        contenido = new ArrayList();
    }

    public SolucionMochila(SolucionMochila original) {
        contenido = new ArrayList();
        contenido.addAll(original.contenido);
    }
```

Son necesarios dos métodos:

- `getPeso`: calcula el peso de una solución, sumando los pesos de cada caja.
- `getValor`: calcula el valor total de una solución, caja por caja (se trata de la implementación del método abstracto definido en la clase madre).

```
    public double getPeso() {
        double peso = 0.0;
        for (Caja b : contenido) {
             peso += b.peso;
        }
        return peso;
    }

    @Override
```

```
    public double getValor() {
        double valor = 0.0;
        for (Caja b : contenido) {
             valor += b.valor;
        }
        return valor;
    }
```

Se redefinen, entonces, los tres métodos de base de los objetos: `toString`, `equals` (para comparar dos mochilas) y `hashCode`. El método `toString` creará simplemente una cadena con el valor, el peso y el contenido de la mochila. Se utilizará para ello un `StringJoiner`.

```
    @Override
    public String toString() {
        StringJoiner sj = new StringJoiner(" - ");
        sj.add("Valor: " + getValor());
        sj.add("Peso: " + getPeso());
        for(Caja b : contenido) {
            sj.add(b.toString());
        }
        return sj.toString();
    }
```

Para el comparador de igualdad, debemos comprobar primero si ambas mochilas contienen el mismo número de objetos, el mismo peso y el mismo valor. Si se da el caso, se comprueba entonces si cada caja contenida en la primera mochila se encuentra en la segunda. No es posible comparar directamente las listas, puesto que los objetos pueden encontrarse en un orden diferente.

```
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof SolucionMochila)) {
            return false;
        }
        SolucionMochila sol = (SolucionMochila) o;
        if (sol.contenido.size() != this.contenido.size() ||
sol.getPeso() != this.getPeso() || sol.getValor() !=
this.getValor()) {
            return false;
        }
        for(Caja b : contenido) {
            if (!sol.contenido.contains(b)) {
                return false;
            }
        }
        return true;
    }
```

El método `hashCode` sirve para diferenciar rápidamente las soluciones si se utilizan en un diccionario, por ejemplo. Desafortunadamente, ninguno de los atributos actuales de la solución es fijo. Aquí se devuelve siempre el mismo valor, es decir, 42. Esto no permite utilizar una tabla de hash, aunque no es nuestro objetivo.

```
    @Override
    public int hashCode() {
        return 42;
    }
```

Terminamos con la implementación del problema de la mochila **ProblemaMochila**, que implementa `IProblema`.

Este objeto contiene una lista de cajas disponibles para incluir en las mochilas llamada `cajasDispo`, así como un peso máximo admitido (en forma de atributo `pesoMax`), un generador aleatorio que se define estático y, por último, una constante que indica el número de vecinos que se creará para cada solución. Este número puede modificarse en función de las necesidades.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;

public class ProblemaMochila implements IProblema {
    protected ArrayList<Caja> cajasDispo = null;
    public double pesoMax;
    public static Random generador = null;
    public final static int NUM_VECINOS = 30;

    // Resto de código aquí
}
```

Hay dos constructores disponibles:

- El constructor por defecto construye el ejemplo presentado al comienzo del capítulo, con una mochila con una capacidad de 20 kg y 12 cajas disponibles.
- Un segundo constructor, que construye un nuevo problema. Se le pasa como parámetro el número de cajas disponibles, el tamaño de la mochila y el valor máximo de cada caja. Estas se crean, a continuación, aleatoriamente.

```
    public ProblemaMochila() {
        // Lista de cajas
        cajasDispo = new ArrayList();
        cajasDispo.add(new Caja("A", 4, 15));
        cajasDispo.add(new Caja("B", 7, 15));
        cajasDispo.add(new Caja("C", 10, 20));
        cajasDispo.add(new Caja("D", 3, 10));
        cajasDispo.add(new Caja("E", 6, 11));
        cajasDispo.add(new Caja("F", 12, 16));
        cajasDispo.add(new Caja("G", 11, 12));
        cajasDispo.add(new Caja("H", 16, 22));
        cajasDispo.add(new Caja("I", 5, 12));
        cajasDispo.add(new Caja("J", 14, 21));
        cajasDispo.add(new Caja("K", 4, 10));
        cajasDispo.add(new Caja("L", 3, 7));

        pesoMax = 20;
```

```
        if (generador == null) {
            generador = new Random();
        }
    }

    public ProblemaMochila(int numCajas, double _pesoMax, double
valorMax) {
        cajasDispo = new ArrayList();
        pesoMax = _pesoMax;
        if (generador == null) {
            generador = new Random();
        }
        for (int i = 0; i < numCajas; i++) {
            cajasDispo.add(new Caja(Integer.toString(i),
generador.nextDouble() * pesoMax, generador.nextDouble() *
valorMax));
        }
    }
```

Se agrega un método `Cajas` que devuelve una copia de la lista de cajas disponibles.

```
    public ArrayList<Caja> Cajas() {
        ArrayList<Caja> copia = new ArrayList();
        copia.addAll(cajasDispo);
        return copia;
    }
```

Por último, es preciso implementar los tres métodos de la interfaz `IProblema`. El primero consiste en crear aleatoriamente una nueva solución. Para ello, se echan a suertes las cajas una a una. Sin embargo, prestaremos atención a no seleccionar cajas ya escogidas ni demasiado pesadas. Se crea para ello un método complementario `EliminarDemasiadoPesadas`. Cuando no pueden agregarse más cajas a la mochila, se devuelve la solución creada.

```
    @Override
    public ISolucion SolucionAleatoria() {
        SolucionMochila solucion = new SolucionMochila();
        ArrayList<Caja> cajasPosibles = Cajas();
        double espacioDispo = pesoMax;
        EliminarDemasiadoPesadas(cajasPosibles, espacioDispo);
        while(espacioDispo > 0 && !cajasPosibles.isEmpty()) {
            int indice = generador.nextInt(cajasPosibles.size());
            Caja b = cajasPosibles.get(indice);
            solucion.contenido.add(b);
            cajasPosibles.remove(indice);
            espacioDispo -= b.peso;
            EliminarDemasiadoPesadas(cajasPosibles, espacioDispo);
        }
        return solucion;
    }
    public void EliminarDemasiadoPesadas(ArrayList<Caja>
cajasPosibles, double  espacioDispo) {
        Iterator<Caja> iterador = cajasPosibles.iterator();
```

```
        while (iterador.hasNext()) {
            Caja b = iterador.next();
            if (b.peso > espacioDispo) {
                iterador.remove();
            }
        }
    }
```

El segundo método consiste en escoger la mejor solución de una lista. Para nosotros, se trata simplemente de buscar la solución con el valor máximo.

Para ello, se recorre la lista y se conserva la mejor solución conforme se avanza.

```
    @Override
    public ISolucion MejorSolucion(ArrayList<ISolucion>
soluciones) {
        if (!soluciones.isEmpty()) {
             ISolucion mejor = soluciones.get(0);
             for (ISolucion sol : soluciones) {
                 if (sol.getValor() > mejor.getValor()) {
                     mejor = sol;
                 }
             }
             return mejor;
        }
        else {
             return null;
        }
    }
```

El último método, `Vecindario`, consiste en devolver el vecindario de la solución que se pasa como parámetro. Para ello, se realizan ligeras modificaciones a la solución: se elimina un objeto al azar y se completa el espacio liberado por otras cajas, aleatoriamente. Se repite tantas veces como vecinos se desee tener.

```
    @Override
    public ArrayList<ISolucion> Vecindario(ISolucion solucionActual) {
        ArrayList<ISolucion> vecindario = new ArrayList();
        for (int i = 0; i < NUM_VECINOS; i++) {
            SolucionMochila solucion = new
SolucionMochila((SolucionMochila)  solucionActual);
            int indice = generador.nextInt(solucion.contenido.size());
            solucion.contenido.remove(indice);
            ArrayList<Caja> cajasPosibles = Cajas();
            double espacioDispo = pesoMax - solucion.getPeso();
            cajasPosibles.removeAll(solucion.contenido);
            EliminarDemasiadoPesadas(cajasPosibles, espacioDispo);
            while(espacioDispo > 0 && !cajasPosibles.isEmpty()) {
                indice = generador.nextInt(cajasPosibles.size());
                Caja b = cajasPosibles.get(indice);
                solucion.contenido.add(b);
                cajasPosibles.remove(indice);
```

```
                espacioDispo -= b.peso;
                EliminarDemasiadoPesadas(cajasPosibles, espacioDispo)
            }
            vecindario.add(solucion);
        }
        return vecindario;
    }
}
```

El problema de la mochila está ahora completamente codificado. Solo nos queda derivar los distintos algoritmos y después al programa principal.

b. Algoritmo voraz

Para implementar los distintos algoritmos, no es preciso codificar la estructura del algoritmo (que ya se encuentra definida en el método `Resolver` de las clases ya codificadas), sino únicamente los métodos que implementan los detalles.

Para el algoritmo voraz, la clase **`AlgoritmoVorazMochila`** tiene que crear solamente dos métodos: uno que permita construir la solución y otro que permita mostrar el resultado.

Esta clase también posee un único atributo que es la solución en curso de construcción. El método `DevolverResultado` consiste, simplemente, en mostrarla.

Para el método `ConstruirSolucion`, se pide en primer lugar la lista de cajas que pueden utilizarse. Se las ordena, a continuación, de mayor a menor según la relación "valor/peso". Mientras quede espacio en la mochila, se agregan cajas en orden.

Para la ordenación, se utiliza una expresión lambda que permite indicar qué caja es mejor que otra.

El código es el siguiente:

```
import java.util.ArrayList;
import java.util.Collections;

public class AlgoritmoVorazMochila extends AlgoritmoVoraz {
    SolucionMochila solucion;

    @Override
    protected void ConstruirSolucion() {
        solucion = new SolucionMochila();
        ProblemaMochila pb = (ProblemaMochila) problema;
        ArrayList<Caja> cajasPosibles = pb.Cajas();
        Collections.sort(cajasPosibles, (Caja b1, Caja b2) ->
(int) (((b2.valor/b2.peso) >= (b1.valor/b1.peso)) ? 1 : -1));
        double espacioDispo = pb.pesoMax;
        for (Caja b : cajasPosibles) {
             if (b.peso <= espacioDispo) {
                 solucion.contenido.add(b);
                 espacioDispo -= b.peso;
             }
        }
    }
```

```
    @Override
    protected void DevolverResultado() {
        ihm.MostrarMensaje(solucion.toString());
    }
}
```

c. Descenso por gradiente

Para el descenso por gradiente, la clase **`DescensoGradienteMochila`** debe contener cuatro métodos.

El criterio de parada es el número de iteraciones sin encontrar una mejora. En efecto, el vecindario, al estar creado aleatoriamente, hace que sea preciso comprobar varios elementos para saber si existe o no una mejora posible alrededor del punto actual. Se fija el número máximo de iteraciones a 50.

```
public class DescensoGradienteMochila extends DescensoGradiente {
    protected int numIteracionesSinMejora = 0;
    protected final static int
NUM_MAX_ITERACIONES_SIN_MEJORA = 50;

    // Métodos aquí
}
```

El primer método es `CriterioParada`, que debe indicar cuándo se ha alcanzado el criterio de parada. Basta con verificar si se ha alcanzado el límite del número de iteraciones sin mejora.

```
    @Override
    protected boolean CriterioParada() {
        return numIteracionesSinMejora >=
NUM_MAX_ITERACIONES_SIN_MEJORA;
    }
```

El segundo método es `Incrementar`, que debe incrementar las distintas variables internas. Aquí, tenemos que incrementar únicamente el número de iteraciones sin mejora.

```
    @Override
    protected void Incrementar() {
        numIteracionesSinMejora++;
    }
```

El tercer método es el que permite actualizar (o no) una solución reemplazándola por la que se pasa como parámetro. Para ello, simplemente se mira si el valor es mejor que el de la solución en curso. En caso afirmativo, se reemplaza esta y se pone a 0 el contador que indica el número de iteraciones sin mejora.

```
    @Override
    protected void Actualizar(ISolucion solucion) {
        if (solucion.getValor() > solucionActual.getValor()) {
            solucionActual = solucion;
```

```
            numIteracionesSinMejora = 0;
        }
    }
```

El último método consiste simplemente en devolver el resultado, a saber, la solución en curso.

```
    @Override
    protected void DevolverResultado() {
        ihm.MostrarMensaje(solucionActual.toString());
    }
```

d. Búsqueda tabú

La búsqueda tabú es más compleja incluso una vez (que ya era el caso del algoritmo genérico). En efecto, requiere tener una lista de posiciones tabús, lista que es de tamaño fijo. Además, es preciso fijar un criterio de parada más complejo, para evitar realizar bucles entre varias posiciones.

Seleccionamos un doble criterio de parada: en primer lugar, contamos el número de iteraciones durante las que no se ha encontrado mejora alguna, limitándose a `NUM_MAX_ITERACIONES_SIN_MEJORA`, constante que se fija previamente. Además, nos detendremos también tras un número de iteraciones prefijado `NUM_MAX_ITERACIONES`.

Nuestra clase **`BusquedaTabuMochila`** posee tres atributos (los dos números de iteraciones y la lista de las posiciones tabú) y tres constantes.

```
import java.util.ArrayList;

public class BusquedaTabuMochila extends BusquedaTabu {
    protected int numIteracionesSinMejora = 0;
    protected int numIteraciones = 0;
    protected ArrayList<SolucionMochila> listaTabu = new ArrayList();

    private final static int NUM_MAX_ITERACIONES = 100;
    private final static int NUM_MAX_ITERACIONES_SIN_MEJORAS = 30;
    private final static int NUM_MAX_POSICIONES_TABUS = 50;

    // Otros métodos aquí
}
```

Esta clase implementa la clase **`BusquedaTabu`**, de modo que hace falta desarrollar sus seis métodos abstractos. El primero consiste en indicar si se ha alcanzado el criterio de parada. Para ello, verificamos si se ha superado alguno de nuestros dos valores máximos de iteraciones.

```
    @Override
    protected boolean CriterioParada() {
        return (numIteraciones > NUM_MAX_ITERACIONES ||
numIteracionesSinMejora > NUM_MAX_ITERACIONES_SIN_MEJORAS);
    }
```

El segundo método consiste en actualizar (o no) la solución actual por la solución que se pasa como parámetro. Para ello, verificamos simplemente si la mejor solución propuesta está contenida en la lista de posiciones tabú. Si no, se actualiza la posición y se agrega a la lista de posiciones tabú. Además, si se alcanza una solución mejor que la obtenida hasta el momento, se actualiza la variable `mejorSolucion`. Por último, ponemos a cero el número de iteraciones sin actualización.

```
@Override
protected void Actualizar(ISolucion solucion) {
    if (!listaTabu.contains(solucion)) {
        solucionActual = solucion;
        AgregarListaTabu(solucion);
        if (mejorSolucion.getValor() < solucion.getValor()) {
            mejorSolucion = solucion;
            numIteracionesSinMejora = 0;
        }
    }
}
```

El método siguiente permite actualizar las variables internas. Incrementamos, entonces, las dos variables correspondientes al número de iteraciones.

```
@Override
protected void Incrementar() {
    numIteracionesSinMejora++;
    numIteraciones++;
}
```

El cuarto es el que permite mostrar la mejor solución.

```
@Override
protected void DevolverResultado() {
    ihm.MostrarMensaje(mejorSolucion.toString());
}
```

El quinto método agrega la posición indicada a la lista de posiciones tabú. Para ello, se verifica en primer lugar si se ha alcanzado el número máximo de posiciones, y en caso afirmativo se elimina la primera de la lista antes de agregar la nueva posición.

```
@Override
protected void AgregarListaTabu(ISolucion solucion) {
    while (listaTabu.size() >= NUM_MAX_POSICIONES_TABUS) {
        listaTabu.remove(0);
    }
    listaTabu.add((SolucionMochila) solucion);
}
```

Para terminar, el último método consiste simplemente en devolver las posiciones de un vecindario que no están en

la lista tabú. Para ello, se utiliza el método `removeAll`.

```
    @Override
    protected ArrayList<ISolucion>
EliminarSolucionesTabus(ArrayList<ISolucion> vecindario) {
        vecindario.removeAll(listaTabu);
        return vecindario;
    }
```

La búsqueda tabú está ahora operacional.

e. Recocido simulado

El recocido simulado debe aceptar soluciones menos buenas que la solución actual, con una probabilidad que depende de la temperatura actual.

La clase **RecocidoSimuladoMochila** hereda de la clase abstracta `RecocidoSimulado`. Solo deben codificarse los detalles de la implementación adaptados al problema.

En primer lugar, necesitamos, como en la búsqueda tabú, dos indicadores para detener el algoritmo: el número de iteraciones desde el comienzo y el número de iteraciones sin encontrar una mejora. Estos dos valores estarán limitados por constantes definidas en el código.

```
public class RecocidoSimuladoMochila extends RecocidoSimulado {
    protected int numIteracionesSinMejora = 0;
    protected int numIteraciones = 0;


    private final static int NUM_MAX_ITERACIONES = 100;
    private final static int NUM_MAX_ITERACIONES_SIN_MEJORAS = 30;

    // Otros métodos aquí
}
```

A continuación, es necesario implementar los distintos métodos abstractos de la clase madre. El primero consiste en actualizar la temperatura en cada iteración. Se aplica simplemente una multiplicación por 0.95, que permite disminuirla de forma gradual.

```
   @Override
   protected void ActualizarTemperatura() {
       temperatura *= 0.95;
   }
```

Para la inicialización de la temperatura, se parte de 5. Este valor debería adoptarse en función del problema, y el único método es empírico.

```
   @Override
   protected void InicializarTemperatura() {
```

```
        temperatura = 5;
    }
```

El tercer método es el que permite detener el algoritmo. Se comprueba entonces que ambos contadores sean inferiores a los valores máximos.

```
    @Override
    protected boolean CriterioParada() {
        return numIteraciones > NUM_MAX_ITERACIONES ||
numIteracionesSinMejora >
NUM_MAX_ITERACIONES_SIN_MEJORAS;
    }
```

El siguiente método es el más complejo de esta clase: permite saber si es necesario o no actualizar la solución actual. Para ello, consultamos en primer lugar si se trata de una solución que entrañe una pérdida de calidad. En caso afirmativo, se calcula la probabilidad de aceptarla gracias a la ley de Metropolis. A continuación, si se extrae un número aleatorio inferior a esta probabilidad (o si la solución propuesta es mejor), se actualiza la solución actual. Además, si es la mejor encontrada hasta el momento, se actualiza la variable `mejorSolucion` y se reinicia el número de iteraciones sin mejora.

```
    @Override
    protected void Actualizar(ISolucion solucion) {
        double probabilidad = 0.0;
        if (solucion.getValor() < solucionActual.getValor()) {
            probabilidad = Math.exp(-1 * (solucionActual.getValor() -
solucion.getValor()) / solucionActual.getValor() / temperatura);
        }
        if (solucion.getValor() > solucionActual.getValor()
|| ProblemaMochila.generador.nextDouble() < probabilidad) {
            solucionActual = solucion;
            if (solucion.getValor() > mejorSolucion.getValor()) {
                mejorSolucion = solucion;
                numIteracionesSinMejora = 0;
            }
        }
    }
```

El quinto método incrementa, simplemente, los dos contadores.

```
    @Override
    protected void Incrementar() {
        numIteracionesSinMejora++;
        numIteraciones++;
    }
```

El sexto y último método ejecuta la visualización de la mejor solución encontrada hasta el momento.

```
     @Override
```

```
    protected void DevolverResultado() {
        ihm.MostrarMensaje(mejorSolucion.toString());
    }
```

El recocido simulado está ahora implementado.

f. Optimización por enjambre de partículas

El último algoritmo para adaptar es la optimización por enjambre de partículas. En lugar de escoger la mejor solución de un vecindario, debemos hacer evolucionar todas nuestras soluciones con el paso del tiempo en función de la mejor solución encontrada hasta el momento y de la mejor solución en curso.

El criterio de parada es simplemente el número de iteraciones. Tenemos, por lo tanto, un atributo encargado de contar las iteraciones desde el inicio y una constante que fija el número de iteraciones que es preciso realizar. La base de nuestra clase **`EnjambreParticulasMochila`** es la siguiente:

```
import java.util.ArrayList;

public class EnjambreParticulasMochila extends EnjambreParticulas {
    protected int numIteraciones = 0;
    private final static int NUM_MAX_ITERACIONES = 200;

    // Otros métodos aquí
}
```

La clase abstracta `EnjambreParticulas` contiene cinco métodos abstractos que hay que implementar. El primero y más complejo consiste en actualizar las distintas soluciones. Para ello, se recorre la lista de toda la población de soluciones.

A cada solución potencial, si no se trata de la mejor encontrada hasta el momento, se le va a agregar un elemento de la mejor solución de la población actual y un elemento de la mejor encontrada desde el inicio. Los elementos echados a suerte se agregan solamente si no existen ya en la mochila.

Tras agregar estos elementos, la mochila puede tener un peso importante. Se eliminan, a continuación, aleatoriamente las cajas hasta alcanzar el límite del peso. Por último, si es posible, se completa la mochila.

En efecto, si estamos en 21 kg y se sustrae un elemento de 7 kg al azar, se pasa a tener 14 kg. Estamos bastante por debajo del límite de 20 kg, de modo que podemos aumentar el valor de la mochila agregando hasta 6 kg de elementos.

```
    @Override
    protected void ActualizarSoluciones() {
        for (ISolucion sol : soluciones) {
            SolucionMochila solucion = (SolucionMochila) sol;
            if (!solucion.equals(mejorSolucion)) {
               // Agrega un elemento de los mejores
               solucion = AgregarElemento (solucion, mejorSolucion);
                  solucion = AgregarElemento (solucion, mejorActual);
               // Disminución del peso si hace falta
               while (solucion.getPeso() >
((ProblemaMochila)problema).pesoMax) {
```

```
                    indice =
ProblemaMochila.generador.nextInt(solucion.contenido.size());
                    solucion.contenido.remove(indice);
                }
                // Por último, se completa
                solucion = Completar(solucion);
            }
        }
    }
    protected SolucionMochila AgregarElemento(SolucionMochila Solucion,
ISolucion SolucionOrigen) {
        int índice = ProblemaMochila.generador.nextInt(((SolucionMochila)
SolucionOrigen).contenido.size());
        Caja b = ((SolucionMochila)SolucionOrigen).contenido.get(indice);
        if (!Solucion.contenido.contains(b)) {
            Solucion.contenido.add(b);
        }
        return Solucion;
    }

    protected SolucionMochila Completar(SolucionMochila Solucion) {
                double espacioDispo =
((ProblemaMochila)problema).pesoMax - solucion.getPeso();
                ArrayList<Caja> cajasPosibles =
((ProblemaMochila)problema).Cajas();
                cajasPosibles.removeAll(solucion.contenido);
((ProblemaMochila)problema).EliminarDemasiadoPesadas(cajasPosibles,
espacioDispo);
                Cajas b;
                int indice;
                while (!cajasPosibles.isEmpty()) {
                    indice =
ProblemaMochila.generador.nextInt(cajasPosibles.size());
                    b = cajasPosibles.get(indice);
                    solucion.contenido.add(b);
                    cajasPosibles.remove(b);
                    espacioDispo =
((ProblemaMochila)problema).pesoMax - solucion.getPeso();
((ProblemaMochila)problema).EliminarDemasiadoPesadas(cajasPosibles,
espacioDispo);
                }
                return solucion;

    }
```

El segundo método debe actualizar las mejores soluciones (la mejor de la población en curso y potencialmente la mejor encontrada desde el inicio). Debemos para ello recorrer la población actual y potencialmente modificar `mejorSolucion`.

```
    @Override
    protected void ActualizarVariables() {
        mejorActual = soluciones.get(0);
        for (ISolucion sol : soluciones) {
```

```
            if (sol.getValor() > mejorActual.getValor()) {
                mejorActual = sol;
            }
        }
        if (mejorActual.getValor() >
mejorSolucion.getValor()) {
            mejorSolucion = mejorActual;
        }
    }
```

El tercer método permite comprobar que no se ha alcanzado el criterio de parada. Basta con verificar el número de iteraciones.

```
    @Override
    protected boolean CriterioParada() {
        return numIteraciones > NUM_MAX_ITERACIONES;
    }
```

El cuarto método incrementa simplemente el número de iteraciones.

```
    @Override
    protected void Incrementar() {
        numIteraciones++;
    }
```

El último método permite mostrar la mejor solución encontrada hasta el momento.

```
    @Override
    protected void DevolverResultado() {
        ihm.MostrarMensaje(mejorSolucion.toString());
    }
```

Todos los algoritmos están ahora adaptados al problema de la mochila.

g. Programa principal

Esta clase es específica de una aplicación de consola. Se trata del programa principal, que contiene el main. La clase **`Mochila`** implementa la interfaz **`IHM`** para permitir la visualización. Se trata simplemente de escribir los resultados por consola.

```
public class Mochila implements IHM {
    @Override
    public void MostrarMensaje(String msg) {
        System.out.println(msg);
    }

    // Otros métodos aquí (incluido el main)
}
```

Creamos, a continuación, un método que permita ejecutar sobre un problema pasado como parámetro los cinco algoritmos seguidos, con algunas visualizaciones para aclararse.

```
private void EjecutarAlgoritmos(IProblema pb) {
    Algoritmo algo;

    System.out.println("Algoritmo voraz");
    algo = new AlgoritmoVorazMochila();
    algo.Resolver(pb, this);
    System.out.println();

    System.out.println("Descenso por gradiente");
    algo = new DescensoGradienteMochila();
    algo.Resolver(pb, this);
    System.out.println();

    System.out.println("Búsqueda tabú");
    algo = new BusquedaTabuMochila();
    algo.Resolver(pb, this);
    System.out.println();

    System.out.println("Recocido simulado");
    algo = new RecocidoSimuladoMochila();
    algo.Resolver(pb, this);
    System.out.println();

    System.out.println("Enjambre de partículas");
    algo = new EnjambreParticulasMochila();
    algo.Resolver(pb, this);
    System.out.println();
}
```

Podemos codificar un método `Ejecutar`, que permite ejecutar los distintos algoritmos sobre el problema de la mochila simple (el que se ha expuesto como ejemplo en este capítulo), y a continuación sobre un problema aleatorio más complejo, con 100 cajas de un valor máximo de 20, para una mochila que puede contener hasta 30 kg.

```
private void Ejecutar() {
    System.out.println("Metaheurísticos de optimización");
    ProblemaMochila pb = new ProblemaMochila();
    EjecutarAlgoritmos(pb);
    System.out.println("*******************************\n");
    pb = new ProblemaMochila(100, 30, 20);
    EjecutarAlgoritmos(pb);
}
```

El programa principal consiste simplemente en crear una instancia de la clase `Mochila` e invocar su método `Ejecutar`.

```
    public static void main(String[] args) {
        Mochila app = new Mochila();
        app.Ejecutar();
    }
```

El programa es completamente operacional.

4. Resultados obtenidos

He aquí un ejemplo de la salida que se obtiene primero con el primer problema, mostrando el valor y el peso de la mochila más su contenido, y a continuación con el segundo problema.

Podemos ver que todas las soluciones son idénticas y se corresponden con el óptimo para la primera mochila.

Para el problema aleatorio (que cambia con cada ejecución), dos algoritmos devuelven el mejor resultado: el descenso por gradiente y la búsqueda tabú.

```
Metaheurísticos de optimización
Algoritmo voraz
Valor: 54.0 - Peso: 19.0 - A (4.0, 15.0) - D (3.0, 10.0) - K
(4.0, 10.0) - I (5.0, 12.0) - L (3.0, 7.0)

Dencenso por gradiente
Valor: 54.0 - Peso: 19.0 - I (5.0, 12.0) - L (3.0, 7.0) - A
(4.0, 15.0) - K (4.0, 10.0) - D (3.0, 10.0)

Búsqueda tabú
Valor: 54.0 - Peso: 19.0 - D (3.0, 10.0) - L (3.0, 7.0) - K
(4.0, 10.0) - I (5.0, 12.0) - A (4.0, 15.0)

Recocido simulado
Valor: 54.0 - Peso: 19.0 - A (4.0, 15.0) - L (3.0, 7.0) - D
(3.0, 10.0) - K (4.0, 10.0) - I (5.0, 12.0)

Enjambre de partículas
Valor: 54.0 - Peso: 19.0 - I (5.0, 12.0) - A (4.0, 15.0) - D
(3.0, 10.0) - K (4.0, 10.0) - L (3.0, 7.0)

**************************************

Algoritmo voraz
Valor: 203.17144468574156 - Peso: 29.258142969507645 - 10
(0.1686748220407619, 9.263750145106899) - 28 (0.613074117508603,
16.070232925823188) - 80 (0.777151588932854, 18.166105374376006) -
55 (0.42883055629077904, 7.58514214367745) - 92 (1.134213817650217,
16.36599833009916) - 16 (0.8409597957424475, 10.138568229837869) -
76 (1.552223740043025, 17.556632151284866) - 33 (1.849713045978345,
19.293071624661565) - 15 (1.5042373440291112, 14.941501644970554) -
93 (3.8628789333855273, 19.348870841862876) - 89 (0.5739693099564236,
2.771662125433718) - 95 (2.384800597917761, 9.306829246086426) -
88 (4.153729512718253, 13.708409186566682) - 0 (5.4617144944335365,
17.26893639263276) - 21 (3.9519712928799997, 11.38573432332156)
```

Descenso por gradiente
Valor: 203.37666349536724 - Peso: 29.602659687033164 - 96
(4.958844474852119, 15.30280646792865) - 92 (1.134213817650217,
16.36599833009916) - 10 (0.1686748220407619, 9.263750145106899) -
89 (0.5739693099564236, 2.771662125433718) - 76 (1.552223740043025,
17.556632151284866) - 16 (0.8409597957424475, 10.138568229837869) -
14 (0.8473867371069355, 2.171348734329721) - 28 (0.613074117508603,
16.070232925823188) - 15 (1.5042373440291112, 14.941501644970554) -
33 (1.849713045978345, 19.293071624661565) - 55 (0.42883055629077904,
7.58514214367745) - 95 (2.384800597917761, 9.306829246086426) -
80 (0.777151588932854, 18.166105374376006) - 93 (3.8628789333855273,
19.348870841862876) - 21 (3.9519712928799997, 11.38573432332156) -
88 (4.153729512718253, 13.708409186566682)

Búsqueda tabú
Valor: 203.37666349536724 - Peso: 29.602659687033167 - 14
(0.8473867371069355, 2.171348734329721) - 93 (3.8628789333855273,
19.348870841862876) - 33 (1.849713045978345, 19.293071624661565) -
80 (0.777151588932854, 18.166105374376006) - 89 (0.5739693099564236,
2.771662125433718) - 76 (1.552223740043025, 17.556632151284866) -
15 (1.5042373440291112, 14.941501644970554) - 92 (1.134213817650217,
16.36599833009916) - 16 (0.8409597957424475, 10.138568229837869) -
88 (4.153729512718253, 13.708409186566682) - 21 (3.9519712928799997,
11.38573432332156) - 55 (0.42883055629077904, 7.58514214367745) -
28 (0.613074117508603, 16.070232925823188) - 95 (2.384800597917761,
9.306829246086426) - 10 (0.1686748220407619, 9.263750145106899) -
96 (4.958844474852119, 15.30280646792865)

Recocido simulado
Valor: 198.0891227327012 - Peso: 28.9111185916602 - 28
(0.613074117508603, 16.070232925823188) - 16 (0.8409597957424475,
10.138568229837869) - 10 (0.1686748220407619, 9.263750145106899) -
80 (0.777151588932854, 18.166105374376006) - 55 (0.42883055629077904,
7.58514214367745) - 33 (1.849713045978345, 19.293071624661565) -
14 (0.8473867371069355, 2.171348734329721) - 0 (5.4617144944335365,
17.26893639263276) - 15 (1.5042373440291112, 14.941501644970554) -
92 (1.134213817650217, 16.36599833009916) - 76 (1.552223740043025,
17.556632151284866) - 93 (3.8628789333855273, 19.348870841862876) -
91 (1.9524452157917482, 2.5376663545894274) - 96 (4.958844474852119,
15.30280646792865) - 95 (2.384800597917761, 9.306829246086426) -
89 (0.5739693099564236, 2.771662125433718)

Enjambre de partículas
Valor: 198.0723383111397 - Peso: 29.580720732778005 - 76
(1.552223740043025, 17.556632151284866) - 15 (1.5042373440291112,
4.941501644970554) - 33 (1.849713045978345, 19.293071624661565) -
14 (0.8473867371069355, 2.171348734329721) - 80 (0.777151588932854,
18.166105374376006) - 16 (0.8409597957424475, 10.138568229837869) -
28 (0.613074117508603, 16.070232925823188) - 88 (4.153729512718253,
13.708409186566682) - 58 (6.936431597685213, 18.846549252433263) -
93 (3.8628789333855273, 19.348870841862876) - 89 (0.5739693099564236,
2.771662125433718) - 91 (1.9524452157917482, 2.5376663545894274) -
10 (0.1686748220407619, 9.263750145106899) - 92 (1.134213817650217,

```
16.36599833009916) - 95 (2.384800597917761, 9.306829246086426) -
55 (0.42883055629077904, 7.58514214367745)
```

Cuando se abordan los metaheurísticos, siendo en su mayor parte aleatorios, como el segundo problema que queremos resolver, es importante analizar un poco los resultados obtenidos.

Para el primer problema y sobre todas las pruebas realizadas, los cinco algoritmos han encontrado siempre el óptimo de 54. Este problema es bastante sencillo, de modo que no es muy significativo.

A continuación, se han realizado 50 pruebas sobre el segundo problema. En cuatro ocasiones (que equivalen a un 8 % de los casos), todos los algoritmos han encontrado el mismo óptimo (que podemos suponer global, incluso aunque nada nos permite afirmarlo).

En los demás casos, uno o varios algoritmos se desmarcan con resultados mejores. En total, el **algoritmo voraz** ha encontrado el mejor valor en el 50 % de los casos.

En efecto, este puede fallar bastante a menudo. Imaginemos una mochila que ya está completa hasta los 16 kg con un peso máximo de 20 kg. Si tenemos la opción entre dos cajas de 3 kg y 4 kg, con respectivos valores de 12 y de 13, el algoritmo voraz cargará la primera caja (con una relación de 12/3 = 4 puntos de valor por kg) en lugar de la segunda, que tiene una relación de 13/4 = 3.25. La mochila está ahora cargada hasta los 19 kg, y el último kilogramo libre se "pierde". La segunda caja, si bien es menos prometedora, habría aprovechado este espacio y aportado un valor superior.

El algoritmo voraz ha fallado aquí al menos en la mitad de los casos.

El algoritmo de **descenso por gradiente** encuentra el óptimo solamente en el 30 % de los casos. En efecto, cada vez se utiliza un único punto de partida, y el algoritmo se detiene a menudo en óptimos locales. No es demasiado eficaz si no se utilizan varios inicios.

Los otros tres algoritmos (**búsqueda tabú**, **recocido simulado** y **optimización por enjambre de partículas**) tienen una tasa de éxito mayor, con tasas próximas al 60 %, respectivamente. Si bien son difíciles de articular, son mejores que los otros dos algoritmos. En efecto, están diseñados para resolver los defectos del descenso por gradiente, intentando evitar al máximo quedar bloqueado en un óptimo local.

En función de los problemas, y jugando con los distintos parámetros, uno de estos algoritmos puede superar a los demás, pero en el conjunto de problemas existentes ninguno es mejor. En inglés se denomina a este fenómeno el "No Free Lunch", que indica que no existe, jamás, un algoritmo milagroso que permita resolver todos los problemas mejor que los demás.

Resumen

Este capítulo ha presentado cinco algoritmos clasificados como metaheurísticos. Todos ellos tienen como objetivo encontrar el óptimo de un problema. Si bien es preferible encontrar el óptimo global, cuando no existe ningún método matemático para obtenerlo y probar todas las soluciones sería demasiado largo, son la alternativa ideal, ya que permiten encontrar buenos óptimos locales e incluso el óptimo global.

El primero es el **algoritmo voraz**. Consiste en construir de manera incremental una única solución, siguiendo lo que parece más adecuado para alcanzar un óptimo.

El **descenso por gradiente** parte de una solución aleatoria. En cada iteración, se analiza el vecindario de esta, y se sigue la dirección más prometedora. Cuando no se produce ninguna mejora en el vecindario, se ha alcanzado el óptimo local. Este algoritmo es simple y funciona bien, aunque a menudo se queda bloqueado en los óptimos locales y no encuentra necesariamente el óptimo global.

La **búsqueda tabú** se ha creado para permitir mejorar el descenso por gradiente. En efecto, en lugar de desplazarse de posición en posición siguiendo una progresión, se desplaza hasta el mejor vecino accesible, sea mejor o no que nosotros. Así, evita quedarse en óptimos locales. Además, para evitar encontrar regularmente las mismas posiciones, se registran también las últimas posiciones recorridas, que se anotan como tabú y no pueden utilizarse. Esta búsqueda permite recorrer varios óptimos locales para maximizar la probabilidad de encontrar un óptimo global.

El **recocido simulado** se inspira en un fenómeno físico utilizado en metalurgia. Cuando se ejecuta el algoritmo, la probabilidad de aceptar desplazarse hacia una solución vecina menos adaptada es grande. Conforme pasa el tiempo, con el sesgo de la temperatura, esta probabilidad disminuye hasta hacerse nula. Se recorren en primer lugar varias zonas del conjunto de búsqueda, y a continuación se va poco a poco estabilizando hacia una zona que parece más interesante, hasta encontrar un óptimo, posiblemente global.

El último metaheurístico es la **optimización por enjambre de partículas**. En lugar de hacer evolucionar una única solución mirando en cada iteración todos sus vecinos, se hace evolucionar en el entorno una población de soluciones. Con cada iteración, cada solución se va a adaptar para dirigirse hacia las zonas que presentan un mayor interés. De este modo, se recorre un espacio mayor, y la probabilidad de encontrar el óptimo global aumenta.

En el problema aleatorio de la mochila, los tres últimos metaheurísticos son equivalentes y producen mejores resultados. La dificultad radica, por el contrario, en seleccionar los parámetros adecuados para cada algoritmo.

Presentación del capítulo

Este capítulo presenta los sistemas multiagentes, que permiten responder a una gran variedad de problemáticas. En estos sistemas, varios agentes, con comportamientos individuales simples, trabajarán de manera conjunta para resolver problemas mucho más complejos.

Estos algoritmos están inspirados en observaciones realizadas en biología (y particularmente en etología). Existen colonias de insectos capaces de resolver problemas complejos (como crear un hormiguero), mientras que cada insecto individualmente no posee grandes capacidades. Este capítulo empieza presentando las principales características de estos insectos sociales.

A continuación se presentan las características mínimas de un sistema para que pueda considerarse como formado por multiagentes, así como las diversas categorías de agentes.

Algunos algoritmos son particulares y constituyen un campo de estudio en sí mismos. El capítulo los presenta a continuación: algoritmos de manadas, colonias de hormigas, sistemas inmunitarios artificiales y autómatas celulares.

Se proporcionan también varias implementaciones en Java. Por último, este capítulo termina con un resumen.

Origen biológico

Los insectos han interesado a los investigadores en biología y etología (la ciencia que estudia el comportamiento) desde hace mucho tiempo. Incluso aunque todavía no se conocen todos los comportamientos sociales, sí se han actualizado los grandes principios.

La mayoría de los insectos (cerca del 90 % de las especies) son solitarios. Cada insecto vive por su lado, con pocos o incluso sin vínculos con sus vecinos. Es el caso, por ejemplo, de la mayoría de las arañas, mosquitos, moscas... Los contactos se limitan a la búsqueda de comida (por ejemplo, en el caso de las moscas), a las zonas de vida (como las larvas de los mosquitos, presentes en muchas aguas estancadas) o a los periodos de reproducción.

Existen, también, insectos sociales, que van desde sociedades muy simples hasta sociedades muy complejas y organizadas. Los que presentan la mayor organización son los llamados **insectos eusociales** y tienen las siguientes características:

- La población se divide en **castas**; cada casta tiene un rol preciso.
- La reproducción está limitada a una casta particular.
- Las larvas y los jóvenes se educan juntos en el seno de la colonia.

Los insectos eusociales no representan más que el 2 % de las especies existentes, ¡aunque en masa representan el 75 % del conjunto de los insectos! Esto pone de relieve que estas sociedades permiten mantener a un gran número de individuos.

Las tres especies eusociales más conocidas son las abejas, las termitas y las hormigas.

1. Las abejas y la danza

La mayoría de las especies de abejas son solitarias. Sin embargo, la especie *Apis mellifera* (la abeja de las colmenas que produce la miel) es una especie eusocial.

Cada colmena es una sociedad completa. Encontramos una reina, encargada de poner los huevos (se fertiliza antes de fundar su colonia y vive hasta cuatro años), obreras (hembras estériles) y algunos machos (llamados zánganos), que no sirven más que para emparejarse con las futuras reinas. Mueren justo después de la reproducción.

Las obreras tienen distintos roles: el trabajo en la cresa (donde se encuentran las larvas), el mantenimiento de la colmena, la búsqueda de comida, su cosecha, la defensa de la colmena...

El fenómeno más fascinante relacionado con las abejas es su **comunicación**. Utilizan **feromonas** (sustancias químicas olorosas) para ciertos comportamientos (como el aviso para las abejas que se habían alejado una vez pasado el peligro que había amenazado a la colmena). Pero, sobre todo, utilizan varias **danzas**.

La misión de las exploradoras es encontrar nuevas fuentes de alimento (de polen), abastecimiento de agua o zonas de cosecha de resina, incluso nuevas zonas para crear o desplazar la colmena. Cuando vuelven al nido, realizarán una danza para indicar a las demás dónde ir y qué pueden esperar encontrar.

Si la fuente está próxima, practican una danza en redondo, girando en un sentido y después en el otro. Las seguidoras entrarán en contacto con la abeja que danza para conocer qué ha encontrado mediante el gusto y el olor. Si la fuente está lejos, a partir de una decena de metros y hasta varios kilómetros, la exploradora realizará una danza en 8: el eje central del 8 respecto a la vertical indica la dirección de la fuente respecto al suelo, el tamaño y la frecuencia del movimiento indican la distancia y el interés de la fuente. Por último, la palpación permite a las seguidoras descubrir la calidad y la sustancia encontrada.

Esta danza es muy precisa y permite localizar fuentes de alimento con un error de menos de 3° respecto a la dirección que se debe tomar. Además, se adaptan al desplazamiento del sol en el cielo y a la presencia de viento, que aumenta o disminuye el tiempo de trayecto.

Gracias a esta comunicación, cada seguidora sabe exactamente a dónde ir para encontrar alimento.

Cada abeja posee, por lo tanto, medios limitados y un rol muy simple, pero la presencia de distintos roles y la comunicación establecida entre los individuos permite a las colmenas sobrevivir y extenderse.

2. Las termitas y la ingeniería civil

Las termitas son también animales eusociales. Viven en inmensas colonias de varios miles de individuos, con la presencia de castas.

En el centro de la colonia viven el rey y la reina, e incluso reinas secundarias. Se rodean de obreras, de soldados (que tienen mandíbulas fuertes o, en el caso de ciertas especies, la capacidad de arrojar productos químicos), de jóvenes, de larvas... Las obreras se ocupan, entre otras tareas, del alimento, del cuidado de las larvas y de la construcción del **termitero**.

Este presenta varias características impresionantes: el interior de un termitero tiene una temperatura y una humedad constantes, a pesar de las temperaturas extremas y variables presentes en África. Este control del clima en el interior se debe a su estructura muy particular, con pozos, una chimenea, pilares, un nido central sobreelevado... Existe una ventilación pasiva. Además, los termiteros pueden alcanzar hasta 8 metros de altura, una circunferencia en la base de hasta 30 metros.

Existen estudios que han investigado cómo son capaces las termitas de construir tales estructuras, denominadas **termiteros catedrales** (por su similitud con nuestras catedrales). En realidad, las termitas no tienen consciencia de la estructura global y los planos que deben seguir. Cada termita construye una bolita de tierra y la deposita en otro lugar, con una probabilidad proporcional a la cantidad de bolitas ya presentes.

La estructura completa es, por lo tanto, **emergente**. Varios agentes muy simples (las termitas) son capaces de resolver problemas complejos (mantener un nido a una temperatura y una humedad constantes).

Cabe destacar que los arquitectos se han inspirado en planos de termiteros para construir edificios que luego necesitan poca o prácticamente ninguna energía para mantener una temperatura agradable.

3. Las hormigas y la optimización de caminos

Las hormigas son, también, insectos eusociales. Sus colonias pueden contener hasta un millón de individuos. Una o varias reinas ponen huevos. Los machos, en sí, no sirven más que para la reproducción y mueren después. Las demás hormigas, hembras estériles, son obreras.

Las obreras tienen varios roles posibles: se ocupan de las larvas, del hormiguero, de buscar alimento, de su recogida, de la defensa de la colonia (soldados). Algunas especies de hormigas tienen roles todavía más marcados: algunas atacarán a otras especies para robar sus larvas y convertirlas en esclavas, otras son capaces de organizar y mantener enjambres de pulgones para obtener su néctar.

Todas estas colonias sobreviven gracias a la comunicación entre sus miembros. Esta se realiza mediante las **feromonas**, captadas a través de las antenas. La principal comunicación es la que permite indicar las fuentes de alimento. Las hormigas exploradoras se desplazan aleatoriamente. Si una de ellas encuentra alimento, vuelve al nido con él. A lo largo del camino de retorno, va depositando feromonas, cuya intensidad depende del alimento y de la longitud del camino.

Las demás exploradoras, cuando encuentran una pista de feromonas, tienen tendencia a seguirla. La probabilidad de seguirla depende, en efecto, de la cantidad de feromonas depositada. Además, las feromonas se evaporan de manera natural.

Gracias a estas reglas sencillas, las hormigas son capaces de determinar los caminos más cortos entre el nido y una fuente de alimento. En efecto, los caminos más largos se utilizarán menos que los caminos más cortos en el mismo tiempo, lo cual refuerza estos últimos.

La comunicación gracias a modificaciones del entorno (trazas de feromonas) se denomina **estigmergia**.

4. Inteligencia social

Todas estas especies demuestran **inteligencia social**. Cada individuo no tiene consciencia de todo lo que acontece en la colonia, pero cumple el trabajo para el que está destinado. En ausencia de jerarquía, emergen comportamientos más complejos, como por ejemplo la construcción de termiteros. Cuanto más numerosos sean los individuos y más importantes sus vínculos, más impresionante será el resultado.

Esta inteligencia social, que permite resolver problemas complejos a partir de individuos con comportamientos muy simples, ha dado lugar al nacimiento de los sistemas multiagentes.

Sistemas multiagentes

Todas las técnicas clasificadas como **sistemas multiagentes** tienen como objetivo implementar esta inteligencia social, que se denomina en informática **inteligencia distribuida**. Para ello, encontramos:

- Un entorno.
- Objetos fijos o no, que son los obstáculos o los puntos de interés.
- Agentes, como comportamientos simples.

Realmente, el objetivo del algoritmo no se codifica jamás, y la solución va a **emerger** de la interacción de todos los elementos entre sí.

1. El entorno

Los objetos y los agentes se encuentran en un **entorno**. Este puede ser más o menos complejo: puede tratarse de un espacio delimitado (como un cobertizo o un bosque), de un grafo o incluso de un espacio puramente virtual.

El entorno se corresponde, por lo tanto, principalmente con el problema que queremos resolver.

Este entorno debe poder evolucionar con el paso del tiempo: los agentes pueden desplazarse, y los objetos, modificarse.

2. Los objetos

El entorno posee **objetos** con los que los agentes pueden interactuar. Estos pueden corresponderse con fuentes de alimento, bloques de construcción, ciudades que visitar, obstáculos...

Nada impone que los objetos sean o no transportables, que sean temporales o permanentes. En este caso, hay que adaptarlos al problema.

En ciertos sistemas no existe ningún objeto, sino solamente los agentes. Su presencia es, de este modo, opcional.

3. Los agentes

Los **agentes** van a vivir en el entorno e interactuar con los objetos y los demás agentes. Es necesario definir, además del comportamiento de los agentes, las relaciones que estos tienen entre sí. Puede tratarse de una relación jerárquica o de enlaces de comunicación.

Además, es importante que cada agente posea un conjunto de operaciones posibles, tanto sobre los objetos (cómo cogerlos, transportarlos o utilizarlos) como sobre los demás agentes (pueden interactuar directamente, por ejemplo, intercambiando objetos).

Mediante estos intercambios, el entorno se modificará, lo que implica una modificación de la acción de los agentes, hasta descubrir la solución (o una solución considerada como suficientemente buena).

Clasificación de los agentes

Los agentes pueden ser de tipos muy diferentes en función de ciertas características.

1. Percepción del mundo

La primera diferencia se produce en la precepción del mundo que tendrán los agentes. Podrán tener una visión de conjunto de todo el mundo (**percepción total**) o únicamente de lo que se encuentra a su alrededor (**percepción localizada**). Además, pueden tener una **visión simbólica** o únicamente aquella **derivada de la percepción**.

Por ejemplo, si tomamos robots que se desplazan por la planta de un edificio, pueden conocer lo que ven (la habitación que los rodea, percepción localizada) o tener un mapa registrado previamente de toda la planta con su posición encima (por ejemplo, mediante un GPS, percepción total).

Además, cuando poseen una cámara, pueden tener, en el caso de una visión simbólica, algoritmos de reconocimiento de imágenes que les permitan reconocer ciertos objetos (pomos de puertas, botones, destinos...) o, en el caso de una visión resultado de la percepción, interactuar en función de las imágenes brutas (tal y como se obtienen por la cámara).

Los investigadores de sistemas multiagentes tienen preferencia por percepciones localizadas, y están muy repartidos entre un enfoque simbólico y un enfoque puramente basado en las percepciones.

2. Toma de decisiones

Los agentes deben escoger qué hacer en todo momento. Pueden tener un plan determinado de avance, lo cual se aconseja en pocas ocasiones, o bien reaccionar en función de sus percepciones.

Poseen, por lo tanto, una o varias reglas que aplicar. En este caso, puede tratarse de un conjunto complejo compuesto por un sistema experto, del uso de lógica fluida, de estados de transición... La elección dependerá del objetivo que se persiga.

Sin embargo, la complejidad del problema que se ha de resolver no tiene relación con la complejidad de las decisiones de los agentes. En efecto, con algunas reglas muy simples es posible resolver problemas muy complejos. Se trata, sobre todo, de encontrar el sistema adecuado.

Un individuo que reacciona directamente a sus percepciones sería un **agente reactivo**. Por el contrario, si reacciona tras reflexionar una decisión en función de conocimientos, se trata de un **agente cognitivo**.

Si retomamos el ejemplo de los robots que se desplazan en un entorno, un robot que evite los obstáculos yendo en la dirección opuesta a la detección de un elemento mediante un sensor de distancia, por ejemplo, sería un agente puramente reactivo. Si escoge su trayecto para buscar un objeto que sabe dónde encontrar (por ejemplo, para encontrar un objeto pedido por un humano), entonces se trata de un agente cognitivo que escogerá la mejor estrategia y la mejor trayectoria.

En la mayoría de los casos, se agrega también un **aspecto estocástico**: la presencia de algo de aleatoriedad va a permitir a menudo que el sistema sea más fluido y más flexible.

3. Cooperación y comunicación

Los agentes no están solos. Es importante, por ello, saber cómo van a cooperar o comunicarse.

Podemos imaginar sistemas puramente reactivos que no realizan cambios en su entorno. Se observan comportamientos de grupo por **emergencia**.

Los individuos pueden, también, **comunicarse directamente**, mediante mensajes de radio, sonidos o mensajes visuales (como luces o señales). Esta comunicación puede tener una duración limitada o no.

Por último pueden, como las hormigas, dejar trazas en el entorno y utilizar la **estigmergia** para comunicarse de forma asíncrona. Un ejemplo de estigmergia es cómo Pulgarcito dejaba piedrecitas blancas para encontrar su camino: modificando su entorno, sus hermanos y él podían volver a casa.

Muchos animales utilizan la estigmergia, y no solo en el caso de especies eusociales. En efecto, los animales que marcan su territorio (como los osos con los arañazos en los árboles o los caballos que hacen montones de estiércol en su territorio) utilizan la estigmergia para indicar a los demás individuos que no deben aproximarse y que este territorio ya está ocupado.

Por último, en ciertos casos, los agentes pueden llegar a **negociar** entre sí para encontrar un consenso que será la solución adoptada. En este caso, conviene prever distintos comportamientos y la posibilidad de realizar estas negociaciones.

4. Capacidad del agente

El último aspecto es relativo a la **capacidad** del agente. En efecto, pueden tener capacidades muy limitadas o bien una gran gama de acciones posibles.

Los agentes reactivos no tienen, por lo general, más que unas pocas acciones posibles (por ejemplo, girar a la derecha o a la izquierda). Los agentes cognitivos tienen, a menudo, una gama de acciones más extensa (por ejemplo, escoger un camino, evitar un obstáculo...).

Además, los agentes pueden, todos ellos, tener las mismas capacidades o ser específicos a una tarea particular, organizados en castas, como ocurre con los insectos.

Por último, en función de los sistemas implementados, los distintos agentes pueden aprender con el paso del tiempo o bien tener conocimientos fijos. Cuando se agrega **aprendizaje**, es preciso determinar el algoritmo subyacente (redes neuronales, algoritmo genético...).

Existen, por lo tanto, muchos tipos de agentes y cada problema tendrá que escoger los mejores parámetros para optimizar el resultado obtenido. Sin embargo, para un mismo problema, existen a menudo varias soluciones correctas, lo que permite una flexibilidad mayor en su implementación.

Principales algoritmos

Existen algunos algoritmos particulares más conocidos que otros, que presentan entornos, objetos y agentes definidos. Aquí se presentan cuatro: los algoritmos de manadas, la optimización por colonias de hormigas, los sistemas inmunitarios artificiales y los autómatas celulares.

En el capítulo dedicado a los metaheurísticos, el algoritmo de optimización por enjambre de partículas podía haberse visto como un sistema multiagente en el que cada solución tenía una visión global de todas las demás soluciones y una memoria (la mejor solución encontrada hasta el momento). Se desplazaban en el espacio de soluciones que servía como entorno. No había objetos. Sin embargo, esta técnica se aleja de la noción de agentes reactivos.

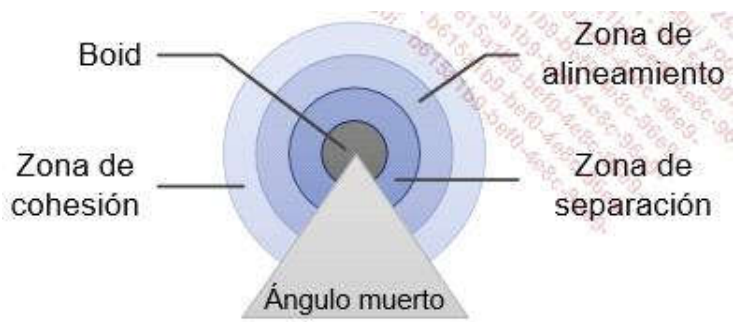
1. Algoritmos de manadas

Partiendo de algunas reglas sencillas, es posible simular los **comportamientos de manadas** o de grupos. Craig Reynolds creó, así, en 1986 los boids, criaturas artificiales que evolucionan en grupo.

Para ello, las criaturas poseen tres comportamientos, vinculados a la presencia de otros individuos en su proximidad:

- Un individuo muy cercano va a provocar un comportamiento para evitarse mutuamente (para evitar invadir a otro individuo): es el **comportamiento de separación**.
- Un individuo próximo modifica la dirección de la criatura, que tiene tendencia a alinearse en la dirección de su vecino: es el **comportamiento de alineamiento**.
- Un individuo a una distancia media va a provocar un acercamiento. En efecto, si una criatura ve a otra, irá hacia ella: es el **comportamiento de cohesión**.

También es posible agregar un "ángulo muerto" tras el boid simulando el hecho de que no puede ver detrás de sí.



En función de los parámetros, en particular de las distancias configuradas, podemos observar individuos completamente aislados o, por el contrario, individuos que se desplazan en manada y que pueden encontrarse tras un objeto, a la manera de los bancos de peces o las nubes de pájaros o de insectos.

Se observa, por tanto, una estructura emergente a partir de algunas reglas muy simples. Además, las trayectorias parecen aleatorias: en realidad el conjunto se convierte en un sistema complejo, si bien es completamente determinista, en el que la mínima modificación del espacio o de la situación inicial produce movimientos muy diferentes.

Podemos aprovechar estos boids para representar tropas en movimiento, en particular en películas y en videojuegos. El efecto producido es muy realista (y también bastante hipnótico en simulación).

2. Optimización por colonia de hormigas

La **optimización por colonia de hormigas** está directamente inspirada en el funcionamiento de las hormigas exploradoras. El objetivo consiste en encontrar una solución óptima gracias a la **estigmergia**. Esta técnica fue creada por Marco Dorigo en 1992.

Al principio, el entorno está virgen. Las hormigas virtuales van a recorrer el espacio aleatoriamente, hasta encontrar una solución. La hormiga va a volver a su punto de partida depositando **feromonas**. Los demás agentes van a verse influenciados por estas feromonas y van a tener cierta tendencia a seguir el mismo camino.

Tras varias iteraciones, todas las hormigas seguirán el mismo camino (que representa la misma solución), y entonces el algoritmo habrá convergido.

El entorno debe representar, bajo el aspecto de un grafo o de un mapa, el conjunto de soluciones. También es posible resolver problemas de búsqueda de rutas (como el A*).

El pseudocódigo es el siguiente:

```
Inicializar el entorno
Mientras (criterio de parada no alcanzado)
   Para cada hormiga
          Si (objetivo no alcanzado)
                 Desplazarse aleatoriamente (siguiendo las pistas)
          Si no
                 Volver al nido dejando feromonas
          Fin Si
   Fin Para
   Actualizar las trazas de feromonas
Fin Mientras
```

La probabilidad de seguir una dirección depende de varios criterios:

- Las posibles direcciones.
- La dirección de la que viene la hormiga (para que no dé media vuelta).
- Las pistas de feromonas a su alrededor.
- Otros metaheurísticos (para favorecer, por ejemplo, una dirección de búsqueda).

En efecto, conviene que la probabilidad de seguir una pista aumente con la cantidad de feromonas, sin ser nunca 1 para no imponer este camino.

Las feromonas depositadas deben, en sí, ser proporcionales a la calidad de la solución o a su longitud. Pueden, también, depositarse de manera constante para ciertos problemas.

Estas deben evaporarse. Para ello, con el paso del tiempo, podemos multiplicar la cantidad por una **tasa de evaporación** inferior a 1 o eliminar cierta cantidad de feromonas.

Las principales dificultades aparecen en la configuración de las probabilidades y de la tasa de evaporación. En función de estas tasas, las hormigas podrán converger demasiado rápido hacia una solución que no sea la óptima o por el contrario no llegar jamás a converger hacia una solución buena.

3. Sistemas inmunitarios artificiales

Estos **sistemas inmunitarios artificiales** se inspiran en los sistemas naturales de los animales vertebrados (como el nuestro). En efecto, varias células colaboran para determinar los elementos benignos y malignos, y atacar aquello que se considere extraño (virus, bacterias, hongos o incluso veneno).

Los sistemas inmunitarios artificiales hacen evolucionar, en un entorno determinado, diferentes agentes de defensa. Cada uno conoce un conjunto de "**amenazas**", que sabe detectar (y combatir si fuera necesario). Estos agentes pueden crearse aleatoriamente al inicio.

Con el paso del tiempo, si un agente encuentra una amenaza identificada, va a atacarla y avisará a los demás agentes cercanos. Estos aprenderán tras su contacto con el primero, y serán capaces de reaccionar contra la amenaza. Las amenazas se conocen, así, rápidamente entre toda la población de agentes, y la reacción será más rápida.

Sin embargo, incluso las amenazas más raras serán detectadas, puesto que al menos un agente las reconocerá. Los agentes también tienen una reacción (si bien más débil) sobre amenazas similares a las conocidas. Pueden aprender para mejorar su respuesta.

Es posible inyectar regularmente ataques conocidos para mantener el reconocimiento de estos entre los agentes, como se produce con las vacunas, que se refuerzan regularmente para entrenar la memoria de nuestro propio sistema inmunitario.

Además, los agentes pueden desplazarse y **aprender** los unos de los otros, lo que permite obtener respuestas mejor adaptadas y más rápidas, incluso contra ataques todavía desconocidos.

Estos sistemas son particularmente apreciados en seguridad para detectar fraudes y combatirlos (ataques sobre los sistemas informáticos, fraude bancario...).

4. Autómatas celulares

En un **autómata celular**, el entorno es una malla regular. Cada agente se sitúa sobre una de las casillas de la malla y no puede cambiar. Posee diferentes estados posibles (por ejemplo, colores).

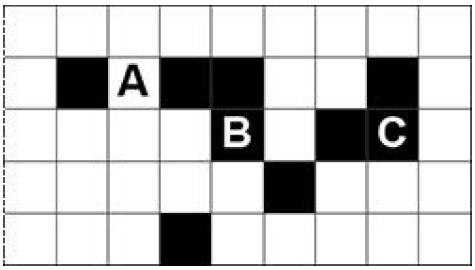
Tras cada iteración, puede cambiar de estado siguiendo algunas reglas simples, siempre basadas en los estados de sus vecinos cercanos. La malla va a evolucionar a lo largo del tiempo.

El comportamiento emergente de las mallas es complejo y permite crear todas las formas.

El autómata celular más conocido es el "**juego de la vida**" (que no tiene nada que ver con un juego desde el punto de vista de que no existe una posible acción sobre él):

- Las células tienen solamente dos estados: vivas o muertas (representadas generalmente por negro o blanco).
- Actualizan su estado en función de sus 8 vecinas inmediatas: una célula rodeada por exactamente tres células vivas se convierte o permanece viva, una célula viva rodeada exactamente por dos células vivas permanece viva, todas las demás células mueren.

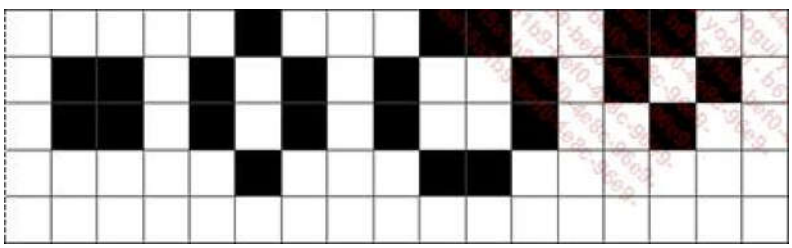
Veamos una malla con tres células llamadas A, B y C. La célula A está muerta, las otras dos células están vivas.



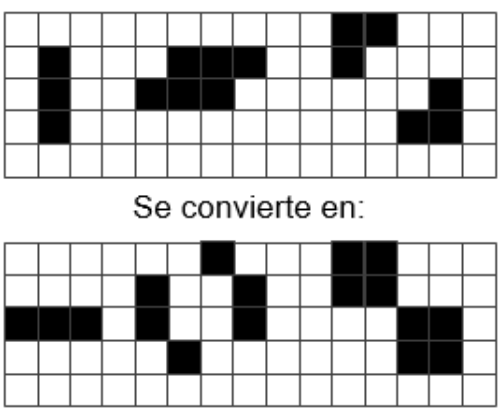
La célula A está rodeada por 2 células vivas solamente, de modo que permanece muerta. La célula B está rodeada de 3 células vivas, de modo que estará viva en la siguiente iteración. Por último, la célula C está rodeada por 2 células vivas, de modo que conserva su estado.

Con estas reglas tan sencillas es posible obtener formas estables (que no cambian con el paso del tiempo), otras que alternan diferentes formas en bucle y otras que se desplazan.

Existen muchas formas que tienen un nombre particular. He aquí cuatro estructuras estables: el bloque, la colmena, la charca y el barco.



Y he aquí algunos osciladores de periodo 2: el parpadeador, el sapo y el faro.



Se ha llegado a demostrar que todas las puertas lógicas (y, o, no) pueden representarse mediante este juego de la vida, lo que significa que cualquier programa informático puede estar representado basándose en un dibujo en una malla con algunas reglas simples. Es posible modelizar cualquier sistema complejo.

El juego de la vida, sin embargo, se utiliza poco en la práctica, aunque muchos investigadores trabajan en sus capacidades, en particular en su capacidad de hacer emerger estructuras complejas.

Dominios de aplicación

Los dominios de aplicación de los sistemas multiagentes son muy numerosos, gracias a la diversidad de los algoritmos.

1. Simulación de multitudes

El primer uso es la **simulación de multitudes**. Muchas aplicaciones utilizan agentes para simular personas que se desplazan en un lugar, lo que permite comprender la reacción en caso de evacuación y descubrir zonas de posibles aglomeraciones.

Encontramos esta simulación en el dominio de la planificación del tráfico, para comprender y simular las modificaciones introducidas por cambios como el hecho de agregar semáforos o reducir la velocidad en ciertos tramos.

Es posible, también, simular tropas, bien sean guerreros, jugadores, animales... Estas simulaciones se utilizan en el mundo del ocio, puesto que permiten realizar animaciones con bajo coste.

Así es como la aplicación MASSIVE, líder en el mercado, ha servido para crear muchos anuncios (como Adidas, Coca Cola, Pepsi...) y sobre todo muchas películas, para simular el movimiento de multitudes (y reducir el gasto en figurantes). La aplicación se ha utilizado también en los siguientes films (entre otros): *El amanecer del planeta de los simios* (2014), *World War Z* (2013), *Godzilla* (2014), *Pompeya* (2014), *Tron Legacy* (2010), *El Señor de los Anillos* (2003), *I, Robot* (2004)...

En los dibujos animados o los videojuegos, estos algoritmos se utilizan también para representar las multitudes de fondo (por ejemplo, los espectadores), sin tener que codificar personaje a personaje, dándoles "vida".

2. Planificación

El segundo gran dominio de aplicación es la **planificación**. En efecto, en particular mediante algoritmos basados en hormigas, es posible resolver cualquier problema que se plantee como un grafo.

Podemos seleccionar y optimizar el uso de las distintas máquinas en una fábrica en función de los pedidos y de las materias primas disponibles, optimizar una flota de vehículos organizándolos más eficazmente o incluso mejorar los horarios de un servicio como el tren.

Existen aplicaciones reales sobre variantes del problema del hombre de negocios, para el uso y la organización de fábricas, para la búsqueda de rutas que se adaptan a la circulación (evitando atascos, por ejemplo) o incluso para el enrutado de paquetes en las redes...

Estos algoritmos están particularmente preparados para trabajar en entornos dinámicos, puesto que permiten adaptar en tiempo real la solución propuesta.

3. Fenómenos complejos

El tercer dominio es relativo a toda la modelización y la comprensión de **fenómenos complejos**.

Se encuentran, así, en biología para la simulación del crecimiento de poblaciones de bacterias en función del medio (mediante autómatas celulares), para la replicación de proteínas, el crecimiento de plantas...

En física permiten simular fenómenos complejos dividiéndolos en pequeños problemas más simples y dejando que la

emergencia cree el sistema completo. Pueden, por ejemplo, simular gotas de agua, niebla, llamas o incluso flujos líquidos.

Por último, en finanzas, pueden permitir optimizar las carteras de acciones o las inversiones.

4. Otros dominios

Hay muchos dominios nuevos que ahora utilizan los sistemas multiagentes, en particular la robítica, que sirve para las operaciones militares, de seguridad, o de robots utilitarios. Índia forma parte de los países que llevan a cabo investigaciones sobre robots militares y que cooperan entre sí.

Amazon utiliza el sistema KIVA, que consiste en una armada de pequeños robots que mueven los palets en los almacenes, para ayudar en la preparación de pedidos.

También cada vez se observan más sistemas basados en múltiples drones que se controlan por una sola persona y un sistema multiagente para coordinar los movimientos.

Los sistemas multiagentes pueden utilizarse en un gran número de dominios y en numerosas aplicaciones. La dificultad radica, principalmente, en la elección del entorno, de los agentes y sobre todo de sus características.

Implementación

A continuación se implementan varios ejemplos. Por su funcionamiento, estos algoritmos son principalmente gráficos; aquí presentamos también el código que, si bien es genérico para las clases de base, son aplicaciones gráficas Java que utilizan `Swing`.

El modelo MVC no se ha respetado, voluntariamente, para mantener el código más ligero y simplificar su comprensión. Se crearán dos clases gráficas para cada simulación:

- una que hereda de `JPanel` que gestiona la visualización gráfica y la actualización de la aplicación.
- una que contiene el `main` y que ejecuta la ventana principal agregándole el panel previamente creado.

1. Banco de peces 2D

La primera aplicación es una simulación de un banco de peces, inspirado en los boids de Reynolds, en dos dimensiones.

Vamos a ver un conjunto de peces, representados como trazos, desplazándose por un océano virtual y evitando zonas peligrosas en su interior (que pueden ser obstáculos físicos o zonas con depredadores).

El comportamiento del banco se obtendrá únicamente por emergencia.

a. Los objetos del mundo y las zonas que es preciso evitar

Antes de codificar los propios agentes, vamos a codificar una primera clase que puede utilizarse a la vez por los objetos y los agentes. Esta, llamada simplemente **`Objeto`**, contiene dos atributos `posX` y `posY` que indican las coordenadas del objeto. Se trata de campos públicos, para optimizar su acceso. En efecto, se realizarán muchos accesos posteriormente y la llamada a un método (con la creación de su contexto) sería una pérdida de tiempo notable.

La base de nuestra clase es la siguiente. Se crean dos constructores, uno por defecto y otro que permite inicializar ambos atributos.

```
public class Objeto {
    public double posX;
    public double posY;

    public Objeto() {}
    public Objeto(double _x, double _y) {
        posX = _x;
        posY = _y;
    }
}
```

Se agregan dos métodos que permite calcular la distancia entre el objeto y otro objeto del entorno.

El primero devuelve la distancia al cuadrado, lo que simplifica los cálculos, y la segunda la distancia exacta si es necesario.

```
   public double DistanciaCuadrado(Objeto o) {
       return (o.posX - posX) * (o.posX - posX) + (o.posY - posY)
* (o.posY - posY);
   }

   public double Distancia(Objeto o) {
       return Math.sqrt(DistanciaCuadrado (o)); }
```

Las zonas que se deben evitar, **ZonaAEvitar**, son objetos situados que poseen una propiedad que indica su alcance (`radio`) y el tiempo restante de vida de la zona (puesto que deberán desaparecer automáticamente), llamado `tiempoRestante`.

Esta clase posee también cuatro métodos: un constructor con parámetros, un accesor que devuelve el radio, un método `Actualizar` que decrementa el tiempo de vida restante y un método `estaMuerto` que devuelve verdadero si el tiempo restante ha llegado a 0.

El código de la clase es el siguiente:

```
public class ZonaAEvitar extends Objeto {
    protected double radio;
    protected int tiempoRestante = 500;

    public ZonaAEvitar(double _x, double _y, double _radio) {
        posX = _x;
        posY = _y;
        radio = _radio;
    }

    public double getRadio() {
        return radio;
    }

    public void Actualizar() {
        tiempoRestante--;
    }

    public boolean estaMuerto() {
        return tiempoRestante <= 0;
    }
}
```

b. Los agentes-peces

Podemos pasar a la clase **Pez**, que representa a los agentes-peces. Estos heredan de la clase `Objeto`. Se le agregan, a continuación, varias constantes, que podrían modificarse si fuera necesario:

- La distancia recorrida en cada iteración (`PASO`) en una unidad arbitraria.
- La distancia que indica cuál es la zona de separación (`DISTANCIA_MIN`) y su versión al cuadrado (`DISTANCIA_MIN_CUADRADO`) para optimizar el cálculo.
- La distancia que indica hasta dónde alcanza la zona de alineamiento (`DISTANCIA_MAX`) y su versión al cuadrado (`DISTANCIA MAX CUADRADO`).

Además, la dirección de los peces se representa mediante el desplazamiento en x y el desplazamiento en y en cada iteración. Estos desplazamientos se codifican mediante los atributos `velocidadX` y `velocidadY`.

El código permite aquí tener velocidades variables entre los individuos, con una gestión de la dirección y de la velocidad simultáneamente. Sin embargo, en el código que sigue, nos contentaremos con velocidades fijas normalizando el vector dirección. Basta con eliminar la normalización para obtener velocidades variables.

Se agrega un constructor que recibe la posición de partida y el ángulo adoptado por el pez. El código de base es el siguiente:

```
import java.util.ArrayList;

public class Pez extends Objeto {
    // Constantes
    public static final double PASO = 3;
    public static final double DISTANCIA_MIN = 5;
    public static final double DISTANCIA_MIN_CUADRADO = 25;
    public static final double DISTANCIA_MAX = 40;
    public static final double DISTANCIA_MAX_CUADRADO = 1600;

    // Atributos
    protected double velocidadX;
    protected double velocidadY;

    // Métodos
    public Pez(double _x, double _y, double _dir) {
        posX = _x;
        posY = _y;
        velocidadX = Math.cos(_dir);
        velocidadY = Math.sin(_dir);
    }
}
```

Se agregan aquí dos accesores para recuperar las velocidades en los dos ejes:

```
    public double getVelocidadX() {
        return velocidadX;
    }

    public double getVelocidadY() {
        return velocidadY;
    }
```

El siguiente método permite calcular la nueva posición del pez. Se trata, simplemente, de agregar a la posición actual la velocidad multiplicada por la longitud del desplazamiento:

```
    protected void ActualizarPosicion() {
        posX += PASO * velocidadX;
```

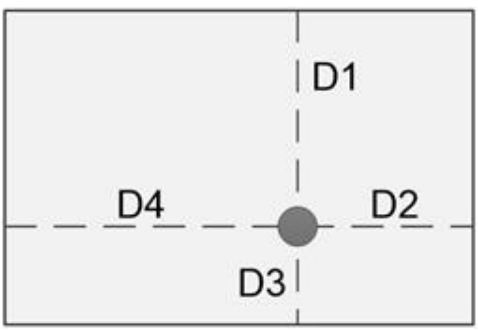
```
        posY += PASO * velocidadY;
    }
```

El método `EnAlineacion` permite saber si existe algún otro pez cerca, es decir, en la zona de alineamiento (entre la distancia mínima y la distancia máxima). Este método utiliza distancias al cuadrado.

```
protected boolean EnAlineacion(Pez p) {
        double distanciaCuadrado = DistanciaCuadrado(p);
        return (distanciaCuadrado < DISTANCIA_MAX_CUADRADO &&
distanciaCuadrado > DISTANCIA_MIN_CUADRADO);
    }
```

En la simulación, los peces deben evitar invadir a los demás peces y también los muros. Sin embargo, los muros no están localizados en un punto determinado, sino que es necesario calcular la distancia respecto a los muros. `DistanciaAlMuro` devuelve la menor distancia.

En el siguiente caso, por ejemplo, la distancia devuelta sería D3.



El código de este método es el siguiente:

```
    protected double DistanciaAlMuro(double muroXMin, double
muroYMin, double muroXMax, double muroYMax) {
        double min = Math.min(posX - muroXMin, posY - muroYMin);
        min = Math.min(min, muroXMax - posX);
        min = Math.min(min, muroYMax - posY);
        return min;
    }
```

Para simplificar el cálculo de las velocidades en los distintos casos que se presentan, agregamos una función que permite normalizarlas. En efecto, lo haremos de modo que la velocidad de un pez sea constante en el tiempo. Se normaliza, por tanto, el vector velocidad:

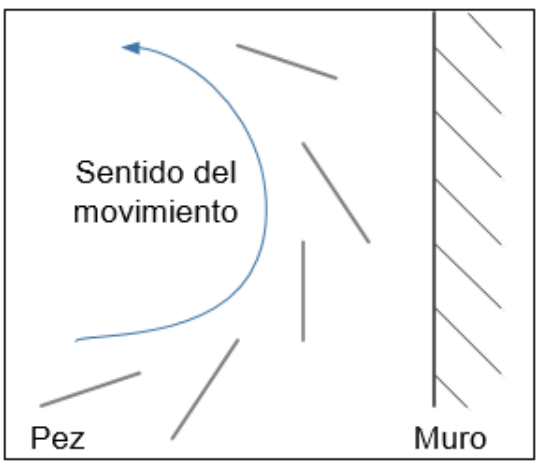
```
    protected void Normalizar() {
        double longitud = Math.sqrt(velocidadX * velocidadX +
velocidadY * velocidadY);
        velocidadX /= longitud;
        velocidadY /= longitud;
    }
```

El comportamiento del pez es muy simple:

- Si hay un muro o una zona que se debe evitar en la zona muy próxima, se evita (reglas 1 y 2).
- Si hay un pez en la zona muy próxima, nos alejamos (regla 3).
- Si hay un pez en la zona próxima, nos alineamos con él (regla 4).

Son necesarios cuatro métodos, uno por comportamiento. Empezamos evitando los muros. Para ello, en primer lugar debemos detenernos en el muro en el caso de que el desplazamiento hubiera permitido salir del océano virtual. A continuación, se modifica la dirección del pez en función del muro: los muros horizontales modifican la velocidad horizontal, sin modificar la velocidad vertical, de modo que se hace girar al pez conservando globalmente su dirección actual.

He aquí el esquema de un pez que llega a un muro y la trayectoria que sigue:



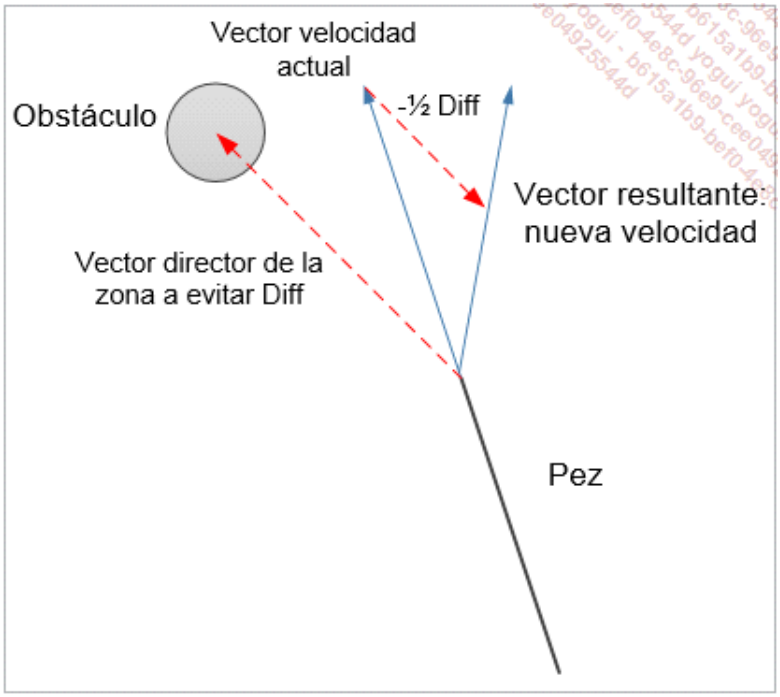
El método termina normalizando el nuevo vector y devuelve verdadero si se detecta un muro (puesto que en este caso no puede aplicarse ningún otro comportamiento). Por razones de legibilidad, el código se ha dividido en varios métodos.

```
    protected boolean EvitarMuros(double muroXMin, double muroYMin,
double muroXMax, double muroYMax) {
       PararEnMuro (muroXMin, murYMin, murXMax, murYMax);
       double distancia = DistanceAlMuro(murXMin, murYMin, murXMax,
murYMax);
       if (distancia < DISTANCE_MIN) {
           CambiarDireccionMuro(distancia, murXMin, murYMin,
murXMax, murYMax);
           Normalizar();
           return true;
       }
       return false;
   }

   private void PararEnMuro(double murXMin, double murYMin, double
murXMax, double murYMax) {
        if (posX < muroXMin) {
            posX = muroXMin;
        }
```

```
        else if (posY < muroYMin) {
            posY = muroYMin;
        }
        else if (posX > muroXMax) {
            posX = muroXMax;
        }
        else if (posY > muroYMax) {
            posY = muroYMax;
        }
       }
   }
private void CambiarDireccionMuro(double distancia, double
murXMin, double muroYMin, double muroXMax, double muroYMax) {
            if (distancia == (posX - muroXMin)) {
                velocidadX += 0.3;
            }
            else if (distancia == (posY - muroYMin)) {
                velocidadY += 0.3;
            }
            else if (distancia == (muroXMax - posX)) {
                velocidadX -= 0.3;
            }
            else if (distancia == (muroYMax - posY)) {
                velocidadY -= 0.3;
            }
    }
```

Para evitar los obstáculos, buscaremos la zona que se debe evitar más próxima a la que estemos. Si existe, efectivamente, un obstáculo muy cerca de nosotros (dos veces el radio del obstáculo), calcularemos el vector dirección `diff` entre el pez y el obstáculo. Aplicaremos una modificación del vector velocidad suprimiendo la mitad de este vector `diff`. Por último, normalizaremos el nuevo vector dirección y devolveremos verdadero si hemos debido evitar una zona.

He aquí, por ejemplo, el resultado obtenido: el antiguo vector dirección se modifica para alejarse de la zona que es preciso evitar.



El código de la función que permite evitar el objeto es:

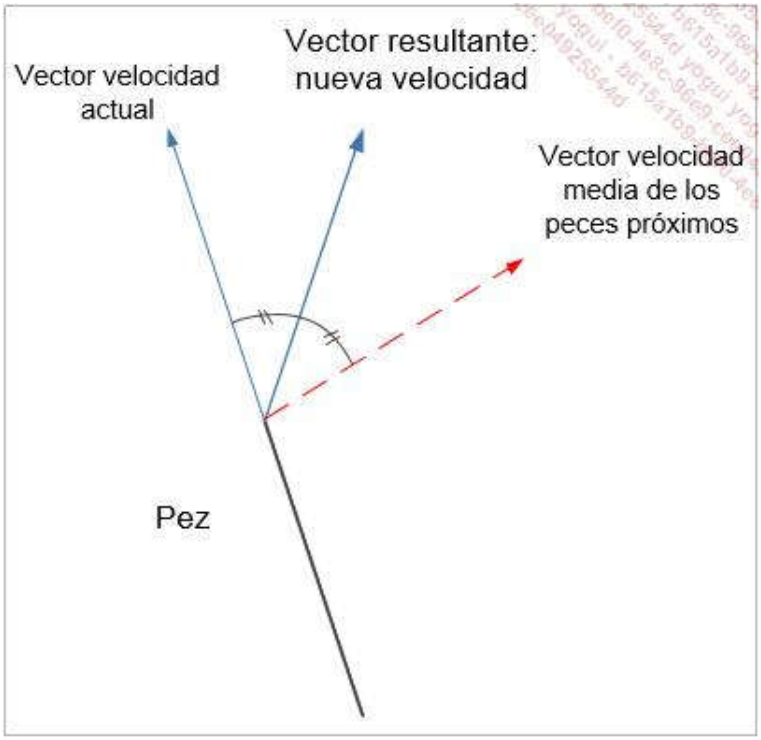
```
    protected boolean EvitarObstaculos(ArrayList<ZonaAEvitar> obstaculos) {
        if (!obstaculos.isEmpty()) {
            // Búsqueda del obstáculo más cercano
            ZonaAEvitar obstaculoProximo = obstaculos.get(0);
            double distanciaCuadrado = DistanciaCuadrado(obstaculoProximo);
            for (ZonaAEvitar o : obstaculos) {
                if (DistanciaCuadrado(o) < distanciaCuadrado) {
                    obstaculoProximo = o;
                    distanciaCuadrado = DistanciaCuadrado(o);
                }
            }

            if (distanciaCuadrado < (4*obstaculoProximo.radio *
obstaculoProximo.radio))  {
                // Si colisiona, se calcula el vector diff
                double distancia = Math.sqrt(distanciaCuadrado);
                double diffX = (obstaculoProximo.posX - posX) / distancia;
                double diffY = (obstaculoProximo.posY - posY) / distancia;
                velocidadX = velocidadX - diffX / 2;
                velocidadY = velocidadY - diffY / 2;
                Normalizar();
                return true;
            }
        }
        return false;
    }
```

Para evitar peces demasiado próximos de manera flexible, se calcula el vector unitario entre el agente y el pez más próximo a él, que sustraeremos de su propia dirección (en realidad, solo se sustrae un cuarto de la diferencia). Terminaremos normalizando nuestro vector velocidad y devolveremos verdadero si hemos evitado un

pez.

```
    protected boolean EvitarPeces(Pez[] peces) {
        // Búsqueda del pez más cercano
        Pez p;
        if (!peces[0].equals(this)) {
            p = peces[0];
        }
        else {
            p = peces[1];
        }
        double distanciaCuadrado = DistanciaCuadrado(p);
        for (Pez pez : peces) {
            if (DistanciaCuadrado(pez) < distanciaCuadrado && !
pez.equals(this)) {
                p = pez;
                distanciaCuadrado = DistanciaCuadrado(p);
            }
        }

        // Evitar
        if (distanciaCuadrado < DISTANCIA_MIN_CUADRADO) {
            double distancia = Math.sqrt(distanciaCuadrado);
            double diffX = (p.posX - posX) / distancia;
            double diffY = (p.posY - posY) / distancia;
            velocidadX = velocidadX - diffX / 4;
            velocidadY = velocidadY - diffY / 4;
            Normalizar();
            return true;
        }
        return false;
    }
```

El último comportamiento se corresponde con el alineamiento. En este caso, se buscan en primer lugar todos los peces en nuestra zona de alineamiento. La nueva dirección del pez será una media entre la dirección de los demás peces y su dirección actual, para tener cierta fluidez en los movimientos.



Vamos a calcular la dirección media de los peces próximos y normalizar el vector al final:

```
    protected void CalcularDireccionMedia(Pez[] peces) {
        double velocidadXTotal = 0;
        double velocidadYTotal = 0;
        int numTotal = 0;
        for (Pez p : peces) {
            if (EnAlineacion(p)) {
                velocidadXTotal += p.velocidadX;
                velocidadYTotal += p.velocidadY;
                numTotal++;
            }
        }
        if (numTotal >= 1) {
            velocidadX = (velocidadXTotal / numTotal + velocidadX) / 2;
            velocidadY = (velocidadYTotal / numTotal + velocidadY) / 2;
            Normalizar();
        }
    }
```

El último método permite actualizar los peces y se corresponde con su funcionamiento global. Para ello, se busca en primer lugar evitar un muro, a continuación un obstáculo y por último un pez. Si no hay nada que evitar, se pasa al comportamiento de alineamiento. Por último, una vez se ha calculado la nueva dirección, se calcula la nueva posición mediante `ActualizarPosicion()`.

```
    protected void Actualizar(Pez[] peces,
ArrayList<ZonaAEvitar> obstaculos, double ancho, double alto) {
        if (!EvitarMuros(0,0,ancho,alto)) {
            if (!EvitarObstaculos(obstaculos)) {
                if (!EvitarPeces(peces)) {
```

```
                    CalcularDireccionMedia(peces);
                }
            }
        }
        ActualizarPosicion();
    }
```

Los agentes están completamente codificados.

c. El océano

El entorno del banco de peces es un océano virtual. Este se actualiza bajo demanda, de manera asíncrona. Es necesario que el océano pueda avisar a la interfaz que la actualización se ha terminado para que se produzca la visualización.

Vamos a utilizar el design pattern Observador: el océano es un objeto observable, y avisará a todos aquellos que le observen cuando se haga su actualización. A la inversa, la interfaz podrá ser avisada a través de una notificación, de que la actualización se ha hecho y que se debe refrescar.

Hasta Java 8, se utilizan las clases `Observer` y `Observable`. Sin embargo, estos no se usan en Java 9. Ahora, se aconseja utilizar un `PropertyChangeSupport` e implementar la interfaz `PropertyChangeListener`.

La clase **Océano** tiene un atributo `support` y dos métodos para agregar o eliminar observadores. Además, el océano incluye un array de peces (por motivos de optimización, como el número de peces es fijo, es preferible un array a una lista) y una lista de obstáculos (cuyo número es variable). Se le agrega también un generador aleatorio y dos atributos para indicar su tamaño (`ancho` y `alto`). Para terminar, se agrega un contador. En efecto, necesitamos un atributo que cambie para desencadenar el envío de la notificación (y los peces no mueran, la lista de peces se considera no modificable).

La base del océano es, entonces, la siguiente:

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.util.ArrayList;
import java.util.Random;

// El océano en el que nadan los peces
public class Oceano
    // Atributos
    protected Pez[] peces;
    protected ArrayList<ZonaAEvitar> obstaculos;
    protected Random generador;
    protected double ancho;
    protected double alto;
    private PropertyChangeSupport support;
    private int contador;

    // Métodos
    public void AgregarChangeListener(PropertyChangeListener pcl) {
         support.addPropertyChangeListener(pcl);
    }

    public void EliminarPropertyChangeListener(PropertyChangeListener pcl) {
```

```
        support.removePropertyChangeListener(pcl);
    }
}
```

Este océano posee también un constructor. Aplicará, en primer lugar, el tamaño que se pasa como parámetro, y a continuación inicializará el número de peces deseados (cada uno se posiciona y alinea aleatoriamente). La lista de obstáculos está vacía al principio.

```
    public Oceano(int _numPeces, double _ancho, double _alto) {
        support = new PropertyChangeSupport(this);
        contador = 0;
        ancho = _ancho;
        alto = _alto;
        generador = new Random();
        obstaculos = new ArrayList();
        peces = new Pez[_numPeces];
        for (int i = 0; i < _numPeces; i++) {
            peces[i] = new Pez(generador.nextDouble() *
ancho, generador.nextDouble() * alto,
generador.nextDouble() * 2 * Math.PI);
        }
    }
```

A continuación, desde la interfaz, se podrán agregar obstáculos. El método `AgregarObstaculo` crea una nueva zona que evitar en las coordenadas indicadas, con el alcance solicitado, y la agrega a la lista actual.

```
    public void AgregarObstaculo(double _posX, double _posY, double radio) {
        obstaculos.add(new ZonaAEvitar(_posX, _posY, radio));
    }
```

La actualización de los obstáculos consiste simplemente en pedir a cada zona que se actualice (es decir, que reduzca su tiempo restante de vida) y a continuación que elimine las zonas que han alcanzado su final de vida.

```
    protected void ActualizarObstaculos() {
        for(ZonaAEvitar obstaculo : obstaculos) {
            obstaculo.Actualizar();
        }
        obstaculos.removeIf(o -> o.estaMuerto());
    }
```

Para realizar la actualización de los peces, se invoca, para cada pez, a su método de actualización:

```
    protected void ActualizarPeces() {
        for (Pez p : peces) {
            p.Actualizar(peces, obstaculos, ancho, alto);
        }
    }
```

Se invoca el método principal desde la interfaz. Consiste en solicitar la actualización de todo el océano. Se actualizan los obstáculos y, a continuación, los peces. Por último, se va a avisar a los suscritos a través de `firePropertyChange` y la actualización del contador.

```
    public void ActualizarOceano() {
        ActualizarObstaculos();
        ActualizarPeces();
        support.firePropertyChange("changed", this.contador,
this.contador+1);
        this.contador++;
    }
```

Todas las clases de base están ahora implementadas. Tan solo queda agregar la interfaz.

d. La aplicación gráfica

El programa principal es una aplicación Java gráfica basada en los componentes `Swing`.

Empezamos definiendo una clase **`OceanoJPanel`** que hereda de `JPanel`. Además, esta clase implementará dos interfaces:

- `PropertyChangeListener`: para poder abonarse al evento y recibir notificación de las actualizaciones.
- `MouseListener`: para gestionar los clic del ratón.

En efecto, como consecuencia de cada pulsación del ratón, se creará una +sección a evitar en el lugar seleccionado.

El código de base del panel es el siguiente:

```
import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.util.Timer;
import java.util.TimerTask;

public class OceanoJPanel extends JPanel implements MouseListener
PropertyChangeListener {
}
```

Para comenzar, el panel va a poseer dos atributos: un atributo `oceano` y un `timer` (que permite lanzar a intervalos regulares la actualización de los peces):

```
    protected Oceano oceano;
   protected Timer timer;
```

Para inicializarlo, hará falta un constructor y un método que lo ejecute. En efecto, el tamaño del panel no está definido en el constructor, sin embargo el océano lo necesita. El constructor fija un fondo azul claro y agrega el `listener` para las acciones del ratón:

```
   public OceanoJPanel() {
       this.setBackground(new Color(150, 255, 255));
       this.addMouseListener(this);
   }
```

El método `Ejecutar` crea un nuevo océano con 250 peces y lanza el `timer`, que solicita la actualización del océano cada 15 milisegundos:

```
    public void Ejecutar() {
       oceano = new Oceano(250, this.getWidth(), getHeight());
       oceano.agregarChangeListener(this);
       TimerTask tarea = new TimerTask() {
           @Override
           public void run() {
               oceano.ActualizarOceano();
           }
       };
       timer = new Timer();
       timer.scheduleAtFixedRate(tarea, 0, 15);
   }
```

Para la visualización, se necesitan varios métodos. El primero dibuja un pez. Para ello, nos contentamos con realizar un trazo de 10 píxeles de largo, partiendo de la "cabeza" del pez hasta su "cola":

```
    protected void DibujarPez(Pez p, Graphics g) {
       g.drawLine((int) p.posX, (int) p.posY, (int) (p.posX - 10
* p.velocidadX), (int) (p.posY - 10 * p.velocidadY));
   }
```

El segundo método permite dibujar una zona de las que se debe evitar. Se trazará un círculo respetando su radio:

```
   protected void DibujarObstaculo(ZonaAEvitar o, Graphics g) {
       g.drawOval((int) (o.posX - o.radio), (int) (o.posY -
o.radio), (int) o.radio * 2, (int) o.radio * 2);
   }
```

Se codifica a continuación el método `update`, que se invoca por el océano cuando se actualiza. Este método consiste simplemente en relanzar la visualización del panel:

```
   @Override
   public void propertyChange(PropertyChangeEvent evt) {
       this.repaint();
```

```
    }
```

Para que esto funcione, hay que redefinir a continuación el método `paintComponent` (que se invoca por `repaint`). Empezamos borrando la ventana, y a continuación se dibujan todos los peces y por último todos los obstáculos.

```
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (Pez p : oceano.peces) {
            DibujarPez(p, g);
        }
        for (ZonaAEvitar o : oceano.obstaculos) {
            DibujarObstaculo(o, g);
        }
    }
```

Queda, por último, gestionar el ratón. Si se hace clic en el panel, se agrega simplemente un obstáculo en el lugar correspondiente, representado con la forma de un círculo de 10 píxeles de radio:

```
    @Override
    public void mouseClicked(MouseEvent e) {
        oceano.AgregarObstaculo(e.getX(), e.getY(), 10);
    }
```

Los demás métodos de `MouseListener` son obligatorios, pero no útiles para nosotros. Su código está, por tanto, vacío:

```
    @Override
    public void mousePressed(MouseEvent e) {}
    @Override
    public void mouseReleased(MouseEvent e) {}
    @Override
    public void mouseEntered(MouseEvent e) {}
    @Override
    public void mouseExited(MouseEvent e) {}
```

Nuestra clase `OceanoJPanel` está terminada. Vamos a pasar a nuestra clase que ejecuta el programa. Esta clase se llama **`Aplicacion`**. Contiene únicamente el método `main`. Este crea en primer lugar una ventana `JFrame`, y a continuación crea un `OceanoJPanel` que se aloja en su interior, hace visible la ventana e inicia la simulación.

```
import javax.swing.JFrame;

public class Aplicacion {
    public static void main(String[] args) {
        // Creación de la ventana
        JFrame ventana = new JFrame();
```

```
        ventana.setTitle("Banco de peces");
        ventana.setSize(600, 400);
        ventana.setLocationRelativeTo(null);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setResizable(false);
        // Creación del contenido
        OceanoJPanel panel = new OceanoJPanel();
        ventana.setContentPane(panel);
        // Visualización
        ventana.setVisible(true);
        panel.Ejecutar();
    }
}
```

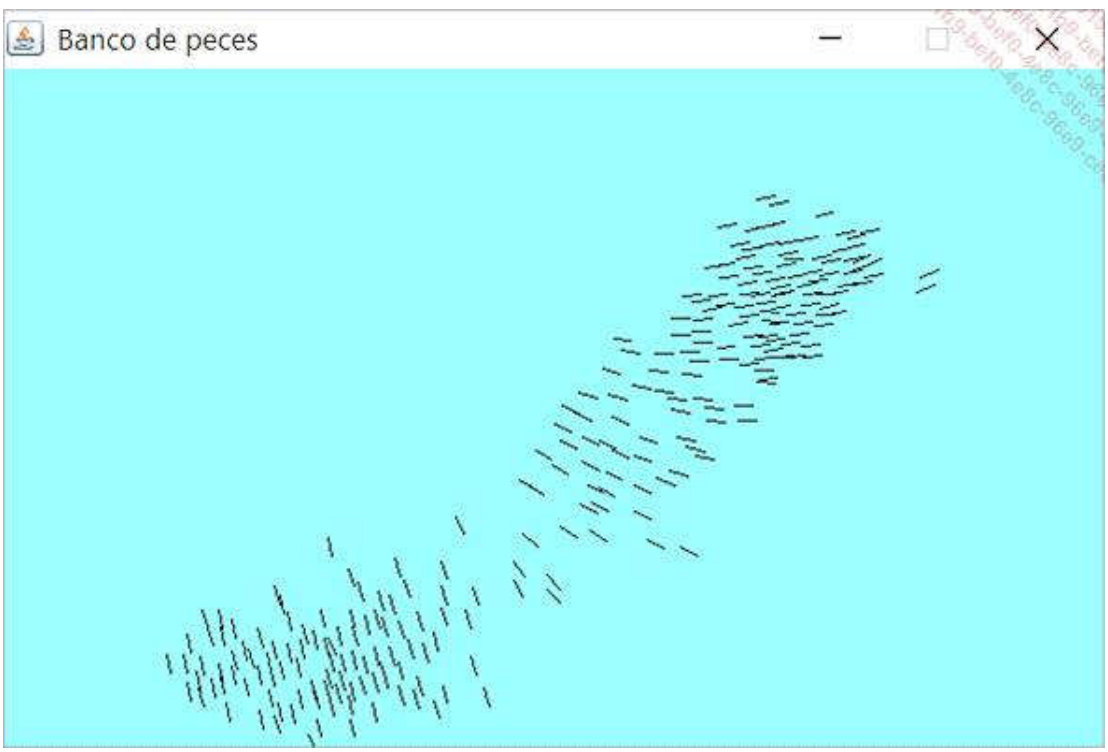
La simulación es ahora completamente funcional.

e. Resultados obtenidos

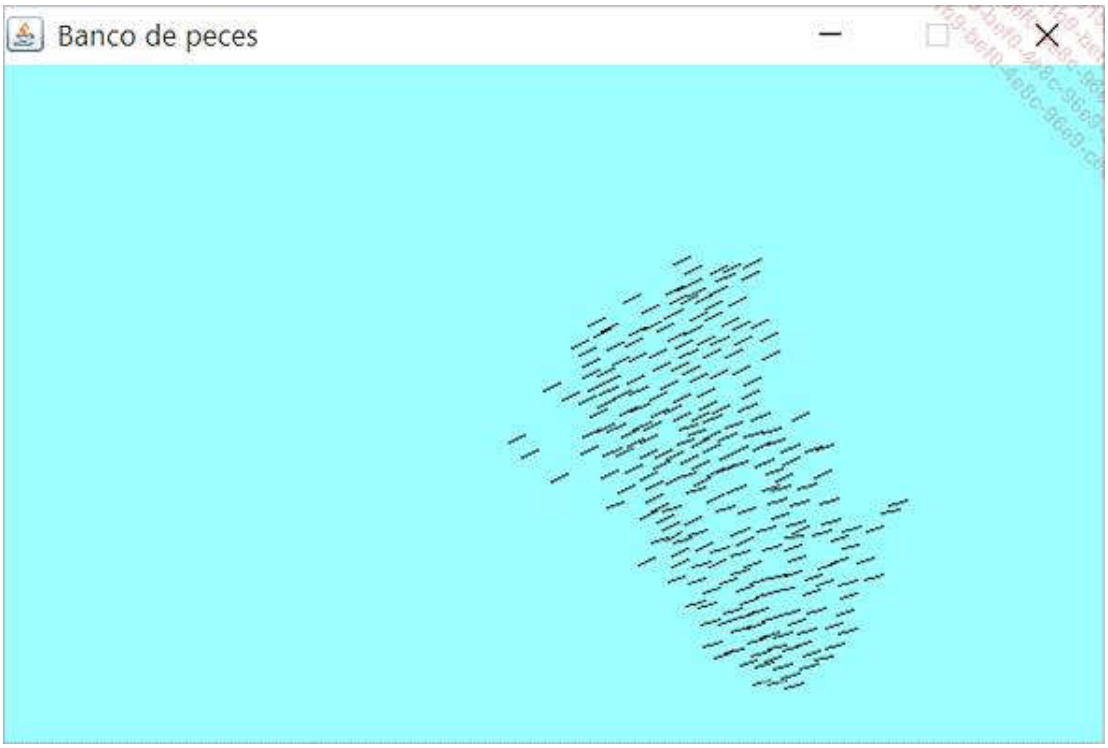
Los peces se aproximan rápidamente los unos a los otros para crear varios grupos que terminan fusionándose. Se desplazan en bancos y evitan los obstáculos. Según las disposiciones, el grupo puede separarse en dos grupos o más, pero el conjunto del banco termina siempre formándose.

El comportamiento en banco de peces es totalmente emergente, las reglas codificadas son muy simples y están basadas únicamente en su vecindario próximo o muy próximo. A diferencia de los boids de Reynolds, nuestros peces no tienen zona de coherencia, lo que explica que el grupo se escinda en ocasiones, aunque, sin embargo, el banco se vuelve a formar a continuación.

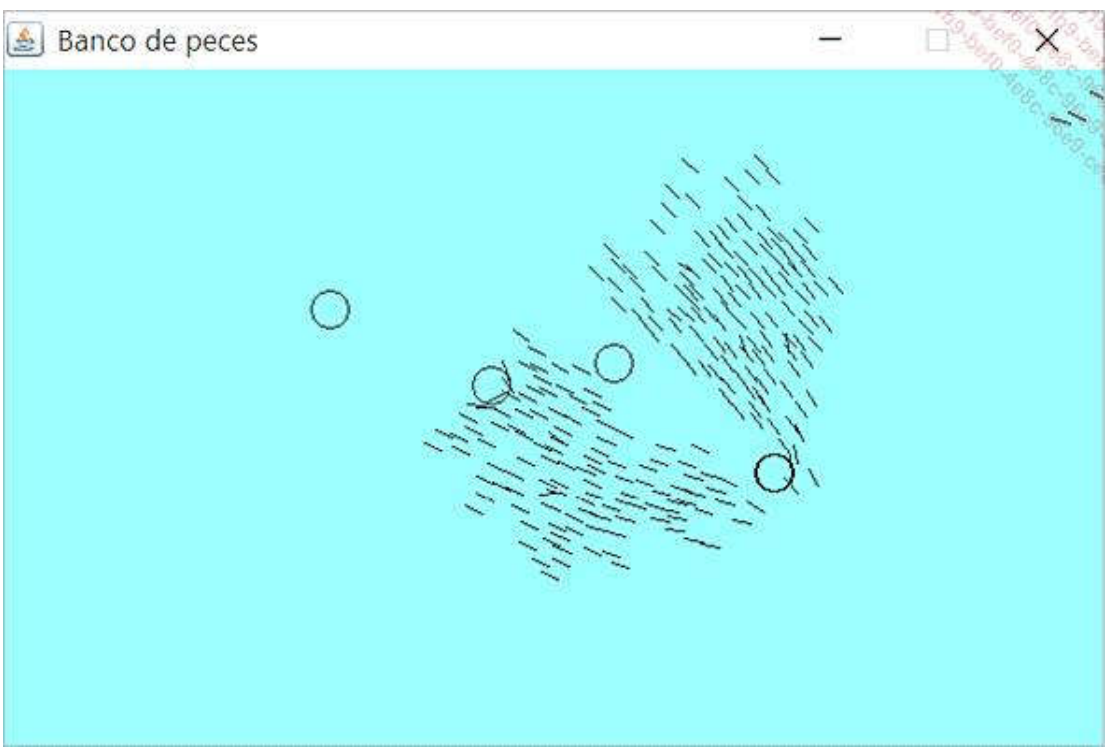
Aquí, se observa un banco de peces en formación. Se encontrarán en la esquina superior izquierda:



Aquí, el banco se forma y los peces nadan en conjunto.



En esta última captura, con zonas para evitar, vemos cómo el banco se ha dividido temporalmente yendo hacia la esquina superior izquierda. Se reagrupará con posterioridad.



2. Recogida selectiva

La segunda aplicación propone realizar una recogida selectiva de residuos por pequeños robots virtuales. En efecto, en el entorno se depositan diversos desperdicios, de tres tipos posibles. Los distintos agentes tienen la orden de recoger objetos para depositarlos allí donde exista, al menos, otro del mismo tipo.

La probabilidad de tomar un objeto depende del número de objetos presentes (de modo que es más raro tomar un objeto de un montón importante). Si los robots que tienen una carga pasan junto a un montón del mismo tipo, entonces la depositarán.

La recogida selectiva de residuos en tres montones (uno por cada tipo) no es sino un comportamiento emergente. No es necesaria ninguna comunicación ni sincronización entre los robots. Esta simulación se inspira en las termitas y en la construcción de termiteros catedrales.

a. Los residuos

Los residuos son objetos situados en el mundo, como los agentes. Reutilizaremos la clase **Objeto** creada para la simulación del banco de peces:

```
public class Objeto {
    public double posX;
    public double posY;

    public Objeto() {}
    public Objeto(double _x, double _y) {
        posX = _x;
        posY = _y;
    }

    public double Distancia(Objeto o) {
        return Math.sqrt(DistanciaCuadrado(0));
    }

    public double DistanciaCuadrado(Objeto o) {
        return (o.posX - posX) * (o.posX - posX) + (o.posY - posY) *
(o.posY - posY);
    }
}
```

Además de heredar de esta clase, los residuos implementados por la clase **Residuo** poseen dos atributos, con los accesores correspondientes:

- Uno que indica el tipo de residuo, como un valor entero.
- Uno que indica el tamaño del montón en número de residuos depositados en este sitio.

Se agrega, además, una constante `DISMINUCION` que permite saber con qué velocidad la probabilidad de coger un elemento de un montón disminuye con el tamaño.

Aquí es de 0.6. Esto significa que, si la probabilidad de recoger un elemento solo es del 100 %, la probabilidad de tomar un elemento de una pila de dos es del 60 %, la de tomarlo de una pila de tres es de 60*0.6 = 36 %, la de tomarlo de una pila de cuatro es de 36*0.6 = 21.6 %, y así sucesivamente.

La base de la clase es la siguiente:

```
public class Residuo extends Objeto {
    protected final static double DISMINUCION = 0.6;
```

```
    protected int tipo;
    protected int tamaño = 1;

    public int getTipo() {
        return tipo;
    }

    public int getTamaño() {
        return tamaño;
    }
}
```

Hay dos constructores disponibles: uno utiliza parámetros relativos a la posición y el tipo del montón, y el otro copia un elemento existente (con un tamaño inicial de 1). Este último constructor será útil cuando un agente recoja un elemento de un montón.

```
    public Residuo(double _posX, double _posY, int _tipo) {
        tipo = _tipo;
        posX = _posX;
        posY = _posY;
    }

    public Residuo(Residuo r) {
        posX = r.posX;
        posY = r.posY;
        tipo = r.tipo;
    }
```

Cada montón tiene una zona de influencia representada por su alcance. En efecto, cuanto más grande sea un montón, más atraerá a los agentes a su alrededor (es más visible, como ocurre con las montañas). Aquí, un elemento solo tiene una visibilidad de 10, y cada elemento suplementario agrega 8 puntos de visibilidad.

```
    public int ZonaInfluencia() {
        return 10 + 8 * (tamaño - 1);
    }
```

Se agregan dos métodos que permiten incrementar o decrementar el tamaño de un montón, lo que representa un agente que deposita o que toma un elemento.

```
    protected void AumentarTamaño() {
        tamaño++;
    }

    protected void DisminuirTamaño() {
        tamaño--;
    }
```

El último método de los montones permite indicar la probabilidad de tomar un elemento de un montón. Sigue el

cálculo explicado durante la declaración de la constante.

```
    protected double ProbabilidadDeTomar() {
        return Math.pow(DISMINUCION, tamaño-1);
    }
```

Los montones de residuos están ahora completamente codificados.

b. Los agentes limpiadores

Los robots o agentes limpiadores también heredan de la clase **`Objeto`**. Esta clase **`AgenteClasificacion`** posee muchos atributos suplementarios:

- La carga actualmente transportada, de tipo `Residuo`.
- El vector de velocidad expresado por sus coordenadas `velocidadX` y `velocidadY`.
- Un valor booleano que indica si está actualmente ocupado depositando o tomando una carga, o no.

Se agregan dos constantes: una indica el tamaño de un montón (`PASO`) y la otra, la probabilidad de cambiar de dirección con el paso del tiempo (`PROB_CAMBIO_DIRECCION`).

Además, los agentes tendrán que acceder regularmente al entorno que los contiene (la clase `Entorno` definida más adelante). Este último será un singleton, de modo que accederemos a él mediante el método `getInstance`. Ambas clases están vinculadas, de modo que codificaremos primero el agente basándonos en los futuros métodos del entorno.

La clase de base es la siguiente:

```
import java.util.ArrayList;
import java.util.Collections;

public class AgenteClasificacion extends Objeto {
    protected final static double PASO = 3;
    protected final static double PROB_CAMBIO_DIRECCION = 0.05;

    protected Residuo carga;
    protected double velocidadX;
    protected double velocidadY;
    protected boolean ocupado = false;

    // Resto del código aquí
}
```

Esta clase posee, como con los peces en la simulación anterior, un método `Normalizar` que permite normalizar los vectores de velocidad.

```
    protected void Normalizar() {
        double longitud = Math.sqrt(velocidadX * velocidadX +
velocidadY * velocidadY);
```

```
        velocidadX /= longitud;
        velocidadY /= longitud;
    }
```

El constructor de la clase recibe como parámetro la posición en X e Y. La velocidad se selecciona aleatoriamente y se normaliza. Se utilizará el generador aleatorio del entorno.

```
    public AgenteClasificacion(double _posX, double _posY) {
        posX = _posX;
        posY = _posY;
        velocidadX =
Entorno.getInstance().generador.nextDouble() - 0.5;
        velocidadY =
Entorno.getInstance().generador.nextDouble() - 0.5;
        Normalizar();
    }
```

Esta clase posee también un método que indica si el agente está cargado o no.

```
    public boolean estaCargado() {
        return carga != null;
    }
```

La actualización de la posición se realiza mediante `ActualizarPosicion`, que utiliza las coordenadas máximas del espacio, mediante el acceso al entorno. Las posiciones son incrementos de la velocidad multiplicada por el paso, y se verifica, a continuación, que no están fuera de la zona autorizada.

```
    public void ActualizarPosicion() {
        posX += PASO * velocidadX;
        posY += PASO * velocidadY;
        double ancho = Entorno.getInstance().getAncho();
        double alto = Entorno.getInstance().getAlto();
        if (posX < 0) {
            posX = 0;
        }
        else if (posX > ancho) {
            posX = ancho;
        }
        if (posY < 0) {
            posY = 0;
        }
        else if (posY > alto) {
            posY = alto;
        }
    }
```

El método más complejo es el que codifica el comportamiento del agente, con la modificación de su dirección.

En primer lugar, se busca cuál es la zona de residuos correspondiente a nuestro agente. Para ello, se busca si se

está dentro de la zona de un montón y si se transporta un residuo del mismo tipo que este.

Pueden darse dos casos:

- El agente no tiene un objetivo potencial (está lejos de un montón) o bien está ocupado: en este caso, se escoge una nueva dirección aleatoriamente con la probabilidad definida antes y, si el agente está fuera de toda zona, se indica que ya no está ocupado.
- El agente tiene un objetivo potencial: en este caso, se adapta su dirección hacia el centro de la zona y, si el agente está cerca del centro, deposita la carga que lleva (si tiene alguna) o bien toma un elemento del montón con una probabilidad definida en la clase `Residuo`. En ambos casos, se avisa al entorno mediante los métodos `TomarResiduo` y `DepositarResiduo`, codificados posteriormente. Además, se indica que el agente está ocupado.

El valor booleano `ocupado` permite de este modo al agente no volver a depositar enseguida el objeto recuperado o volver a coger un elemento que acababa de depositar, asegurando que sale de la zona de acción antes de poder interactuar de nuevo.

En cualquier caso, la nueva dirección se normaliza para mantener una velocidad constante.

```
    protected void ActualizarDireccion(ArrayList<Residuo> residuos) {
        // ¿Dónde ir?
        ArrayList<Residuo> enZona = new ArrayList();   µ
        enZona.addAll(residuos);
        enZona.removeIf(r -> (Distancia(r) > r.ZonaInfluencia()));
        Collections.sort(enZona, (Residuo r1, Residuo r2) ->
(Distancia(r1) < Distancia(r2) ? -1: 1));
        Residuo objetivo = null;
        if (carga != null) {
            enZona.removeIf(r -> r.tipo != carga.tipo);
        }
        if (!enZona.isEmpty()) {
            objetivo = enZona.get(0);
        }

        // ¿Tenemos un objetivo?
        if (objetivo == null || ocupado) {
            // Desplazamiento aleatorio
            if (Entorno.getInstance().generador.nextDouble() <
PROB_CAMBIO_DIRECCION) {
                velocidadX =
Entorno.getInstance().generador.nextDouble()  - 0.5;
                velocidadY =
Entorno.getInstance().generador.nextDouble()  - 0.5;
            }
            if (ocupado && objetivo == null) {
                ocupado = false;
            }
        }
    else {
        // Ir al objetivo
            velocidadX = objetivo.posX - posX;
            velocidadY = objetivo.posY - posY;
            // ¿Objetivo alcanzado?
            if (Distancia(objetivo) < PASO) {
```

```
                if (carga == null) {
                    if
(Entorno.getInstance().generador.nextDouble() <
objetivo.ProbabilidadDeTomar()) {
                        carga =
Entorno.getInstance().TomarResiduo(objetivo);
                    }
                }
                else {
                    Entorno.getInstance().DepositarResiduo(objetivo);
                    carga = null;
                }
                ocupado = true;
            }
        }
        Normalizar();
        }
```

El comportamiento de los agentes se describe por completo en este método. No existe comunicación entre ellos, ni noción alguna acerca del mapa o del objetivo global que se quiere alcanzar.

c. El entorno

La última clase genérica es **`Entorno`**, que representa a nuestro entorno. Para poder indicar a la interfaz que hay disponible una actualización, utilizamos también aquí el patrón de diseño `Observador`.

Para obtener más explicaciones acerca de este patrón, consulte el utilizado en la clase `Oceano` de la simulación correspondiente al banco de peces.

Además, esta clase es un singleton. De esta manera, todos los agentes pueden obtener una referencia a ella mediante el método `getInstance`.

Su código de base es el siguiente:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Entorno {
    // Gestión del singleton
    private static Entorno instance;

    public static Entorno getInstance() {
        if (instance == null) {
            instance = new Entorno();
        }
        return instance;
    }
```

```
    public void AgregarChangeListener(PropertyChangeListener pcl) {
        support.addPropertyChangeListener(pcl);
    }
    // Resto del código aquí
}
```

Esta clase posee también varios atributos:

- La lista de montones de residuos en el entorno.
- La lista de agentes limpiadores.
- El tamaño del espacio definido por `ancho` y `alto`.
- Un generador aleatorio.
- El número de iteraciones desde el comienzo.
- El soporte para el pattern Observador.

```
    protected Random generador;
    protected double ancho;
    protected double alto;
    protected ArrayList<Residuo> residuos;
    protected ArrayList<AgenciaClasificacion> agentes;
    protected int numIteraciones = 0;
    private PropertyChangeSupport support;
```

El constructor por defecto inicializa ambas listas y el generador aleatorio. Un método `Inicializar` permite poblar el entorno, creando residuos y los agentes con un generador aleatorio.

```
    private Entorno() {
        residuos = new ArrayList();
        agentes = new ArrayList();
        generador = new Random();
        support = new PropertyChangeSupport(this);
    }

    public void Inicializar(int _numResiduos, int _numAgentes, double
_ancho, double _alto, int _numTiposResiduos) {
        ancho = _ancho;
        alto = _alto;
        residuos.clear();
        for (int i = 0; i < _numResiduos; i++) {
             Residuo residuo = new Residuo(generador.nextDouble() *
ancho, generador.nextDouble() * alto,
generador.nextInt(_numTiposResiduos));
             residuos.add(residuo);
        }
        agentes.clear();
        for (int i = 0; i < _numAgentes; i++) {
             AgenteClasificacion agente = new AgenteClasificacion
(generador.nextDouble()
* ancho, generador.nextDouble() * alto);
             agentes.add(agente);
```

```
    }
}
```

Dos accesores permiten recuperar el tamaño del espacio:

```
public double getAncho() {
    return ancho;
}
public double getAlto() {
    return alto;
}
```

Hemos visto que los agentes necesitan dos métodos. El primero, `DepositarResiduo`, permite indicar que un agente ha depositado un nuevo elemento en un montón existente. Basta con incrementar el tamaño del montón afectado:

```
public void DepositarResiduo(Residuo r) {
    r.AumentarTamaño();
}
```

El método `TomarResiduo` comprueba el tamaño del montón: si hay un solo elemento, es el que recuperará el agente, y eliminará el montón de la lista de residuos. Si, por el contrario, el montón contiene varios elementos, se decrementará su tamaño y devolverá un nuevo elemento creado mediante copia (con una carga igual a 1).

```
public Residuo TomarResiduo(Residuo r) {
    if (r.tamaño == 1) {
        residuos.remove(r);
        return r;
    }
    else {
        r.DisminuirTamaño();
        Residuo carga = new Residuo(r);
        return carga;
    }
}
```

El último método es el que actualiza el entorno. Para cada agente, se le pide actualizar su dirección y después su posición. A continuación, se incrementa el número de iteraciones. Como los agentes se ven "atraídos" por el primer montón a su alcance, existe cierto sesgo. Cada 500 iteraciones se invierte el orden de los montones de cara a contrarrestar este sesgo. Por último, antes de incrementar el número de iteraciones, se indica que ha habido cambios y se avisa a los observadores.

```
public void Actualizar() {
    for (AgenteClasificacion agente : agentes) {
          agente.ActualizarDireccion(residuos);
          agente.ActualizarPosicion();
    }
    support.firePropertyChange("changed", numIteraciones,
```

```
numIteraciones+1);
        numIteraciones++;
        if (numIteraciones % 500 == 0) {
            Collections.reverse(residuos);
        }
   }
```

d. La aplicación gráfica

Como con la simulación anterior, vamos a crear un `JPanel` especializado para mostrar nuestros residuos y nuestros agentes.

Este hereda de `JPanel`, pero tendrá que implementar también las interfaces `PropertyChangeListener` (para poder obtener notificaciones de las actualizaciones del entorno) y `MouseListener` (para poder poner en pausa la aplicación cuando se haga clic sobre ella y relanzarla a continuación).

Su código de base es el siguiente:

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.util.Timer;
import java.util.TimerTask;
import javax.swing.JPanel;

public class ClasificacionJPanel extends JPanel implements
PropertyChangeListener,
MouseListener {
   // Aquí el código
}
```

Nuestro panel poseerá varios atributos:

- Un `timer`, que permitirá lanzar la actualización del entorno cada 10 ms, y la `TimerTask` que se ejecutará.
- Un booleano que indica si la aplicación está en pausa o no.
- Una referencia hacia el entorno para evitar tener que invocar a `getInstance` demasiado a menudo.

```
   Timer timer;
   boolean enCurso = false;
   TimerTask tarea;
   Entorno entorno;
```

El constructor permite indicar un color de fondo y agregar el `listener` para los clics del ratón. Es el método `Ejecutar` el que realmente va a inicializar el entorno (con 50 residuos de tres tipos y 30 agentes) y abonarse a las actualizaciones.

```
public ClasificacionJPanel() {
    this.setBackground(Color.WHITE);
    this.addMouseListener(this);
}

public void Ejecutar() {
    entorno = Entorno.getInstance();
    entorno.Inicializar(50, 30, getWidth(), getHeight(), 3);
    entorno.AgregarChangeListener(this);
}
```

Para la gestión del ratón, el método importante es `mouseClicked`, encargado de gestionar los clics. Este comprueba si la aplicación está en curso o no: en caso afirmativo, se detiene el `timer`; en caso contrario, se lanza un nuevo `timer`. La aplicación está en pausa tras su arranque.

```
@Override
public void mouseClicked(MouseEvent e) {
    if (enCurso) {
        // Se detiene el timer
        timer.cancel();
        timer = null;
        enCurso = false;
    }
    else {
        // Se lanza el timer
        timer = new Timer();
        tarea = new TimerTask() {
            @Override
            public void run() {
                entorno.Actualizar();
            }
        };
        timer.scheduleAtFixedRate(tarea, 0, 10);
        enCurso = true;
    }
}
```

Los demás métodos del `listener` estarán vacíos:

```
@Override
public void mousePressed(MouseEvent e) { }
@Override
public void mouseReleased(MouseEvent e) { }
@Override
public void mouseEntered(MouseEvent e) { }
@Override
public void mouseExited(MouseEvent e) { }
```

Ahora se va a gestionar el método `propertyChange`, que se invoca desde el entorno cuando este se actualiza. Se va a pedir simplemente a la interfaz que se redibuje gracias a `repaint`. Además, se agrega una visualización

por consola: el número de montones de residuos restantes y el número de agentes cargados. Esto permite saber cuándo se ha recogido todo (es decir, cuándo hay únicamente tres montones y ningún agente esté cargado).

```
   @Override
   public void propertyChange(PropertyChangeEvent evt) {
       this.repaint();
       int agentesCargados = 0;
       for (AgenteClasificacion a : entorno.agentes) {
           if (a.estaCargado()) {
               agentesCargados++;
           }
       }
       System.out.println(entorno.residuos.size() + " - " +
agentesCargados);
   }
```

Solo queda crear los métodos de visualización. Empezamos con el pintado de un agente. Se trata simplemente de un cuadrado de 3 píxeles de lado, que será gris si el agente está cargado y negro en caso contrario. Su código es bastante sencillo:

```
   public void DibujarAgente(AgenteClasificacion agente, Graphics g) {
       if (agente.estaCargado()) {
           g.setColor(Color.GRAY);
       }
       else {
           g.setColor(Color.BLACK);
       }
       g.fillRect((int) agente.posX - 1, (int) agente.posY - 1, 3, 3);
   }
```

Para los residuos, es algo más complicado. En efecto, hay que escoger en primer lugar el color que depende del tipo de montón. Tomaremos aquí, en orden, rojo, verde y azul. A continuación, hay dos formas por montón: un cuadrado de 3 píxeles de lado para representar el centro y un círculo parcialmente transparente indicando la zona de influencia del montón (que aumenta con el tamaño del montón).

```
   public void DibujarResiduo(Residuo r, Graphics g) {
       // Selección del color
       Color color;
       switch(r.tipo) {
           case 1 :
               color = Color.RED;
               break;
           case 2 :
               color = Color.GREEN;
               break;
           default :
               color = Color.BLUE;
       }
       g.setColor(color);
       // Base: cuadrado
       g.fillRect((int) r.posX - 1, (int) r.posY - 1, 3, 3);
```

```
        // Zona de influencia(redonda)
        color = new Color(color.getRed(), color.getGreen(),
color.getBlue(), 50);
        g.setColor(color);
        int zona = r.ZonaInfluencia();
        g.fillOval((int) r.posX - zona, (int) r.posY - zona, zona
* 2, zona * 2);
    }
```

Por último, podemos escribir el método `paintComponent`. Este, tras invocar el método de la clase madre, debe pedir la visualización de cada agente y luego la de cada residuo.

```
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (AgenteClasificacion agente : entorno.agentes) {
            DibujarAgente(agente, g);
         }
        for (Residuo residuo : entorno.residuos) {
            DibujarResiduo(residuo, g);
        }
    }
```

El panel está terminado, solo queda la clase que contiene el `main`, **`Aplicacion`**. Este debe crear una nueva ventana, incluir un `OrdenacionJPanel` dentro y a continuación ejecutar la aplicación:

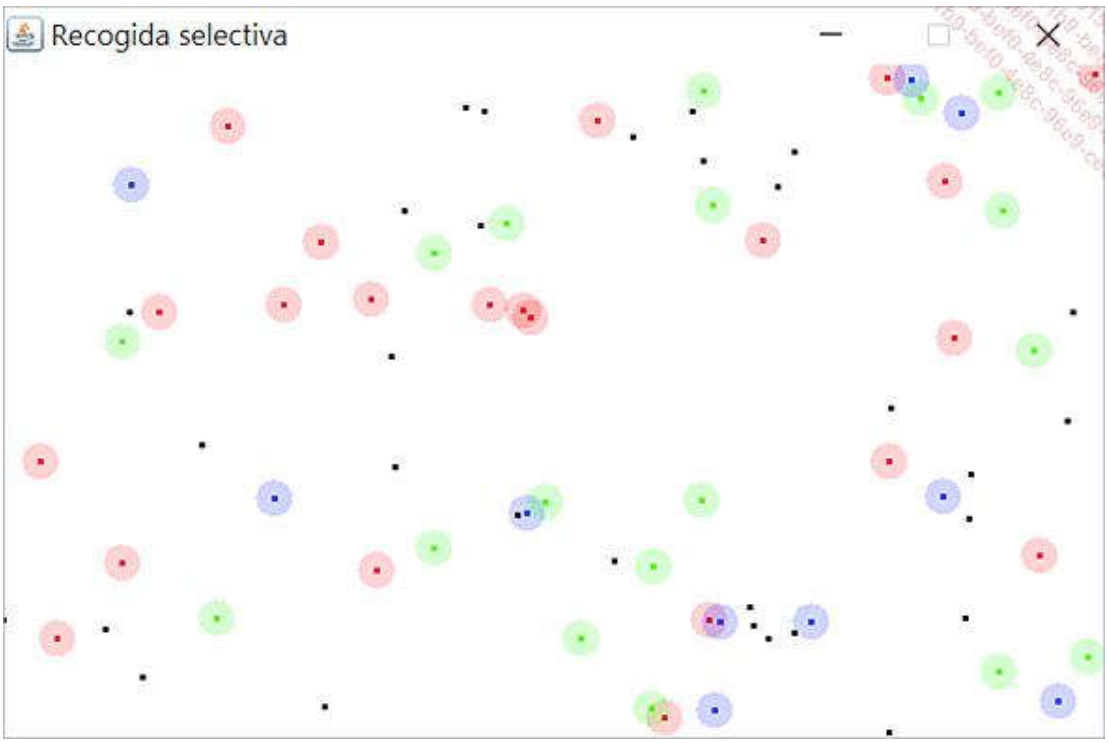
```
import javax.swing.JFrame;

// Clase que contiene el main, creación de la ventana + arranque
de simulación
public class Aplicacion {
    public static void main(String[] args) {
        // Creación de la ventana
        JFrame ventana = new JFrame();
        ventana.setTitle("Recogida selectiva");
        ventana.setSize(600, 400);
        ventana.setLocationRelativeTo(null);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setResizable(false);
        // Creación del contenido
        ClasificacionJPanel panel = new ClasificacionJPanel();
        ventana.setContentPane(panel);
        // Visualización
        ventana.setVisible(true);
        panel.Ejecutar();
    }
}
```

¡Ha terminado el código de la aplicación!

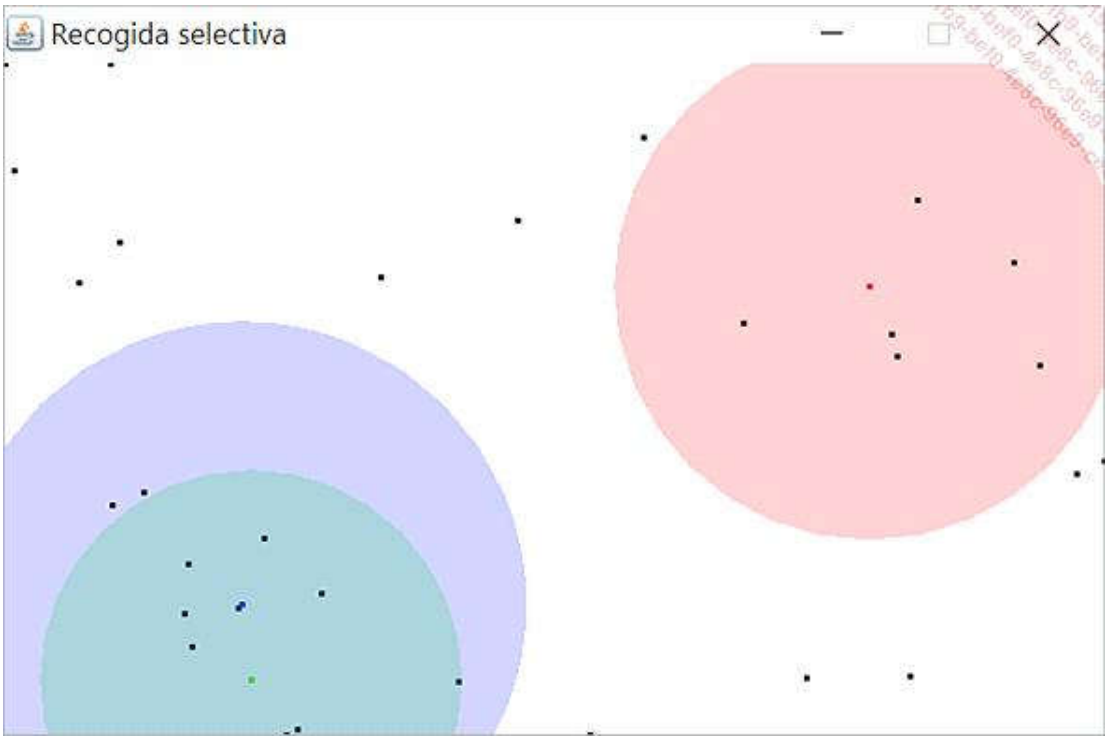
e. Resultados obtenidos

Aparte de algunas simulaciones en las que los agentes retiran todos los residuos de un tipo, lo que no permite volver a depositarlos posteriormente, todas las demás situaciones convergen hacia la presencia de tres montones, uno por cada tipo.

En una simulación típica, se parte de la siguiente situación (los residuos están rodeados, mientras que los agentes son pequeños cuadrados solitarios):



Pasados varios segundos o algunos minutos de simulación, los agentes han desplazado los residuos. Entonces podemos obtener esta configuración:



Vemos que existen únicamente tres montones, uno por cada tipo. Podemos ver también que en esta simulación existe un montón más pequeño que los otros dos; eso es debido a que los residuos se reparten aleatoriamente entre los distintos tipos.

Sin embargo, el comportamiento de la recogida selectiva ha funcionado de forma correcta únicamente por emergencia.

3. El juego de la vida

Esta aplicación es un pequeño juego de la vida. Las células, inicializadas aleatoriamente al comienzo, evolucionan en una malla según las reglas de Conway.

Se observa una estabilización al cabo de varias iteraciones, con estructuras que no evolucionan y otras que lo hacen en función de un ciclo de vida entre varias posiciones posibles.

El usuario puede, en cualquier momento, poner la aplicación en pausa o volver a iniciarla, y agregar o eliminar células vivas haciendo clic en la ventana.

a. La malla

Los agentes son muy simples, pues se trata sencillamente de células que no pueden desplazarse y que toman solo dos estados posibles: viva o muerta. No vamos a codificarlas en una clase aparte, sino directamente en la malla, que representa el entorno. Además, no existe ningún otro objeto.

La malla **`Malla`** se dibuja tras cada actualización, de modo que aquí también vamos a utilizar el patrón de diseño `Observador`. Por lo tanto, le vamos a agregar un atributo `PropertyChangeSupport`.

La malla posee, además, un ancho y un largo definido en número de células y un array de dos dimensiones que contiene todas las células, que son simplemente valores booleanos que indican si la célula situada en dicha casilla está viva o no. Se agrega un entero para contar el número de iteraciones (este valor cambia en cada iteración, el contador nos permitirá notificar a la interfaz el final de la actualización).

La base del código es la siguiente:

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.util.Random;

public class Malla {
    protected int ancho;
    protected int alto;
    protected boolean[][] contenido;
    private PropertyChangeSupport support;
    private int numIteraciones;

    public void AgregarChangeListener(PropertyChangeListener pcl) {
        support.addPropertyChangeListener(pcl);
    }
}
```

El constructor recibe como parámetro el ancho y el largo de la malla, así como la densidad de células vivas al inicio. Estas se inicializan al azar gracias a un generador aleatorio al mismo tiempo que se crea el array de células.

```
    public Malla(int _ancho, int _alto, double _densidad) {
        ancho = _ancho;
        alto = _alto;
        Random generador = new Random();
        support = new PropertyChangeSupport(this);
        numIteraciones = 0;

        contenido = new boolean[ancho][alto];
        for (int i = 0; i < ancho; i++) {
            for (int j = 0; j < alto; j++) {
                if (generador.nextDouble() < _densidad) {
                    contenido[i][j] = true;
                }
            }
        }
    }
```

Necesitamos dos métodos para poder actualizar la malla: uno que permita cambiar el estado de una célula concreta y otro que permita saber el número de células vecinas vivas.

Para cambiar el estado de una célula, basta con invertir el valor booleano:

```
    public void CambiarEstado(int fila, int columna) {
        contenido[fila][columna] = !contenido[fila][columna];
    }
```

Para contar el número de células vecinas vivas, hay que mirar en la zona adyacente a la célula (una zona de 3*3 casillas centrada en la célula). Sin embargo, hay que prestar atención a no salirse de la malla, de modo que

verificamos en primer lugar las coordenadas mínimas y máximas respecto a las dimensiones del entorno. Es preciso, también, no contar la célula del centro.

```
    public int NumVecinosVivos(int columna, int fila) {
        int i_min = Math.max(0, columna-1);
        int i_max = Math.min(ancho-1, columna+1);
        int j_min = Math.max(0, fila-1);
        int j_max = Math.min(alto-1, fila+1);
        int num = 0;
        for (int i = i_min; i <= i_max; i++) {
             for (int j = j_min; j <= j_max; j++) {
               if (contenido[i][j] && !(i==columna && j==fila)) {
                   num++;
               }
            }
        }
        return num;
    }
```

El último método es la actualización. Para ello, se crea en primer lugar una nueva malla virgen (con todas las células consideradas como muertas). A continuación se recorre la malla completa y para cada célula se cuenta cuántas vecinas tiene en la malla actual:

- Si tiene tres vecinas, estará viva en la siguiente malla.
- Si tiene dos vecinas y está viva, entonces permanecerá viva.
- En los demás casos, estará muerta (y, por tanto, no se hace nada).

Se reemplaza, a continuación, la malla anterior por la nueva malla calculada y se produce la notificación a los abonados indicando el fin de la actualización.

Se agrega a este método un valor booleano como parámetro que indica si se quiere actualizar realmente la aplicación (valor por defecto) o simplemente si se quiere producir el evento para recuperar el estado actual de las células. Este último se utilizará cuando el usuario quiera cambiar el estado de una célula para actualizar la visualización.

```
    public void Actualizar(boolean conAplicacion) {
        if (conAplicacion) {
            boolean[][] nuevaMalla = new
boolean[ancho][alto];
            for (int i = 0; i < ancho; i++) {
               for (int j = 0; j < alto; j++) {
                 int num = NumVecinosVivos(i, j);
                 if (num == 3 || (num == 2 && contenido[i][j])) {
                 nuevaMalla[i][j] = true;
                 }
               }
            }
            contenido = nuevaMalla;
        }
        support.firePropertyChange("changed", this.numIteraciones,
this.numIteraciones+1);
```

```
        this.numIteraciones++;
    }
```

No hay otras clases necesarias para gestionar el juego de la vida, de modo que solo queda la representación gráfica.

b. La aplicación gráfica

También aquí, como con las otras dos simulaciones, vamos a crear dos clases: una que herede de `JPanel`, que gestionará la representación, y una que contendrá simplemente el `main`.

Empezamos con la clase **`JuegoDeLaVidaJPanel`**. Esta hereda de `JPanel` e implementa las interfaces `PropertyChangeListener` (para poder abonarse a las notificaciones de la malla) y `MouseListener`. Esta vez, hacer clic con el botón derecho permitirá poner la aplicación en pausa o relanzarla, y los clics con el botón izquierdo permitirán modificar el estado de la célula situada bajo el ratón.

La base de esta clase es por tanto la siguiente:

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.util.Timer;
import java.util.TimerTask;
import javax.swing.JPanel;

public class JuegoDeLaVidaJPanel extends JPanel implements
PropertyChangeListener, MouseListener {
   // Aquí el código
}
```

Se agregan a continuación los atributos: un `Timer` y un `TimerTask` para poder poner la aplicación en pausa o retomarla, un booleano que indica si la simulación está lanzada y una referencia a una `Malla`.

```
Timer timer;
boolean enCurso = false;
Malla malla;
TimerTask tarea;
```

Como con las otras simulaciones, el constructor se contenta con configurar el color de fondo y después suscribirse a los clics del ratón.

```
   public JuegoDeLaVidaJPanel() {
       this.setBackground(Color.WHITE);
       this.addMouseListener(this);
   }
```

El método importante es el método `Ejecutar`, que va a inicializar la malla, con una densidad de células vivas del 10 %, se abonará a la malla y ejecutará el `timer` que permite actualizar la representación gráfica cada 500 ms.

```
   public void Ejecutar() {
       malla = new Malla(this.getWidth() / 3, getHeight() / 3,
0.1);
       malla.AgregarChangeListener(this);
       timer = new Timer();
       tarea = new TimerTask() {
           @Override
           public void run() {
               malla.Actualizar(true);
           }
       };
       timer.scheduleAtFixedRate(tarea, 0, 500);
       enCurso = true;
   }
```

Para la actualización, basta con reiniciar la visualización de la malla:

```
   @Override
   public void propertyChange(PropertyChangeEvent evt){
       this.repaint();
   }
```

A nivel de los métodos gráficos, basta con dibujar las células vivas, con cuadrados de 3 píxeles de lado.

```
   public void DibujarCelula(Graphics g, int i, int j) {
       g.fillRect(3*i-1, 3*j-1, 3, 3);
   }
```

El método `paintComponent` va a solicitar el pintado de cada célula si está viva:

```
   @Override
   public void paintComponent(Graphics g) {
       super.paintComponent(g);
       for (int i = 0; i < malla.ancho; i++) {
           for (int j = 0; j < malla.alto; j++) {
               if (malla.contenido[i][j]) {
                   DibujarCelula(g, i, j);
               }
           }
       }
   }
```

Para la gestión de los clics, en primer lugar hay que determinar de qué botón se trata:

- En el caso de un clic izquierdo, se cambia el estado de la célula situada debajo del ratón.
- En el caso de un clic derecho, se pone en pausa o se relanza la simulación.

```
   @Override
   public void mouseClicked(MouseEvent e) {
       if (e.getButton() == MouseEvent.BUTTON1) {
           malla.CambiarEstado(e.getX() / 3, e.getY() / 3);
           malla.Actualizar(false);
       }
       else if (e.getButton() == MouseEvent.BUTTON3) {
           if (enCurso) {
               timer.cancel();
               timer = null;
           }
           else {
               timer = new Timer();
               tarea = new TimerTask() {
                   @Override
                   public void run() {
                       malla.Actualizar(true);
                   }
               };
               timer.scheduleAtFixedRate(tarea, 0, 500);
           }
           enCurso = !enCurso;
       }
   }
```

Los demás métodos quedarán vacíos.

```
   @Override
   public void mousePressed(MouseEvent e) {}
   @Override
   public void mouseReleased(MouseEvent e) {}
   @Override
   public void mouseEntered(MouseEvent e) {}
   @Override
   public void mouseExited(MouseEvent e) {}
```

El panel está ahora terminado. Tan solo queda la clase que contiene el `main`, **`Aplicacion`**. Como para las demás simulaciones, en primer lugar se debe crear una ventana, a continuación incluir el panel previamente creado y por último ejecutar la simulación:

```
import javax.swing.JFrame;

public class Aplicacion {
   public static void main(String[] args) {
       // Creación de la ventana
       JFrame ventana = new JFrame();
       ventana.setTitle("Juego de la vida");
       ventana.setSize(600, 400);
```

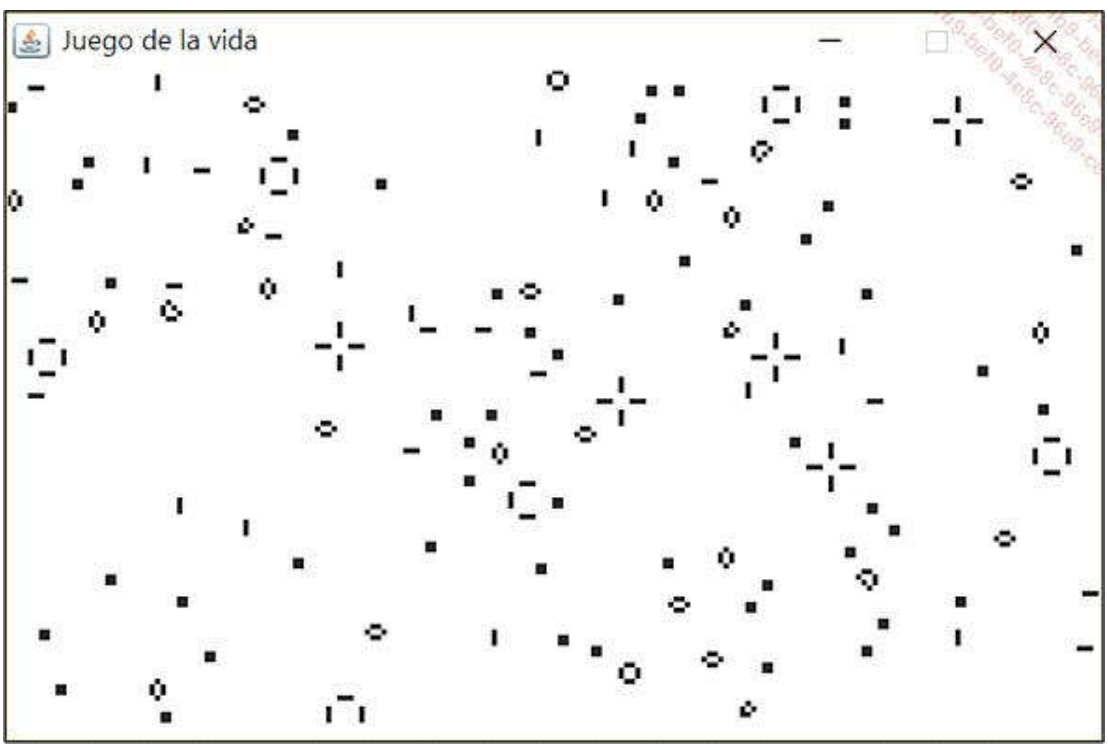
```
        ventana.setLocationRelativeTo(null);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setResizable(false);
        // Creación del contenido
        JuegoDeLaVidaJPanel panel = new JuegoDeLaVidaJPanel();
        ventana.setContentPane(panel);
        // Visualización
        ventana.setVisible(true);
        panel.Ejecutar();
    }
}
```

La simulación está ahora operacional.

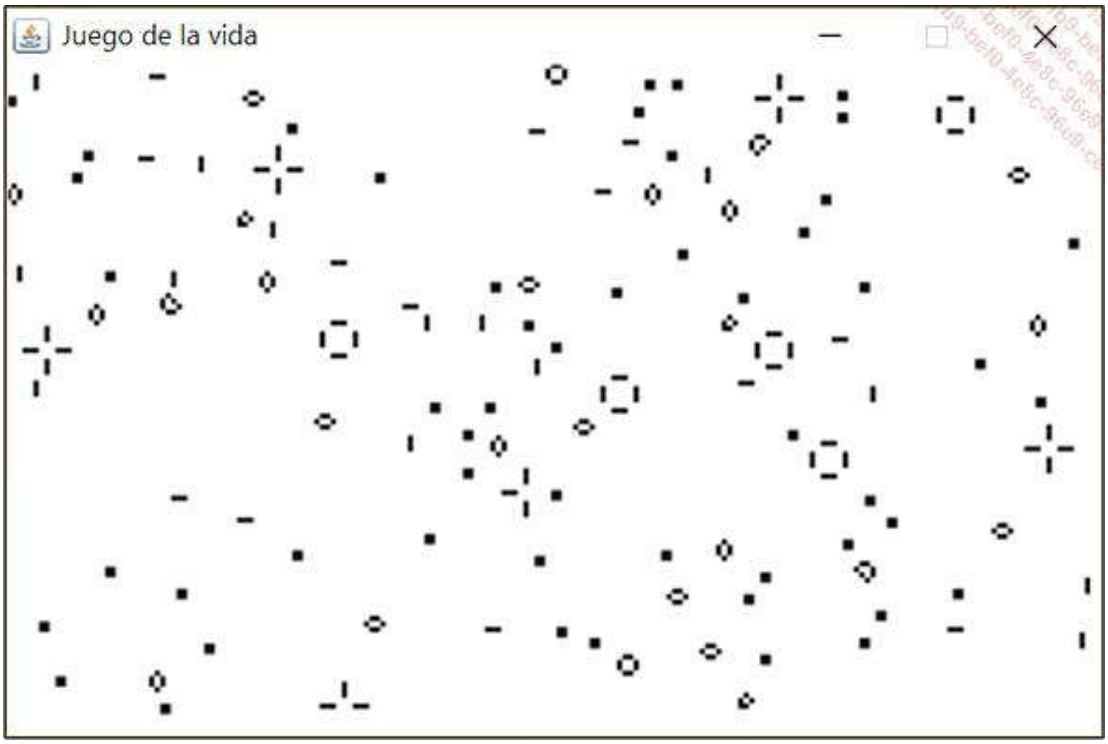
c. Resultados obtenidos

Tras la segunda iteración, todas las células aisladas desaparecen. En las siguientes iteraciones, tienen lugar "explosiones" en la malla, dejando tras de sí estructuras estables u oscilantes, en su mayoría de periodo 2. De vez en cuando aparecen también algunos barcos que se desplazan hasta encontrar otra estructura.

Tras varias generaciones, solo permanecen las estructuras estables y oscilantes. He aquí, por ejemplo, el estado final de una simulación en su iteración N:



En su iteración N+1, se obtiene la siguiente ventana:



La siguiente iteración (N+2) es idéntica a la iteración N: nuestra malla contiene únicamente estructuras estables (bloques, charcas y colmenas, presentadas antes) y estructuras oscilantes de periodo 2: parpadeadores (a menudo situados en cruz o en círculo en función del tiempo), sapos y, con menos frecuencia, barcos.

En raras ocasiones pueden aparecer otras estructuras.

El usuario puede, en todo momento, agregar o eliminar células vivas. La malla va a evolucionar, a continuación, hasta estabilizarse de nuevo.

Resumen

Los sistemas multiagentes permiten resolver un gran número de problemas, tanto en la simulación de multitudes, en la planificación y la búsqueda de rutas o en la simulación de problemas complejos, para comprenderlos mejor y ayudar en su estudio.

Se basan, todos ellos, en observaciones realizadas sobre los insectos eusociales, que son capaces de resolver tareas muy complejas a partir de reglas muy simples. La solución aparece mediante emergencia, y no según un plan preprogramado. Las abejas encuentran nuevas fuentes de alimento, las hormigas se comunican mediante feromonas para optimizar el acceso al alimento y las termitas construyen enormes termiteros climatizados.

En informática, los sistemas multiagentes contienen un entorno en el que se encuentran objetos y agentes. No existen más que unas pocas reglas que seguir, y cada problema puede tener una o varias modelizaciones posibles.

Existen, sin embargo, algunos algoritmos más conocidos entre los sistemas multiagentes. Podemos citar los algoritmos que simulan el comportamiento de manadas basadas en boids, la optimización por colonias de hormigas y sus feromonas artificiales, los sistemas inmunitarios artificiales que permiten detectar y reaccionar a ataques o amenazas y autómatas de estados finitos; entre ellos, el más conocido es el juego de la vida de Conway.

En todos los casos, es la multiplicación de agentes y los vínculos que se establecen entre ellos, directamente por comunicación o indirectamente por estigmergia (o incluso sin comunicación entre ellos), lo que permite hacer emerger la solución. Se observa la potencia de la inteligencia distribuida.

En un mundo donde existen cada vez más elementos conectados a través de Internet, comprendemos rápidamente que estos sistemas multiagentes tienen un gran porvenir y numerosas posibilidades todavía por explotar. Podríamos, de este modo, comunicar objetos conectados y hacer emerger comportamientos inteligentes basados en elementos de la vida cotidiana.

Presentación del capítulo

Durante mucho tiempo, el objetivo de la inteligencia artificial ha sido simular la inteligencia humana y obtener un sistema artificial capaz de reflexionar, tomar decisiones y aprender.

Por lo tanto, los investigadores se han interesado muy rápidamente por el funcionamiento del cerebro para intentar reproducirlo. De esta manera, Mac Culloch y Pitts en 1943 definieron las primeras neuronas artificiales.

En la actualidad, ya no buscamos crear cerebros con todas sus capacidades, sino tener sistemas que puedan resolver algunos problemas complejos, sobre los que los sistemas clásicos están limitados. De esta manera nacieron las **redes neuronales artificiales**.

Este capítulo empieza explicando los orígenes biológicos, interesándose por el funcionamiento del cerebro y más concretamente de las neuronas.

A continuación se presenta el Machine Learning, dominio que cubre varias técnicas entre las que se encuentran las redes neuronales, con las formas de aprendizaje y los tipos de problemas relacionados, así como las técnicas matemáticas.

Estas técnicas tienen límites, así que a continuación se presenta la neurona formal. Se explica el perceptrón, uno de los modelos más sencillos de redes, así como su aprendizaje. Un ejemplo permite entender mejor su funcionamiento.

Las redes de tipos perceptrones no son suficientes para resolver muchos problemas complejos. Entonces pasamos a las redes de tipo "feed-forward", que son más potentes.

El capítulo continúa con una presentación de las diferentes mejoras que se pueden añadir a estas redes. Para terminar, se exponen otros tipos de redes y los principales dominios de aplicación, para terminar esta parte teórica.

Se propone la implementación en Java de una red MLP con aprendizaje, así como su aplicación a dos problemas diferentes. Un resumen cierra este capítulo.

Origen biológico

Desde hace mucho tiempo, sabemos que el razonamiento se hace gracias al **cerebro**. Por lo tanto, esto se estudió desde muy pronto (desde el siglo 18).

Hay "mapas" del cerebro, que indican sus principales estructuras y sus funciones asociadas. Aunque todavía no se ha entendido todo, sabemos por ejemplo que el cerebelo es muy importante para la coordinación de los movimientos, o que el hipotálamo gestiona las funciones importantes como dormir, el hambre o la sed.

Al contrario de lo pudiéramos pensar a causa de una idea extendida, incluso que aparece en las películas recientes como Lucy de Luc Besson, que se estrenó en 2014, se utiliza el 100% de nuestro cerebro. Sin embargo, en un momento dado, solo se moviliza una parte de este en función de las necesidades. Una zona que no se utilizara habitualmente, sufriría una fuerte degeneración y desaparecería rápidamente.

Las células más importantes del córtex cerebral son las **neuronas**. Hay muchas, y su número se acerca a cien mil millones en cada ser humano. Estas células demandan mucha energía y son frágiles. Están protegidas y se nutren de las células gliales (el 90% de las células del cerebro), que sin embargo no tienen ninguna función en el razonamiento.

Sabemos que las neuronas se comunican entre ellas a través de impulsos eléctricos. En efecto, los "receptores" (ojo, oído, piel, etc.) envían los impulsos eléctricos a las neuronas a través de los nervios, que los tratan y transmiten o no a otras células.

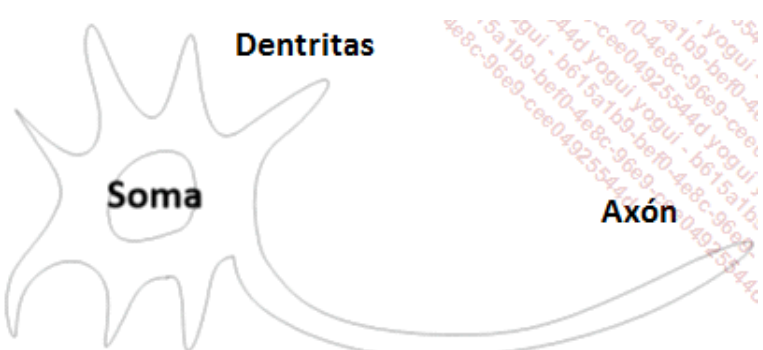
Por lo tanto, cada neurona tiene alrededor de su corazón (llamado soma):

- **dendritas**, que son su entradas,
- un largo **axón** que le sirve de salida.

Por lo tanto, las señales eléctricas llegan al soma siguiendo las dendritas y después se tratan: según la intensidad y la suma de los impulsos recibidos, la neurona envía o no un impulso a lo largo de su axón. Este se relaciona con las dendritas de otras neuronas.

El enlace físico entre dos neuronas se hace gracias a las psinopsis, sin embargo su funcionamiento no se explica en detalle porque no es útil para la comprensión de las redes artificiales.

Por lo tanto, una neurona se puede esquematizar como sigue (el axón tiene ramificaciones que le permiten conectarse a otras neuronas):



Por lo tanto, cada neurona es una entidad muy sencilla, que simplemente hace un trabajo sobre los impulsos recibidos para seleccionar o no, enviar una de salida. La potencia del cerebro reside de hecho en el número de neuronas e interconexiones entre ellas.

Las redes neuronales están inspiradas en esta neurona simple. Antes de estudiarlas más en detalle, vamos a interesarnos por las técnicas más sencillas de Machine Learning, que las redes neuronales permiten mejorar mucho.

Machine Learning

El Machine Learning o ML (traducido normalmente en español como "aprendizaje automático"), se comprende con todas las técnicas que permiten a un algoritmo aprender a partir de ejemplos, sin programación directa de la resolución.

El ML es vital en muchos dominios en los que se pide que un ordenador resuelva problemas que no sabemos modelizar exactamente, pero para los que se puede obtener ejemplos.

La mayoría de las técnicas de ML son algoritmos puramente matemáticos (resultado de las estadísticas), donde también encontramos técnicas relacionadas con la inteligencia artificial. Este es el caso de la metaheurística o de los algoritmos genéticos, pero sobre todo de las redes neuronales (normalmente llamadas "Deep Learning").

1. Formas de aprendizaje y ejemplos

El ML permite resolver diferentes tipos de problemas, que se reparten entre dos formas de aprendizaje principales. También se presenta una tercera forma de aprendizaje, aunque muy poco utilizada.

a. Aprendizaje no supervisado

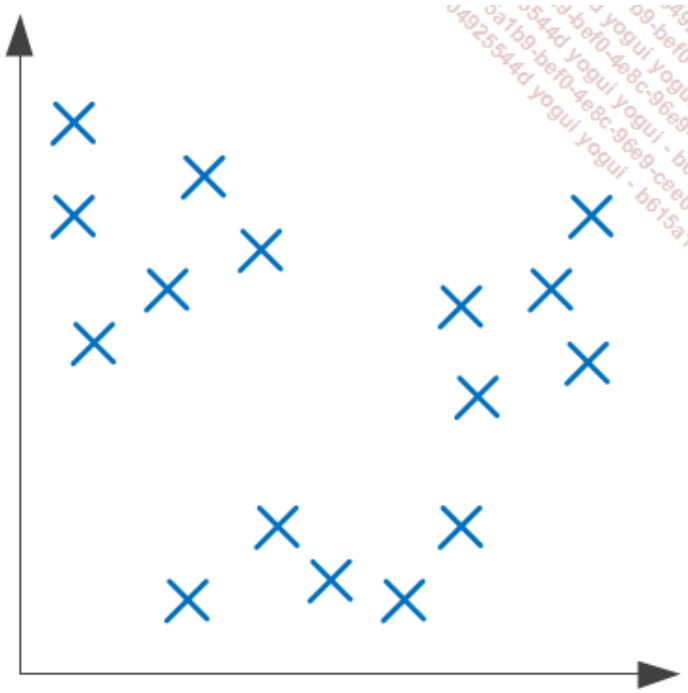
El **aprendizaje no supervisado** no es la forma de aprendizaje más habitual, pero su utilización tiende a aumentar estos últimos años. En esta forma de aprendizaje no hay resultado esperado. Se utiliza esta forma de aprendizaje para hacer el **clustering** (también llamado segmentación): tenemos un conjunto de datos y buscamos determinar las clases de hechos.

En realidad, existen otras aplicaciones del aprendizaje no supervisado, pero el clustering representa la gran mayoría de las aplicaciones actuales. Por ejemplo, podemos citar a los sistemas de recomendaciones utilizados para las tiendas en línea o la publicidad.

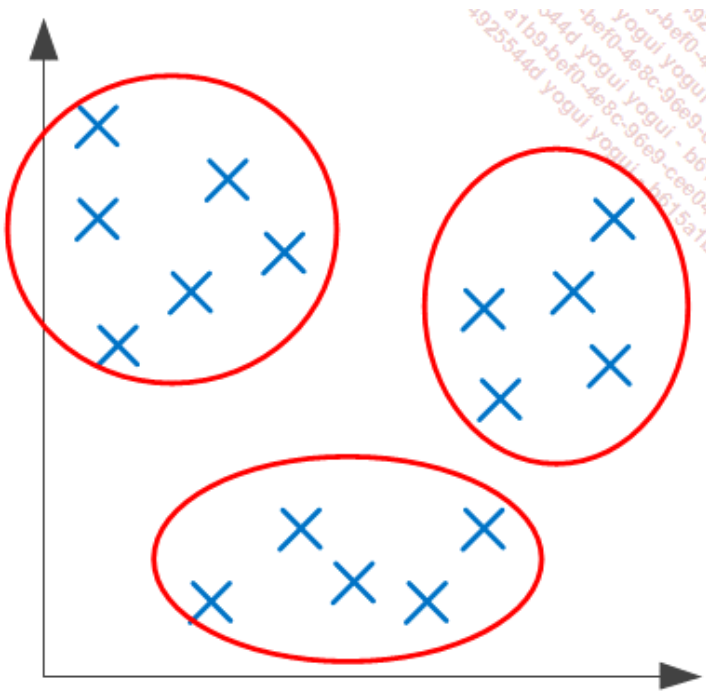
Podemos imaginar una situación donde se tiene una base de datos de clientes y en la que buscamos obtener diferentes perfiles en función de sus compras o presupuesto. A priori no se sabe cuántos perfiles hay o cuales son. Este normalmente es un problema de clustering.

Por lo tanto, vamos a buscar maximizar la coherencia de los datos dentro de una misma clase (que por ejemplo podría ser "los deportes de los 20 a los 25 años") y a minimizarla entre las clases.

Imaginemos que tenemos el siguiente conjunto de datos:



Si buscamos determinar las clases en estos datos, sería posible definir las tres siguientes:



De esta manera, se maximiza la semejanza entre los datos de una misma clase (los puntos de una clase están próximos), minimizando las semejanzas entre las clases (están separadas entre ellas).

Los algoritmos de aprendizaje no supervisado salen del marco de este libro y por lo tanto, no se presentan. Sin embargo, aquí se pueden utilizar tanto algoritmos de Machine Learning clásicos (como el algoritmo K-Means), como de redes neuronales.

b. Aprendizaje supervisado

El **aprendizaje supervisado** seguramente sea la forma de aprendizaje más habitual.

En el aprendizaje supervisado, se proporciona un conjunto de ejemplos al algoritmo de aprendizaje. Esto va a comparar la salida obtenida por la red, con la salida esperada.

Por ejemplo, se puede crear un sistema que permita estimar el precio de un apartamento a partir de un conjunto de datos. Estos ejemplos contendrán un conjunto de apartamentos con sus características (tamaño, número de habitaciones, número de baños, localización, etc.) y el precio de venta.

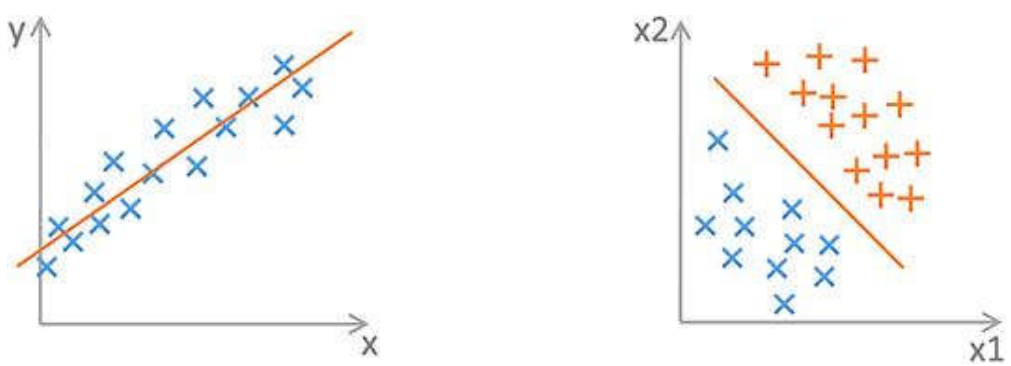
Por lo tanto, vamos a entrenar al algoritmo con los datos que conocemos, pidiéndole una estimación y comparándola con el precio real. Esta diferencia va a permitir mejorar el algoritmo hasta tener resultados satisfactorios.

Hay dos tareas principales en aprendizaje supervisado: la **regresión** y la **clasificación**.

El problema de la predicción del precio de un apartamento es un problema de **regresión**. Buscamos obtener un valor real (el precio), a partir de datos en entrada.

En el caso de la **clasificación**, el objetivo es predecir la clase de un elemento a partir de sus atributos. Solo puede haber dos clases (por ejemplo, predecir si un cliente podrá devolver su crédito o no) o varias, pero con un número limitado (por ejemplo, reconocer un número a partir de una foto de la cifra manuscrita). En este último caso, tendríamos 10 clases que se corresponderían con las cifras del 0 al 9.

En la siguiente imagen, tenemos dos ejemplos. A la izquierda queremos predecir y (eje de ordenadas) en función de x (eje de abscisas), gracias a los ejemplos (las cruces). Podríamos encontrar un modelo que se correspondiera con la línea trazada. Es una tarea de regresión. A la derecha, tenemos dos clases (las cruces y los signos de suma) que queremos separar gracias a dos características x1 y x2. También aquí, el modelo buscado podría ser la línea trazada. Sin embargo, aquí sumamos un problema de clasificación (a dos clases que queremos separar, para este ejemplo).



Las redes neuronales pueden resolver estos dos tipos de problemas. Sin embargo no son los únicos algoritmos. Presentaremos los dos principales en Machine Learning (regresión lineal y regresión logística), después la última forma de aprendizaje.

c. Aprendizaje por refuerzo

En el **aprendizaje por refuerzo**, se indica al algoritmo si la decisión tomada era correcta o no, después de algún tiempo de utilización. Por lo tanto, se proporciona una información global. Sin embargo, el algoritmo no sabe exactamente qué debería haber decidido.

Por ejemplo, así es como los animales (y los humanos), aprender a andar: sabemos qué queremos obtener (el caminar), pero no cómo obtenerlo (los músculos que se deben utilizar, con su orden). El bebé prueba a andar,

tanto si se cae (ha fallado) como si consigue dar un paso (ha tenido éxito). Por refuerzo positivo o negativo, terminará por entender lo que le permite no caerse y cada vez lo hará un poco mejor, para poder llegar a correr a continuación.

De nuevo esta forma de aprendizaje se utiliza muy poco, aparte de en algunos dominios como la robótica. En el caso de las redes neuronales, normalmente se utiliza esta forma de aprendizaje cuando se quiere obtener comportamientos complejos que hagan intervenir una sucesión de decisiones. Por ejemplo, es el caso para crear adversarios inteligentes en los videojuegos. En efecto, se busca un programa que tome diferentes decisiones que lo sitúen en una posición ganadora.

El aprendizaje por refuerzo también se puede hacer gracias a los **metaheurísticos**. En efecto, permiten optimizar las funciones sin conocimientos a priori. Sin embargo, una técnica normalmente utilizada es el uso de los **algoritmos genéticos**. Gracias a la evolución, permiten optimizar los pesos y encontrar estrategias ganadoras, sin información particular sobre lo que se esperaba.

2. Regresión y algoritmo de regresión lineal

La regresión es un problema habitual en Machine Learning. Cada vez se busca predecir un valor real (o en todo caso, con muchos valores posibles), a partir de las características de un elemento. A continuación se muestran ejemplos de datos que podríamos querer predecir en regresión:

- La cantidad de lluvia que va a haber en función de la información meteorológica.
- El consumo de un coche a partir de la información de este (como su peso o su tamaño) y sobre su motor.
- El precio de un apartamento a partir de sus características.
- Las ventas del mes próximo en función de las ventas pasadas.
- …

La técnica más sencilla en Machine Learning es la regresión lineal. En ella buscamos trazar una recta, pasando junto a un conjunto de puntos.

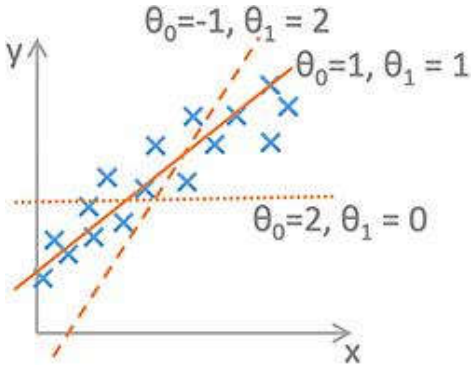
Se habla de recta en dos dimensiones, pero en realidad se trata de un plano en 3D y de un hiperplano a partir de cuatro dimensiones. Sin embargo, es más fácil entenderlo en 2D que en una dimensión superior, por la posibilidad de visualizar los datos fácilmente.

Si llamamos X a los datos en entrada (que contienen n características) e Y a los datos de salida, buscamos encontrar las variables θ tales que:

$$Y = \theta_0 + \sum_{i=1}^{n} \theta_i * X_i$$

Por lo tanto, θ contiene n+1 reales, que se corresponden con los pesos (o coeficientes) de cada una de las características, a las que se añade θ_0 que es el sesgo (o coordenada en el origen, e incluso "perturbación").

Por lo tanto, la dificultad consiste principalmente en encontrar los valores correctos para los pesos. En efecto, a continuación se muestra un ejemplo de datos para el que podríamos tener solo tres posibilidades para los pesos:



Observe que la recta punteada y la recta con trazos, no se corresponden con los datos proporcionados, mientras que la recta completa parece corresponder mucho más.

Por lo tanto, es necesario poder estimar la calidad de una posibilidad. Para esto, normalmente se utiliza el error de los mínimos cuadrados, es decir, la media de las distancias al cuadrado entre un punto y su predicción sobre la curva. De esta manera, cuanto más cerca pase la recta de los puntos, más disminuye el error. También puede utilizar las "funciones de coste" cuya formulación matemática es un poco más compleja, pero que también indica la distancia entre una recta propuesta y los puntos de aprendizaje.

De esta manera, en nuestro ejemplo, encontraríamos que el error más bajo es el de la recta completa, seguido de la recta con trazos y por último, la recta punteada.

Una vez que es posible determinar la calidad de una solución, hay un algoritmo que permite encontrar la mejor solución posible (que raramente tiene un error de 0, porque en un problema real los puntos nunca están exactamente alineados).

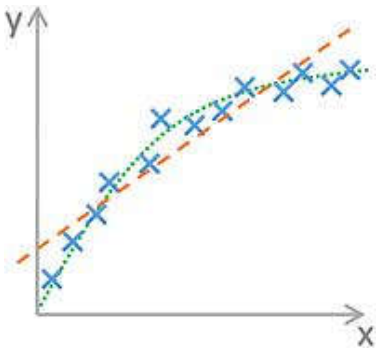
Como no existe solución sencilla ni matemática, si el número de ejemplos y de características es grande, se utiliza normalmente un algoritmo iterativo: el descenso por gradiente. Esto se presenta más en detalle en el capítulo sobre los metaheurísticos.

Lo que hay que recordar es que parte de una solución aleatoria. A continuación va a intentar mejorarla, siguiendo el ciclo siguiente hasta un criterio de parada:

- cálculo del coste de la solución actual,
- cálculo de la derivada,
- modificación de la solución actual en la siguiente derivada.

Sin embargo, este algoritmo tiene límites. El principal es que solo permite obtener rectas, mientras que algunas veces los problemas no son lineales.

A continuación se muestra un ejemplo de datos y dos soluciones propuestas. La recta (con trazos) no pasa tampoco cerca de los ejemplos que la curva (punteada). Por lo tanto, aquí una regresión lineal no se adapta bien.

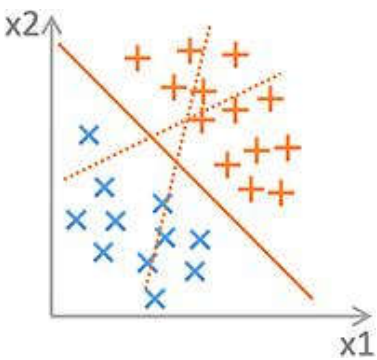


Por lo tanto, este problema no se podría resolver con una regresión lineal. Las redes neuronales no están limitadas a los problemas lineales y por lo tanto, permitirían resolver eficazmente este tipo de problemas. Además, hay que saber que la mayoría de los problemas complejos no son lineales.

En realidad, se pueden resolver problemas no lineales añadiendo características como la potencia o las multiplicaciones de las características originales. Sin embargo, en el caso en que tenemos muchos atributos de entrada, esta solución rápidamente alcanza el límite y la convergencia puede ser muy lenta.

3. Clasificación y algoritmo de separación

El problema de la clasificación es diferente: en lugar de querer encontrar una recta que pase por todos los puntos, se pretende encontrar una que separe los puntos en dos categorías.



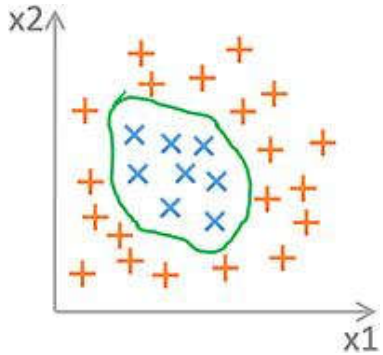
La ecuación general de la recta es la misma que para la regresión lineal, y se utilizará también el descenso por gradiente como algoritmo para mejorar una solución inicial aleatoria.

Sin embargo, en lugar de utilizar como error los mínimos cuadrados, se utilizará una función de coste, que tiene en cuenta el hecho de que un ejemplo esté o no en la categoría correcta.

Normalmente, se utiliza también una etapa adicional, transformando cada punto del espacio en un valor único y comparando este valor, con un valor límite. La función utilizada normalmente es la función sigmoidea, también llamada función logística, de la que hablaremos más adelante. Permite obtener un "grado de pertenencia" para los nuevos puntos probados, además de su clasificación.

El principio y la ecuación eran lo mismo, todavía estamos limitados. Con una regresión logística, solo se pueden resolver los problemas linealmente separables en dos clases.

A continuación se muestra un ejemplo de problema que no se puede resolver por una regresión logística clásica, porque no es linealmente separable (vemos que las cruces se sitúan en mitad de los signos de sumar):



Para solventar este límite, se pueden utilizar las redes neuronales.

Neurona formal y perceptrón

La neurona artificial, también llamada **neurona formal**, retoma el funcionamiento de la neurona biológica.

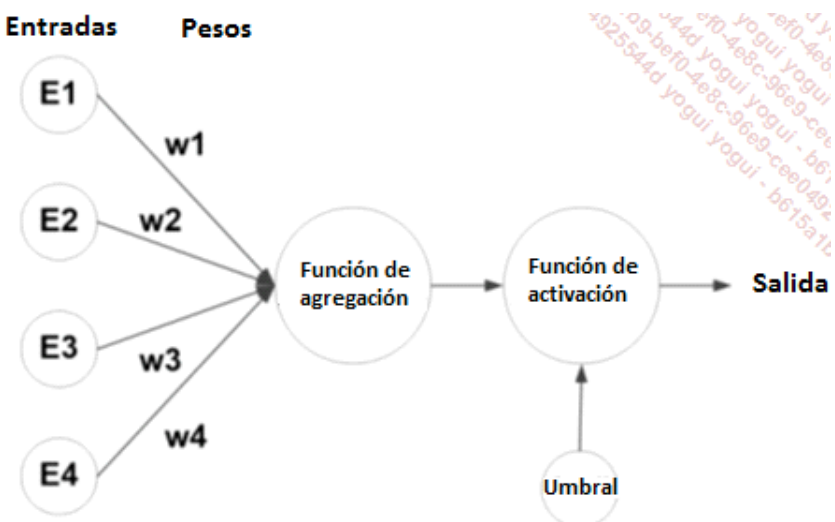
1. Principio

Una neurona recibe entradas y proporciona una salida, gracias a diferentes características:

- **Pesos** asignados a cada una de las entradas, que permita modificar la importancia de algunas respecto a otras.
- Una **función de agregación**, que permita calcular un único valor a partir de las entradas y de los pesos correspondientes.
- Un **umbral** (o sesgo), que permita indicar cuándo debe actuar la neurona.
- Una **función de activación**, que asocia a cada valor agregado un único valor de salida dependiendo del umbral.

La noción de tiempo, importante en biología, no se tiene en cuenta para la mayoría de las neuronas formales.

Por lo tanto, la neurona formal se puede resumir con la siguiente forma:



Como para la regresión lineal y la regresión logística, la principal dificultad será el aprendizaje de los pesos (y del umbral, que se puede ver como un peso particular). Las funciones de agregación y de activación se eligen a priori.

Es la presencia de la función de activación, si es no lineal, lo que permitirá solventar la condición de linealidad de los algoritmos de Machine Learning vistos anteriormente, así como el número de neuronas y sus conexiones entre ellas (porque podemos tener varias neuronas, que es el caso más habitual).

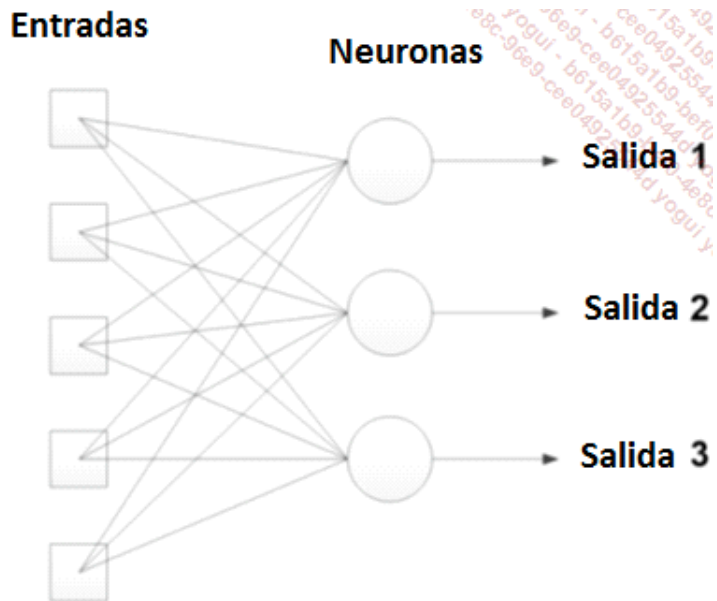
2. Red de tipo "perceptrón"

El **perceptrón** es la más sencilla de las redes neuronales. Un perceptrón es una red que contiene p neuronas formales. Cada una se relaciona con las n entradas. Esta red permite tener p salidas.

En el caso de una tarea de regresión, tendríamos una única salida (y por lo tanto, una única neurona), que dará un valor real. En el caso de una tarea de clasificación, tendremos una salida por clase y es la salida con el valor más fuerte la que indicará la clase elegida por la red.

En el caso particular donde solo tengamos dos clases, solo se utilizará una única salida y se seleccionará la clase en función del valor obtenido (en general, esta será la primera clase si el valor es inferior a 0.5 o la segunda clase en otro caso).

Por lo tanto, con tres neuronas y cinco entradas tenemos una red con tres salidas. A continuación se muestra la estructura obtenida en este caso:



Cada neurona realiza la agregación y la activación, con pesos y umbral diferente al de otras neuronas. Por lo tanto, aquí la red tiene 3 * 5 = 15 pesos a ajustar, a los que añaden tres valores de umbrales (uno por neurona), es decir 18 argumentos.

En los casos reales, las redes tienen decenas, incluso centenares de neuronas, y las entradas pueden ser millares. En efecto, en el caso de una red de neuronas que tratará una imagen para reconocer una cifra manuscrita, tendríamos tantas entradas como píxeles si tomamos una imagen en blanco y negro, y tres veces más es en RGB. De esta manera, una imagen de 64x64 píxeles se corresponde con 4.096 entradas.

Por lo tanto, el número de pesos que hay que tomar normalmente es muy grande, y es imposible determinarlos manualmente.

3. Funciones de agregación y activación

a. Función de agregación

El objetivo de esta función es transformar el conjunto de las entradas y los pesos en un único valor, que se utilizará para la función de activación. Es posible imaginar varias funciones de agregación. Las dos más habituales son:

- la suma ponderada,
- el cálculo de distancia.

En el caso de la **suma ponderada**, simplemente se va a hacer la suma de todas las entradas (E), multiplicadas por su pesos (w). Matemáticamente, esto se expresa en la forma:

$$\sum_{i=1}^{n} E_i * w_i$$

Exactamente esto es lo que ya hacen la regresión lineal o la regresión logística.

En el segundo caso, el del **cálculo de las distancias**, se va a comparar las entradas con los pesos (que son las entradas esperadas por la neurona) y calcular la distancia entre las dos.

Recuerde que la distancia es la raíz de la suma de las diferencias al cuadrado, que se expresa como sigue:

$$\sqrt{\sum_{i=1}^{n} (E_i - w_i)^2}$$

Por supuesto, se pueden imaginar otras funciones de agregación. La importante es asociar un valor único al conjunto de las entradas y los pesos gracias a una función lineal.

En la gran mayoría de las redes neuronales, se utiliza la suma ponderada.

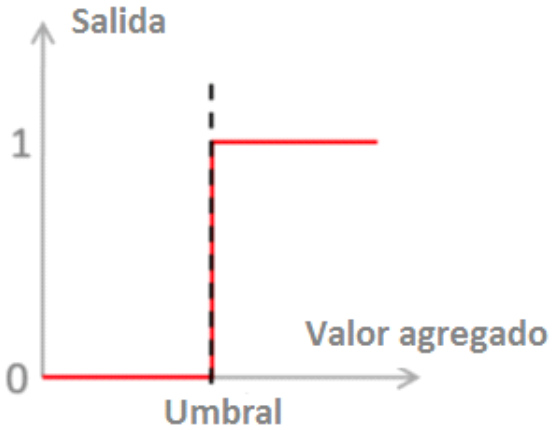
b. Función de activación

Una vez que se ha calculado un valor único, la neurona compara este valor con un umbral y decide la salida. Para esto, se pueden utilizar varias funciones. A continuación se muestran las principales.

Función "heavyside"

La función signo o **heavyside** en inglés, es una función muy sencilla: devuelve +1 o 0.

De esta manera, si el valor agregado calculado es más grande que el umbral, devuelve +1, si no 0 (o -1 según las aplicaciones).



Por ejemplo, esta función permite la clasificación, indicando que un objeto es o no en una clase dada. También se puede posicionar para otras aplicaciones, pero es difícil de utilizar, porque no indica en qué punto es fuerte un valor. Por lo tanto, los principales algoritmos de aprendizaje no funcionan con esta función.

Se utilizaba al inicio de las redes neuronales porque es muy sencilla de calcular, pero en la actualidad casi no se utiliza.

Función sigmoidea

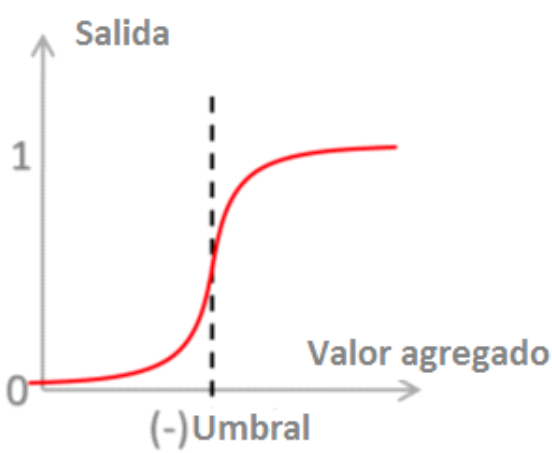
La función **sigmoidea** utiliza una exponencial. También se llama "función logística" y la encontramos cada vez más en la regresión logística (en este caso, la regresión logística se corresponde con una red de neuronas a una neurona).

Matemáticamente se define por:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Está comprendida entre 0 y 1, y vale 0,5 si x vale 0.

En la neurona, el método se llama con x = valor agregado + umbral.



Esta función permite un aprendizaje más fácil, gracias a su pendiente, sobre todo si los valores agregados están cercanos a 0. En efecto, es más fácil saber hacia qué dirección ir para mejorar los resultados, al contrario de lo que sucede con la función heavyside, que no es derivable.

La derivada de la sigmoidea, utilizada durante el aprendizaje, es:

$$f'(x) = f(x) \,.\, (1 - f(x))$$

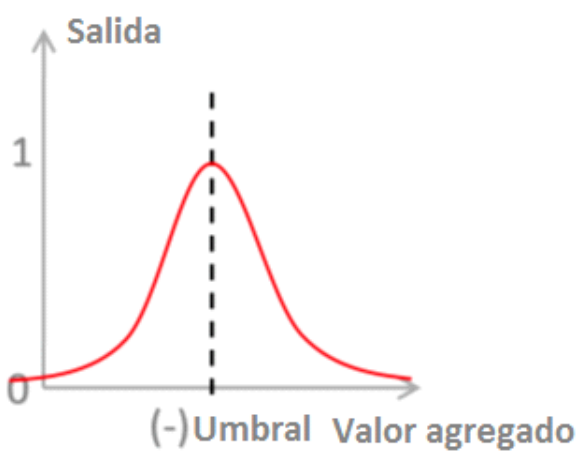
Una función muy cercana de la sigmoidea, que también la encontramos mucho, es la función tanh(x) (**tangente hiperbólica**) que es muy cercana gráficamente y en su utilización.

Función gaussiana

La siguiente es la función **gaussiana**. También llamada "campana de Gauss", es simétrica, con un máximo obtenido en 0.

Su expresión es más compleja que para la función sigmoidea, pero se puede simplificar como sigue, siendo k y k' las constantes-dependientes de la desviación estándar deseada:

$$f(x) = k.e^{-x^2/k'}$$



Se utilizara la diferencia entre el valor agregado y el umbral como abscisa. Normalmente se utiliza con la distancia como función de agregación, y únicamente en las redes particulares (llamadas RBF para *Radial BasisFunction*).

Esta función también es derivable, lo que permite un buen aprendizaje. Sin embargo, al contrario de lo que sucede con las funciones anteriores, solo tiene un efecto local (alrededor del umbral) y no sobre el espacio de búsqueda completo. Según los problemas a resolver, esto puede ser una ventaja o un inconveniente.

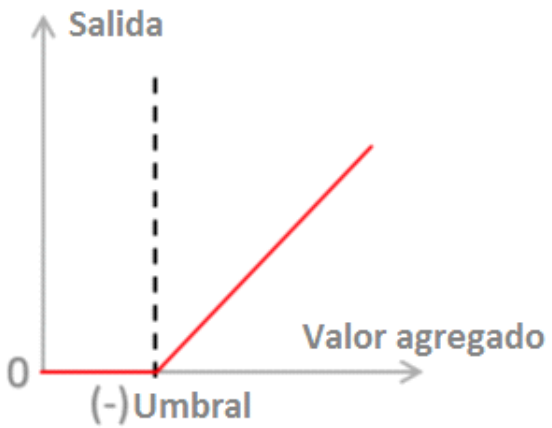
Función ReLU

Durante estos últimos años, la función ReLU (para *Rectified Linear Units*) se utiliza cada vez más. En efecto, es a la vez sencilla de calcular y sencilla de derivar, y su derivada es suficientemente importante para facilitar el aprendizaje.

Su expresión simplemente es:

$$f(x) = \max(x, 0)$$

De nuevo aquí, se agregará el umbral a la suma ponderada calculada durante la agregación, y tendremos la siguiente curva:



Por lo tanto, su derivada vale 1 si f(x) es estrictamente positivo y 0 en caso contrario.

Atención, al contrario de lo que sucede con el resto de funciones presentadas anteriormente, esta no está limitada entre 0 y 1, sino sobre el conjunto de los reales positivos. Esto puede ser muy interesante, por ejemplo para una regresión, con la condición de tenerla en cuenta correctamente durante la definición de la red.

Función Softmax

Esta última función es un poco particular. Se utiliza en clasificación cuando no es suficiente con tomar la clase que obtiene el valor más fuerte, sino que se persigue obtener las probabilidades de pertenencia a una clase.

Por lo tanto, esta función permite agregar diferentes salidas y normalizar los valores, de manera que la suma sea 1. De esta manera, cada valor de salida se corresponde directamente con el porcentaje de pertenencia.

Esta función no utiliza función de agregación y no tiene pesos. Por lo tanto, se trata de una neurona "degenerada" en el sentido que no necesita de aprendizaje.

El valor de salida vale:

$$f(x_i) = \frac{x_i}{\sum_j x_j}$$

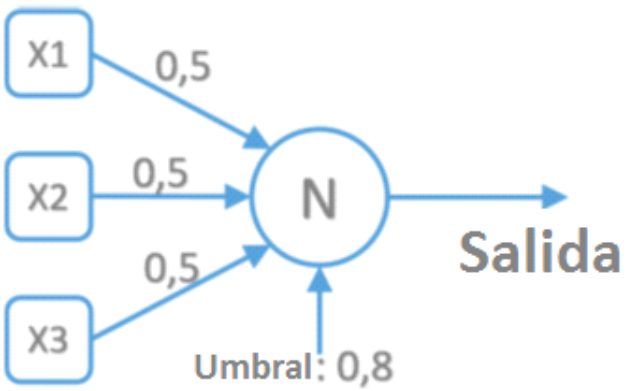
Por lo tanto, esto vuelve a calcular la suma de todas las salidas y para cada una de ellas, el porcentaje que se representa en esta suma. De esta manera, la suma de las salidas vale siempre 1.

4. Ejemplo de red

Antes de continuar y estudiar los algoritmos de aprendizaje, vamos a ver un pequeño ejemplo con una red de

neuronas muy sencilla, que contiene una única neurona.

También supondremos que las entradas son binarias, que la función de agregación es la suma ponderada y que la función de activación es una función "heavyside".



Supongamos que tenemos la entrada (1,0,1). Empezamos calculando la agregación de las entradas, es decir la suma ponderada entre las entradas y los pesos. Se obtiene:

Suma = 1 * 0,5 + 0 * 0,5 + 1 * 0,5 = 1

A continuación se compara la suma con el umbral. Aquí, la suma es superior al umbral, por lo tanto la salida vale 1.

Imaginemos ahora que la entrada vale (0,1,0). Entonces la suma valdría 0,5. Es inferior al umbral, por lo tanto en este caso la salida vale 0.

Continuamos para todos los valores de entradas posibles, y a continuación se muestra la tabla de salidas obtenidas:

X1	X2	X3	Salida
0 0	0 0	0 1	0 0
0 0	1 1	0 1	0 1
1 1	0 0	0 1	0 1
1 1	1 1	0 1	1 1

Vemos que la neurona devuelve 1 cuando hay al menos dos entradas verdaderas y 0 en caso contrario. Esta función es difícil de expresar en forma de Y u O, pero para una red de neuronas es sencillo.

5. Aprendizaje

Si seleccionar los pesos y el umbral manualmente no es posible, se utiliza un algoritmo de aprendizaje. Como para la regresión lineal y la regresión logística, se utilizará el **método de descenso por gradiente**.

Este método solo funciona para las redes de tipo perceptrón. Si hay varias salidas, se aplicará el método de descenso por gradiente a cada uno de ellos. A continuación vamos a suponer que solo tenemos una única neurona.

n es el número de entradas y m el número de ejemplos utilizados para el aprendizaje. Además, las entradas son $X = (x_1, x_2, \dots, x_{n)}$ y las salidas esperadas $Y = (y_1, y_2, \dots, y_{n)}$. Las salidas obtenidas son $S = (s_1, s_2, \dots, s_{n)}$.

Se utiliza una función de coste para definir el error, cuya definición es:

$$J = \frac{1}{m}\sum_{i=1}^{m}[-y_i \log(s_i) - (1 - y_i) log(1 - s_i)]$$

Esta función de coste es convexa, por lo tanto el descenso por gradiente convergerá hacia la solución óptima si los argumentos le eligen correctamente.

En cada pasada (una pasada = una iteración = un camino sobre el conjunto de aprendizaje), se calcula un delta sobre cada peso de la red. Esto indica la modificación que se debe hacer al final de la pasada.

Se determina como sigue:

$$dw_j = \frac{1}{m}\sum_{i=1}^{n}(s_i - y_i)x_j$$

A continuación se modifican los pesos, usando la fórmula:

$$w_j = w_j - \alpha . dw_j$$

Por lo tanto, el nuevo peso se corresponde con el antiguo, modificado el delta (que depende de las entradas y del error), multiplicado por un factor α. Este factor se llama tasa de aprendizaje. Por lo tanto, la modificación de un peso es más importante si el error es fuerte y/o si la entrada es importante.

La **tasa de aprendizaje** depende, del problema a resolver: si es demasiado baja, ralentiza mucho la convergencia. Muy grande y podría impedir encontrar la solución óptima (incluso hacer divergir el algoritmo).

Hace algunos años habitualmente se utilizaba una tasa adaptativa, fuerte al inicio del aprendizaje y que disminuía a continuación a lo largo del tiempo. Actualmente es preferible una tasa fija, bien elegida, haciendo varios intentos sobre los subconjuntos de datos para encontrar la tasa que parece óptima.

Por lo tanto, se puede resumir este algoritmo con el siguiente pseudo-código:

```
MientrasQue criterio de parada no alcanzado
   Para cada ejemplo:
       Calcular la salida si
   FinPara
```

```
   Calcular el coste total

   Para cada peso:
       Calcular los deltas para cada peso
       Modificar el peso
   FinPara
FinMientrasQue
```

El algoritmo presentado también sirve en el caso de la regresión lineal o, adaptando el cálculo del error, al de la regresión logística.

Red feed-forward

Las redes de tipo "**feed-forward**" o **en capas**, permiten superar los límites de los perceptrones. En efecto, estos ya no están limitados a los problemas linealmente separables.

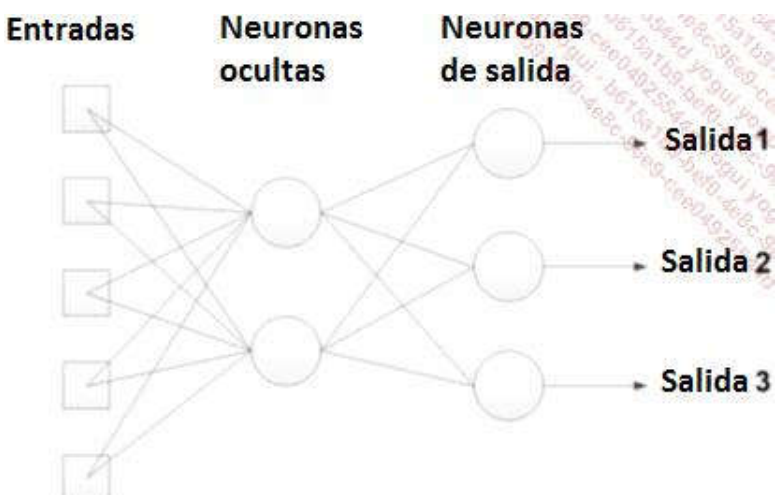
1. Red con capa oculta

Están compuestas por una o varias capas ocultas de neuronas, relacionadas con las entradas o las capas anteriores y una capa de salida, relacionada con las neuronas ocultas. Se llaman feed-forward porque la información solo puede ir desde las entradas hasta las salidas, sin volver atrás.

El número de capas ocultas y el número de neuronas de cada capa, se seleccionan por parte del usuario (normalmente haciendo pruebas sobre varios valores). Hablamos de "Deep Learning" cuando el número de capas es importante (a partir de tres o cuatro capas ocultas).

Cuantas más capas tiene la red y más complejas son, permite un mejor aprendizaje y más rápido, que con una o dos capas ocultas. En efecto, cada capa se puede ver como una etapa adicional en el algoritmo (por ejemplo, para el reconocimiento de cifras, podemos imaginar que la primera capa detecta los trazos y los bucles, que la capa siguiente permite acumular algunas características de la primera capa y que una tercera capa, deduce de todo esto el número escrito).

A continuación se muestra un ejemplo de red con cinco entradas y tres salidas, de manera que hay dos neuronas ocultas.



En este caso, hay que ajustar los pesos y umbrales de todas las neuronas ocultas (aquí 12 argumentos), así como los pesos y umbrales de las neuronas de salida (9 argumentos). Por lo tanto, el problema completo contiene 21 valores a determinar.

Las redes que utilizan neuronas de tipo perceptrón, se llaman **MLP** por *MultiLayer Perceptron*, mientras que las que utilizan neuronas con función de activación gaussiana, se llaman **RBF** (por *Radial BasisFunction*). Las redes MLP y RBF son habituales pero cada vez más, están siendo sustituidas por redes más adaptadas para las imágenes (CNN) o para las series temporales y los sonidos (SNN). Estos dos tipos de redes se ven más adelante, pero salen del ámbito de este libro.

2. Aprendizaje por retropropagación del gradiente

El algoritmo de descenso por gradiente tal y como se ha explicado anteriormente, solo funciona para los perceptrones que no tienen capa oculta.

En el caso de las redes feed-forward, existe un aprendizaje posible: por **retropropagación del gradiente** (llamado Backpropagation en ingles). Es una extensión del algoritmo de descenso por gradiente.

Por lo tanto, va a corregir en primer lugar los pesos entre las neuronas de salida y las neuronas ocultas, y después propagar el error de regreso (de ahí su nombre) y corregir los pesos entre las neuronas ocultas y las entradas. Esta corrección se hará ejemplo después de ejemplo y serán necesarias varias pasadas (con ordenes diferentes si es posible), para converger hacia una solución óptima.

Simplificamos aquí el algoritmo, suponiendo que solo tenemos una capa de neuronas ocultas. El principio es el mismo para varias capas. Es suficiente con propagar el error de vuelta a todas las capas.

También se puede aplicar de una sola vez todas las modificaciones, sobre todo si el lenguaje permite el cálculo matricial. Sin embargo, si el número de ejemplos es muy grande, normalmente es más rápido calcular las modificaciones de los pesos en cada ejemplo o por grupo de ejemplos (llamados batchs).

Supongamos que el número de ejemplos es m, h es el número de neuronas ocultas y n el número de salidas.

Por lo tanto, la primera etapa consiste en calcular la salida de cada neurona oculta llamada o_i, para un ejemplo dado. A continuación, hacemos lo mismo que para las neuronas de salida, cuya salida se calcula a partir de la suma ponderada de las salidas de la capa oculta.

Es la fase de propagación hacia adelante ("**forward propagation**").

El error cometido siempre es:

$$Error = y_i - s_i$$

Con y_i la salida esperada y s_i la obtenida.

La evaluación de la red se basa sobre el error cuadrático. Aquí suponemos que se trata de la evaluación clásica para una tarea de regresión.

Generalmente, para las tareas de clasificación se calculará el coste J, lo que modificará su derivada y por lo tanto las ecuaciones: $J = \frac{1}{m}\sum_{i=1}^{m}[-y_i \log(s_i) - (1-y_i)\log(1-s_i)]$.

A continuación pasamos a la fase de propagación hacia atrás ("**backward propagation**"). La primera etapa consiste en calcular los deltas de las neuronas de salida. Para esto, se calcula para cada una la derivada de la función de activación multiplicada por el error, que se corresponde con nuestro delta.

$$\delta_i = s_i . (1 - s_i). (y_i - s_i)$$

A continuación es necesario hacer lo mismo para las neuronas ocultas, relacionadas cada una con las k neuronas de

salida. Por lo tanto, el cálculo es:

$$\delta_i = o_i \,.\, (1 - o_i) . \sum_k \delta_k * w_{i \to k}$$

En efecto, este cálculo tiene en cuenta la corrección de las neuronas de salida (δ_k) y el peso que los relaciona.

Además, también aquí, cuando más importante sean los pesos, más fuerte será la corrección a aplicar.

Cuando se calculan todos los deltas, los pesos se pueden modificar haciendo el siguiente cálculo, donde solo el último valor cambia según se trate de una neurona oculta (tomamos la entrada) o de una neurona de salida (tomamos su entrada, es decir, la salida de la neurona oculta):

$$w_i = w_i + \tau \,.\, \delta_i \,.\, (x_i \; OU \; o_i)$$

Por lo tanto, el pseudo-código es el siguiente:

```
MientrasQue criterio de parada no alcanzado
   Inicializar los di

   Para cada ejemplo:
         Calcular la salida si

         Para cada peso de las neuronas de salida:
               di = si * (1 - si) * (yi - si)
         FinPara

         Para cada peso de las neuronas ocultas:
               sum = 0
               Para cada enlace hacia la neurona de salida k:
                     sum += dk * wi hacia k
               FinPara
               di = oi * (1 - oi) * sum
         FinPara

         Para cada peso de la red:
               Si enlace hacia neurona de salida:
                     wi += tasa * di * oi
               En caso contrario
                     wi += tasa * di * si
               FinSi
        FinPara
   FinPara


   Si necesario, modificación de la tasa
```

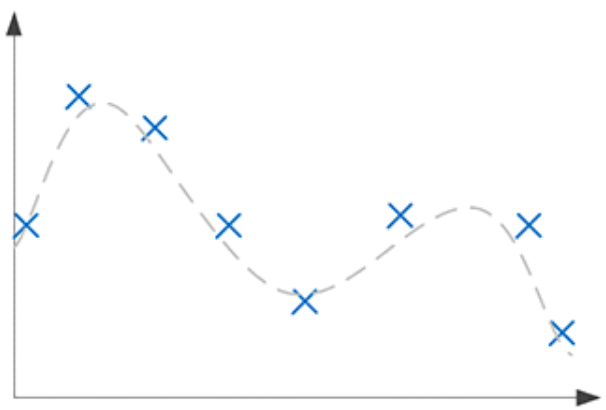
```
FinMientrasQue
```

3. Sobre-aprendizaje

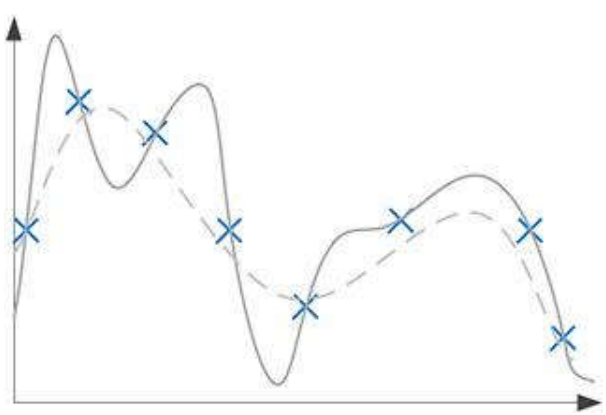
La red aprende a partir de los datos que se le proporcionan y va a encontrar una función global, que permita limitar sus errores. Al inicio, el error será importante y después disminuirá para cada camino de datos de ejemplo y ajuste de los pesos.

Sin embargo, a partir de un determinado umbral, la red va a aprender los puntos proporcionados y perder completamente en **generalización**, sobre todo si los datos proporcionados son ligeramente erróneos o poco numerosos: entonces tenemos **sobre-aprendizaje** (u **over-fitting** en inglés).

A continuación se muestra por ejemplo un problema sencillo, donde hay que encontrar la función que generaliza los puntos dados. Una correcta solución se proporciona como línea punteada.



Cuando aparece el sobre-aprendizaje, entonces nos podemos encontrar con una función de este tipo, que pasa por los puntos (por lo tanto, el error global es nulo), pero pierde completamente en generalización:



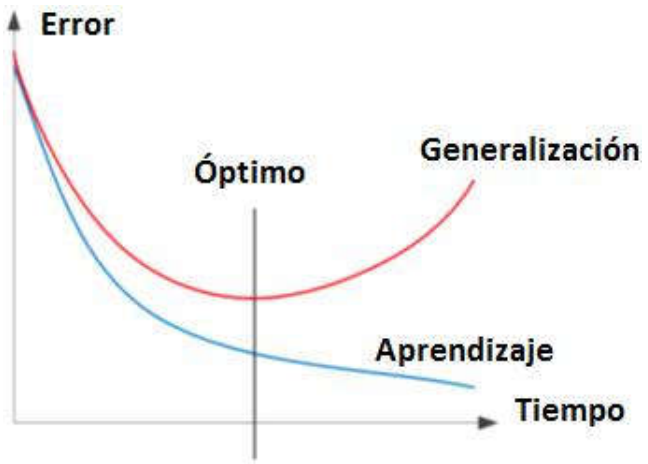
Por lo tanto, es necesario no solo evaluar la calidad del aprendizaje, sino también la capacidad de generalización de la red.

Para evitar el sobre-aprendizaje o al menos detectarlo, vamos a separar nuestro conjunto de datos en tres sub-conjuntos.

El primero es el **conjunto de aprendizaje**. Este es el más importante y normalmente contiene el 60% de los ejemplos. Sirve al algoritmo de aprendizaje para adaptar los pesos y umbrales de la red.

El segundo conjunto contiene entorno a 20% de los ejemplos. Es el **conjunto de generalización**. Al final de cada pasada, se prueba el error global sobre este conjunto (que no se ha utilizado para cambiar los pesos). Nos indica en qué momento aparece el sobre-aprendizaje.

En efecto, si se traza a lo largo del tiempo el error medio del conjunto de aprendizaje y del conjunto de validación, se obtienen las siguientes curvas:



El error del conjunto de aprendizaje baja a lo largo del tiempo. Por el contrario, si en un primer momento el error de la generalización baja, comienza a aumentar cuando el sobre-aprendizaje empieza. Por lo tanto, es en este momento cuando hay que parar el aprendizaje.

El último conjunto es el **conjunto de validación**. Es el que permite determinar la calidad de la red, para comparar por ejemplo varias arquitecturas (como un número de neuronas ocultas diferente). Los ejemplos de este conjunto no se verán por parte de la red hasta que el aprendizaje haya terminado, por lo tanto no intervienen del todo en el proceso.

Cuando tenemos muchos datos de aprendizaje (caso del big data), no se respetarán estos ratios (60-20-20), pero intentaremos tener hasta el que sea necesario en los conjuntos de generalización y de pruebas, para conservar un máximo de ejemplos de aprendizaje. De esta manera, se puede tener un reparto de tipo 98-1-1.

Normalmente hablamos de compromiso sesgo-varianza. En efecto, el error del conjunto de aprendizaje se llama **sesgo**. El error situado entre el conjunto de aprendizaje y el conjunto de validación se llama **varianza**. Observe que durante la fase de aprendizaje el sesgo disminuye, mientras que la varianza aumenta ligeramente. Al contrario, en la fase de sobre-aprendizaje el sesgo disminuye poco, pero la varianza aumenta ligeramente.

4. Mejoras del algoritmo

Hay muchas mejoras posibles en el algoritmo de descenso por gradiente, incluso si siempre es la base de los cálculos. Estas mejoras permiten facilitar la convergencia acelerando el aprendizaje y limitando la varianza.

Aquí solo se presentan las principales mejoras.

a. Batch, mini-batch y gradiente estocástico

Cuando se aplica la modificación de los pesos al final de la fase total de los ejemplos de aprendizaje, el algoritmo se llama "**batch gradient descent**". Permite evitar las oscilaciones en la función de coste.

Sin embargo, cuando el conjunto de aprendizaje es muy grande (por ejemplo de millones de ejemplos) o si los ejemplos se van conociendo a medida que avanza el aprendizaje (aprendizaje "live"), entonces la convergencia se hace imposible o en todo caso, difícil.

Por lo tanto, una variante se llama **descenso por gradiente estocástico**. Consiste en aplicar la variación de los pesos a cada ejemplo presentado en la red. Esto parece aumentar mucho la convergencia y permite el aprendizaje live. Sin embargo, como las variaciones se aplican a cada ejemplo, la función de coste varía mucho y es difícil seguir el funcionamiento del aprendizaje.

Por lo tanto, cada vez más se utiliza un compromiso entre estos dos extremos. Se trata del **mini-batch**, en el que se aplica las variaciones después de un número definido de ejemplos (que forman un mini-batch). Cuando todos los ejemplos se han presentado al mismo tiempo, podremos recrear los mini-batchs diferentes, modificando el orden de los ejemplos.

De esta manera, se aumenta la velocidad de convergencia como para el gradiente estocástico, y se limitan las fluctuaciones en la función de coste durante el aprendizaje.

b. Regularización

La **regularización** permite limitar el sobre-aprendizaje. Para esto, se añade un factor a la función de coste que se corresponde con una proporción dada de los pesos. De esta manera, el algoritmo debe aprender los ejemplos conservando los pesos bajos, lo que tiene por efecto hacerlo más general.

Normalmente, a la proporción que se tiene en cuenta se le llama λ. Si λ vale 0, entonces no hay regularización.

Atención: si λ se hace demasiado grande, se hace imposible cualquier convergencia y el algoritmo intentará tener pesos bajos. Por lo tanto, es necesario hacer pruebas en función del problema a resolver.

c. Dropout

El **dropout** es una técnica contra-intuitiva. En efecto, consiste en eliminar las neuronas de manera aleatoria sobre algunas capas antes del aprendizaje. Las neuronas eliminadas cambian en cada pasada por los ejemplos.

De esta manera, se comprueba que las redes son más robustas y por lo tanto, más genéricos, lo que limita mucho el riesgo de sobre-aprendizaje.

d. Variación del algoritmo de descenso por gradiente

Aparecen variaciones del algoritmo, que modifican los pesos en función de otros factores y que permita acelerar la convergencia hacia la solución óptima. Normalmente, estos algoritmos tienen en cuenta las modificaciones pasadas para crear una "inercia" y por lo tanto, tener una convergencia que va a continuar en la dirección correcta, incluso acelerar.

Se diferencian del cálculo de la modificación adicional, así como de la manera en que se gestiona el final de la convergencia, cuando el algoritmo se debe ralentizar para converger hacia la solución óptima local o global concretamente.

Por ejemplo, se puede mencionar Adam, RMSProp, Adagrad e incluso la utilización de los impulsos.

e. Creación de nuevos datos: data augmentation

Cuanto más numerosos sean los datos de aprendizaje, mejores resultados se pueden obtener (gracias a un mejor aprendizaje sobre el conjunto de aprendizaje y a una mejor generalización que da por lo tanto, mejores

resultados).

Sin embargo, algunas veces sucede que es difícil obtener los datos. En este caso, hay que crear algunas veces nuevos datos a partir de los datos existentes. También se pueden crear datos por todas partes con una simulación.

Tomemos un ejemplo de reconocimiento de matrículas. Podemos imaginar varias posibilidades para obtener nuevas imágenes:

- Hacer girar las imágenes y deformarlas.
- Modificar el tamaño de estas, si las matrículas pueden tener varios tamaños.
- Modificar la luminosidad y el contraste de las matrículas existentes, conservando valores normales para las condiciones de utilización de la aplicación.
- Crear nuevas placas a partir de un software que cree imágenes, teniendo varios formatos de matrícula y varios textos dentro, incluso cambiando los tipos de letra.
- Etc.

Sin embargo preste atención: agregar un ruido aleatorio a las imágenes normalmente no da buenos resultados. Al contrario, puede ayudar al sobre-aprendizaje, presentando ejemplos muy parecidos entre ellos desde el punto de vista del algoritmo de aprendizaje.

Otras arquitecturas

Las redes feed-forward se utilizan mucho, pero estas no son las únicas redes. Existen múltiples arquitecturas que permiten trabajar sobre diferentes problemas.

No entraremos en detalle sobre sus implementaciones, pero vamos en presentar los aspectos principales.

1. Red de neuronas con consolación

Las **redes neuronales consolativas** (o con consolación, identificadas con CNN en inglés), están adaptadas para trabajar sobre las imágenes. En efecto, los píxeles no son completamente independientes y normalmente es útil trabajar sobre zonas de la imagen.

Estas redes están compuestas por múltiples capas, que pueden tener funciones diferentes. En particular, se distinguen las capas de consolación, que tratan las partes de la imagen (es necesario verlas como extractores de características), las capas de pooling (que combinan las salidas de las capas de convolución para detectar funcionalidades de más alto nivel) y las capas más clásicas de clasificación (normalmente con la función ReLU como función de activación).

Sin embargo, estas redes son muy difíciles de entrenar partiendo de 0. Generalmente se utiliza una red conocida pre-entrenada, y solo haremos el entrenamiento sobre nuestro dataset sobre algunas generaciones. Esta utilización de una red existente, se llama *transfer learning* o **aprendizaje por transferencia**.

De esta manera se llega a resultados superiores al 95% o 99% de clasificaciones correctas, con un tiempo de aprendizaje que permanece bajo (del orden de una hora).

2. Mapas de Kohonen

Los **Mapas de Kohonen**, o **Mapas auto-adaptativos**, contienen una matriz de neuronas. A lo largo del tiempo, cada neurona se va a asociar a una zona del espacio de entrada, desplazándose por la superficie de esta.

Cuando el sistema se estabiliza, el reparto de las neuronas se corresponde con la topología del espacio. Por lo tanto, de esta manera se puede hacer un reparto discreto del espacio.

Sin embargo, estos mapas no se pueden utilizar en las aplicaciones comerciales por su complejidad de implantación.

3. Red de neuronas recurrentes

En una **red de neuronas recurrentes** no solo hay enlaces de una capa a las siguientes, sino también hacia las capas anteriores.

De esta manera, la información tratada en una etapa se puede utilizar para el tratamiento de las siguientes entradas.

Esto permite tener sucesiones de valores en la salida que son dependientes, como una serie de instrucciones para un robot o un efecto de memorización del paso del tiempo pasado.

Sin embargo, estas redes son difíciles de ajustar. De hecho, el efecto temporal complica los algoritmos de aprendizaje, y la retro-propagación no puede funcionar como tal. Sin embargo, estos últimos años, se han hecho grandes progresos en este dominio, en particular con de las variantes llamadas GRU (*Gated Recurrent Unit*) y LSTM (*Long Short-Term Memory*).

En la actualidad, las redes neuronales recurrentes se utilizan masivamente en todas las aplicaciones que tratan datos secuenciales, como el reconocimiento y el tratamiento de la palabra, la música, las traducciones e incluso los chatbots que encontramos cada vez más habitualmente en los sitios web.

4. Red de Hopfield

Las **redes de Hopfield** son redes completamente conectadas: cada neurona está relaciona con el resto.

Cuando introducimos una entrada en la red, solo se modifica el estado de una neurona al mismo tiempo, hasta la estabilización de la red. Por lo tanto, el estado estable es la "firma" de la entrada.

El aprendizaje consiste en determinar los pesos de manera que las entradas diferentes produzcan estados estables diferentes, pero entradas casi idénticas conduzcan al mismo estado.

De esta manera, si los errores manchan ligeramente una entrada, esta se reconocerá por la red. De esta manera, se puede imaginar un sistema que permita el reconocimiento de las letras, incluso si estas están dañadas o son menos legibles.

El aprendizaje en estas redes se hace gracias a una variante de la **ley de Hebb**. Esto indica que hay que reforzar la conexión entre dos neuronas si están activas al mismo tiempo, y reducir el peso en caso contrario.

Dominios de aplicación

Las redes neuronales se utilizan en muchos dominios. Son una técnica muy correcta cuando los criterios siguientes se cumplen:

- Hay muchos ejemplos disponibles para el aprendizaje, y además es posible crear uno fácilmente.
- No existen relaciones conocidas entre las entradas y las salidas expresadas por las funciones.
- La salida es más importante que la manera de obtenerla, las redes neuronales no permiten tener una explicación sobre el proceso utilizado internamente.

1. Reconocimiento de patterns

La tarea más habitual que se asigna a las redes neuronales, es el **reconocimiento de patterns**.

En esta tarea, diferentes patterns se presentan a la red durante el aprendizaje. Cuando se deben clasificar nuevos ejemplos, entonces se puede reconocer los motivos: se trata de una tarea de **clasificación**.

De esta manera, las redes neuronales pueden reconocer los caracteres manuscritos o las formas. Las aplicaciones permiten leer las matrículas en una imagen, incluso en presencia de defectos de iluminación sobre la matrícula en sí misma.

2. Estimación de funciones

La **estimación de funciones** o **regresión** consiste en asignar un valor numérico a partir de entradas, generalizando la relación existente entre ellas. Las entradas pueden representar características o series temporales, según las necesidades.

De esta manera, son posibles las aplicaciones en medicina. Existen redes neuronales que toman como entrada características pertenecientes a radios de la mano y la muñeca y son capaces de determinar la severidad de la artrosis, una enfermedad que afecta a las articulaciones.

También es posible en finanzas determinar la buena salud bancaria de un individuo para asociarle una "puntuación de crédito", que indique si la aceptación de un crédito es arriesgada o no. En bolsa, las redes neuronales permiten estimar los mercados de valores e/o indicar los valores que parecen prometedores.

3. Creación de comportamientos

Si la red lo permite, se trata de un **comportamiento** que podrá ser definido por esta. Las aplicaciones en robótica y en la industria son numerosas.

De esta manera es posible controlar un vehículo autónomo, que recibiría como entrada la información del entorno y como salida proporciona las órdenes de desplazamiento.

Otros estudios dan la posibilidad de controlar robots, permitiéndoles aprender trayectorias o series de acciones.

Alstom, por ejemplo, los utiliza también para controlar de manera más eficaz los procesos industriales complejos.

4. Aplicaciones actuales

Todas las grandes empresas (o casi) del mundo digital, actualmente hacen Deep Learning sobre diferentes tareas. Las empresas cuyo núcleo de negocio no sea el entorno digital también se lanzan, porque esto les permite mejorar sus procesos actuales y proporcionar nuevos servicios a sus clientes (normalmente a través de objetos conectados para recuperar los datos).

Además, hay disponibles **frameworks** (como TensorFlow, MxNet o Keras) y también **plataformas** "listas para su uso" en los clouds de Amazon (AWS Machine Learning), Google o Microsoft Azure, por ejemplo.

Google empezó a utilizar el Deep Learning en 2011 con el proyecto Google Brain. Su primer proyecto era una red de neuronas capaz de reconocer chats sobre vídeos de YouTube. Tuvieron éxito y publicaron sus resultados en 2012. Posteriormente compraron Deep Mind (en 2014) y de esta manera consiguieron crear un programa capaz de batir a un jugador de Go, el juego considerado como el más difícil de aprender (más incluso que el ajedrez).

Estos proyectos han sido muy mediáticos, pero el Deep Learning y las redes neuronales también se utilizan mucho en los sistemas de **recomendación** (para la publicidad como Google Ad o para las ventas como en Amazon).

También lo encontramos en los sistemas de **reconocimiento facial**. Algunos algoritmos llegan incluso a reconocer a una persona que ocultara su cara con gafas, sobrero o bufanda.

Los teléfonos también hacen un uso masivo para el reconocimiento de voz de los interlocutores. Utilizan modelos pre-entrenados anteriormente y habitualmente situados en los servidores en el cloud. Sin embargo, cada vez más fabricantes (entre ellos Apple con el iPhone X), instalan chips dedicados en los teléfonos para el aprendizaje en tiempo real.

Implementación

Los MLP (*MultiLayer Perceptron*) son redes muy utilizadas. Son redes feed-forward, con neuronas de tipo perceptrón. La función de agregación es una suma ponderada y la función de activación una sigmoidea, lo que permite un aprendizaje por retro-propagación. Sobre todo los encontramos en regresión.

La red codificada aquí tiene una única capa oculta. El número de neuronas de las diferentes capas, así como el número de entradas, son parametrizables.

A continuación se presentan dos problemas:

- El problema del XOR (O exclusivo), que es sencillo de resolver y permite probar si los algoritmos funcionan.
- El problema "Abalone", que es de tipo regresión y se utiliza mucho para comparar algoritmos de aprendizaje.

La aplicación es en Java y solo utiliza librerías estándares, de manera que sea fácilmente reutilizable.

1. Puntos y conjuntos de puntos

Los problemas utilizados con las redes neuronales, necesitan muchos puntos para el aprendizaje. Por lo tanto, no es conveniente introducirlos a mano en el código.

Por lo tanto, se utilizarán archivos de texto con tabulaciones como separadores.

La primera clase es **`PuntoND`**, que se corresponde con un ejemplo (que contiene N dimensiones). Este contiene una tabla de valores considerados como entradas y una tabla de valores de salida. Hacemos estos atributos públicos, pero se declaran `final`. De esta manera, no son modificables.

El comienzo de la clase es la siguiente:

```
public class PuntoND {
   public final doble[] entradas;
   public final doble[] salidas;

   // Constructor aquí
}
```

El constructor recibe como argumentos la cadena que se corresponde con la línea del archivo de texto y el número de salidas de los ejemplos (los valores son tanto las entradas como las salidas). Inicialmente, el contenido se separa de los caracteres que se corresponde con la tecla tabulación ('\t'), gracias a la función `split`. A continuación, las entradas y las salidas se transforman en nombres reales.

```
   public PuntoND(String str, int _numSalidas) {
       String[] contenido = str.split("\t");
       entradas = new doble[contenido.length - _numSalidas];
       for (int i = 0; i < entradas.length; i++) {
           entradas[i] = Double.parseDouble(contenido[i]);
       }
       salidas = new doble[_numSalidas];
       for (int i = 0; i < _numSalidas; i++) {
           salidas[i] =
```

```
Double.parseDouble(contenido[entradas.length + i]);
        }
    }
```

La segunda clase es **ColeccionPuntos**, que se corresponde con el conjunto de puntos de ejemplo. Estos estarán separados en un conjunto de aprendizaje (`ptsAprendizaje`) y un conjunto de generalización (`ptsGeneralizacion`) que permita detectar el sobre-aprendizaje.

La base de la clase es la siguiente:

```
import java.util.ArrayList;
import java.util.Random;

public class ColeccionPuntos {
   protected PuntoND[] ptsAprendizaje;
   protected PuntoND[] ptsGeneralizacion;

   // Métodos aquí
}
```

Se añaden dos métodos, que permiten recuperar los puntos de aprendizaje y los puntos de generalización.

```
   PuntoND[] getPtsAprendizaje() {
       return ptsAprendizaje;
   }

   PuntoND[] getPtsGeneralizacion() {
       return ptsGeneralizacion;
   }
```

El último método es el constructor. Recibe como argumentos la cadena que se corresponde con la integralidad del archivo en forma de tabla (una línea por casilla), el número de salidas y el ratio que se corresponde con los puntos de aprendizaje. Por ejemplo, 0.8 indica que el 80% de los puntos se utilizan para el aprendizaje y por lo tanto, el 20% para la generalización.

Las etapas son las siguientes:

- Los puntos se leen y se crea uno por uno a partir de su contenido.
- El conjunto de aprendizaje se crea tomando el número de ejemplos necesario. Estos se eligen de manera aleatoria entre los puntos restantes.
- Para terminar, el conjunto de generalización se crea a partir de los ejemplos todavía no seleccionados.

Por lo tanto, el código es el siguiente:

```
   public ColeccionPuntos(String[] _contenido, int _numSalidas,
doble _ratioAprendizaje) {
       // Lectura del archivo total
       int numLineas = _contenido.length;
       ArrayList<PuntoND> puntos = new ArrayList();
```

```
        for (int i = 0; i < numLineas; i++) {
            puntos.add(new PuntoND(_contenido[i], _numSalidas));
        }

        // Creación de los puntos de aprendizaje
        int numPtsAprendizaje = (int) (numLineas *
_ratioAprendizaje);
        ptsAprendizaje = new PuntoND[numPtsAprendizaje];
        Random generador = new Random();
        for (int i = 0; i < numPtsAprendizaje; i++) {
            int indices = generador.nextInt(puntos.size());
            ptsAprendizaje[i] = puntos.get(indice);
            puntos.remove(indice);
        }

        // Creación de los puntos de generalización
        ptsGeneralizacion = (PuntoND[]) puntos.toArray(new
PuntoND[puntos.size()]);
    }
```

La lectura del archivo se hará en el programa principal, porque esta depende de la plataforma elegida y del origen de los archivos (almacenamiento local, en línea, acceso por servicio web, etc…).

2. Neurona

La base de nuestra red es la neurona, codificada en la clase **`Neurona`**. Esta tiene dos atributos:

- La tabla de pesos que la relacionan con las diferentes entradas y el sesgo (que será el último valor).
- La salida, que es un número real y que se guardará, porque sirve para el aprendizaje y evitará tener que recalcularlo. También tiene un descriptor de acceso.

```
import java.util.Random;

public class Neurona {
    protected doble[] pesos;
    protected doble salida;

    public doble getSalida() {
        return salida;
    }

    // Métodos aquí
}
```

Nunca es útil recuperar todos los pesos, sino únicamente un peso particular. Para esto, el método `getPesos` devuelve el que se corresponde con el índice solicitado. Además, el método `setPesos` modifica el valor de un peso dado, lo que es necesario para el aprendizaje.

```
    public doble getPesos(int indice) {
        return pesos[indice];
```

```
    }

    public void setPesos(int indice, doble valor) {
        pesos[indice] = valor;
    }
```

El constructor recibe como argumento el número de entradas de esta célula. En efecto, cada neurona, ya esté oculta o sea de salida, tiene un número de pesos que se corresponde con el número de entradas más el sesgo. Los pesos se inicializan de manera aleatoria entre -1 y +1.

Normalmente se aconseja empezar con los pesos más bajos y según los problemas, podría ser interesante modificar los valores iniciales dividiéndolos por 10, 50 o 100, por ejemplo.

La salida se inicializa a NaN (*Not A Number*), de manera que se pueda diferenciar el hecho de que se haya calculado o no.

```
    public Neurona(int _numEntradas) {
        salida = Double.NaN;

        Random generador = new Random();
        pesos = new doble[_numEntradas + 1];
        for (int i = 0; i < _numEntradas + 1; i++) {
            pesos[i] = generador.nextDouble() * 2.0 - 1.0;
        }
    }
```

El método de evaluación recibe como argumento una tabla de valores. Si la salida todavía no se ha calculado, empezamos haciendo la suma ponderada de los pesos multiplicados por las entradas y después, se calcula la salida utilizando una sigmoidea como función de activación.

```
    protected doble Evaluar(doble[] entradas) {
        if (Double.isNaN(salida)) {
            doble x = 0.0;
            int numEntradas = entradas.length;
            for (int i = 0; i < numEntradas; i++) {
                x += entradas[i] * pesos[i];
            }
            x += pesos[numEntradas];
            salida = 1.0 / (1.0 + Math.exp(-1.0 * x));
        }

        return salida;
    }
```

Un segundo método de evaluación funciona a partir de un PuntoND. Lo único que hace es llamar al método anterior:

```
    protected doble Evaluar(PuntoND punto) {
        return Evaluar(punto.entradas);
    }
```

El último método permite reinicializar la salida, de manera que se pueda tratar un nuevo ejemplo.

```
   protected void Eliminar() {
       salida = Double.NaN;
   }
```

3. Red de neuronas

Estando las neuronas implementadas, es posible pasar a la red completa, en una clase **RedNeuronas**.

En primer lugar, esta contiene cinco atributos:

- una tabla con las neuronas ocultas `neuronasOcultas`,
- una tabla con las neuronas de salida `neuronasSalida`,
- tres enteros que indican el número de entradas, neuronas ocultas y salidas de la red.

```
public class RedNeuronas {
   protected Neurona[] neuronasOcultas;
   protected Neurona[] neuronasSalida;
   protected int numEntradas;
   protected int numOcultas;
   protected int numSalidas;

   // Resto de la clase aquí
}
```

El constructor recibe el número de entradas, neuronas ocultas y salidas como argumentos. Entonces se crean las neuronas (capa oculta y capa de salida).

```
   public RedNeuronas(int _numEntradas, int _numOcultas, int
_numSalidas) {
       numEntradas = _numEntradas;
       numOcultas = _numOcultas;
       numSalidas = _numSalidas;

       neuronasOcultas = new Neurona[numOcultas];
       for (int i = 0; i < numOcultas; i++) {
           neuronasOcultas[i] = new Neurona(_numEntradas);
       }

       neuronasSalida = new Neurona[numSalidas];
       for (int i = 0; i < numSalidas; i++) {
           neuronasSalida[i] = new Neurona(numOcultas);
       }
   }
```

El siguiente método es el que permite evaluar la salida para un ejemplo dado. Para esto, lo primero es reinicializar

las salidas de las diferentes neuronas. A continuación se calcula la salida de cada neurona oculta y después, la de las neuronas de salida. El método termina devolviendo la tabla de las salidas obtenidas.

```
   protected doble[] Evaluar(PuntoND punto) {
       // Eliminamos la salida anterior
       for (Neurona n: neuronasOcultas) {
           n.Eliminar();
       }
       for (Neurona n: neuronasSalida) {
           n.Eliminar();
       }

       // Cálculo de las salidas de las neuronas ocultas
       doble[] salidasOcultas = new doble[numOcultas];
       for (int i = 0; i < numOcultas; i++) {
           salidasOcultas[i] = neuronasOcultas[i].Evaluar(punto);
       }

       // Cálculo de las salidas de las neuronas de salida
       doble[] salidas = new doble[numSalidas];
       for (int i = 0; i < numSalidas; i++) {
           salidas[i] =
neuronasSalida[i].Evaluar(salidasOcultas);
       }

       return salidas;
   }
```

El último método y el más complejo es el que permite ajustar los pesos de la red, gracias al algoritmo de retro-propagación. Como argumentos recibe el punto comprobado y la tasa de aprendizaje.

La primera etapa consiste en calcular los deltas para cada neurona de salida, en función de la fórmula vista anteriormente. A continuación, se calcula el delta de las neuronas ocultas. Para terminar, se actualizan los pesos de las neuronas de salida (sin olvidar su sesgo) y el de las neuronas ocultas (de nuevo con su sesgo). Se utiliza aquí un descenso estocástico, lo que significa que los pesos se actualizan para cada punto (y no en cada pasada).

```
   protected void AjustarPesos(PuntoND punto, doble
tasaAprendizaje) {
       doble[] deltasSalida = CalcularDeltasSalida(punto);
       doble[] deltasOcultos =
CalcularDeltasOcultas(deltasSalida);
       AjustarPesosSalida(deltasSalida, tasaAprendizaje);
       AjustarPesosOcultas(deltasOcultos, tasaAprendizaje, punto);
   }

   private doble[] CalcularDeltasSalida(PuntoND punto) {
       doble[] deltasSalida = new doble[numSalidas];
       for (int i = 0; i < numSalidas; i++) {
           doble salidaObtenida = neuronasSalida[i].salida;
           doble salidaEsperada = punto.salidas[i];
           deltasSalida[i] = salidaObtenida * (1 - salidaObtenida)
* (salidaEsperada - salidaObtenida);
       }
```

```
        return deltasSalida;
    }

    private doble[] CalcularDeltasOcultas(doble[] deltasSalida) {
        doble[] deltasOcultos = new doble[numOcultas];
        for (int i = 0; i < numOcultas; i++) {
            doble salidaObtenida = neuronasOcultas[i].salida;
            doble suma = 0.0;
            for (int j = 0; j < numSalidas; j++) {
                suma += deltasSalida[j] *
neuronasSalida[j].getPesos(i);
            }
            deltasOcultos[i] = salidaObtenida * (1 - salidaObtenida) *
somme;
        }
        return deltasOcultos;
    }

    private void AjustarPesosSalida(doble[] deltasSalida, doble
tasaAprendizaje) {
        doble valor;
        for (int i = 0; i < numSalidas; i++) {
            Neurona neuronaSalida = neuronasSalida[i];
            for (int j = 0; j < numOcultas; j++) {
                valor = neuronaSalida.getPesos(j) + tasaAprendizaje
* deltasSalida[i] * neuronasOcultas[j].getSalida();
                neuronaSalida.setPesos(j, valor);
            }
            valor = neuronaSalida.getPesos(numOcultas) +
tasaAprendizaje * deltasSalida[i] * 1.0;
            neuronaSalida.setPesos(numOcultas, valor);
        }
    }

    private void AjustarPesosOcultas(doble[] deltasOcultos, doble
tasaAprendizaje, PuntoND punto) {
        doble valor;
        for (int i = 0; i < numOcultas; i++) {
            Neurona neuronaOculta = neuronasOcultas[i];
            for (int j = 0; j < numEntradas; j++) {
                valor = neuronaOculta.getPesos(j) + tasaAprendizaje
* deltasOcultos[i] * punto.entradas[j];
                neuronaOculta.setPesos(j, valor);
            }
            valor = neuronaOculta.getPesos(numEntradas) +
tasaAprendizaje * deltasOcultos[i] * 1.0;
            neuronaOculta.setPesos(numEntradas, valor);
        }
    }
```

Ahora la red de neuronas está completa, algoritmo de aprendizaje incluido.

4. Interface hombre-máquina

Antes de codificar el sistema que gestiona la red, se codifica una pequeña interfaz GUI que nos permitirá visualizar diferentes mensajes. Según la aplicación elegida, el programa principal podrá a continuación implementar el que permita hacer salidas por la consola, notificaciones, etc.

Por lo tanto, solo contiene un único método.

```
public interfaz IHM {
   void MostrarMensaje(String msg);
}
```

5. Sistema completo

La última clase genérica **`Sistema`**, es la que gestiona toda la red y el bucle de aprendizaje. Se utilizan varios criterios de parada:

- Ver aparecer el sobre-aprendizaje, es decir, que el error del conjunto de generalización aumenta en lugar de disminuir en cinco generaciones consecutivas.
- Alcanza el resultado esperado, es decir, que el error del conjunto de aprendizaje es inferior a un umbral.
- Se alcanza el número máximo de iteraciones y por lo tanto, nos paramos.

En primer lugar, la clase **`Sistema`** contiene varios atributos:

- los datos de aprendizaje `datos`,
- una red de neuronas adjunta `red`,
- una IHM utilizada para las visualizaciones,
- la tasa de aprendizaje inicial,
- el error máximo,
- el número máximo de iteraciones.

Se proponen valores por defecto para la configuración de la red. Por lo tanto, la base es:

```
public class Sistema {
   protected ColeccionPuntos datos;
   protected RedNeuronas red;
   protected IHM ihm;

   protected doble tasaAprendizaje = 0.3;
   protected doble errorMax = 0.005;
   protected int numIteracionesMax = 10001;

   // Métodos aquí
}
```

El primer método es el constructor. Este recibe muchos argumentos: el número de entradas, de neuronas ocultas, de salidas, el contenido del archivo de datos, el porcentaje de ejemplos de aprendizaje respecto a la generalización

y la IHM.

```
   public Sistema(int _numEntradas, int _numOcultas, int _numSalidas,
String[] _datos, doble _ratioAprendizaje, IHM _ihm) {
       datos = new ColeccionPuntos(_datos, _numSalidas,
_ratioAprendizaje);
       red = new RedNeuronas(_numEntradas, _numOcultas, _numSalidas);
       ihm = _ihm;
   }
```

Los siguientes métodos permiten modificar la configuración, cambiando la tasa de aprendizaje (`setTasaAprendizaje`), el error máximo (`setErrorMax`) y el número máximo de iteraciones (`setNumIteracionesMax`).

```
   public void setTasaAprendizaje(doble valor) {
       tasaAprendizaje = valor;
   }

   public void setErrorMax(doble valor) {
       errorMax = valor;
   }

   public void setNumIteracionesMax(int valor) {
       numIteracionesMax = valor;
   }
```

El último método es el principal `Lanzar`. Empezamos inicializando las diferentes variables. A continuación, mientras no se alcance uno de los criterios de parada, en bucle hacemos:

- Actualizar los errores de la iteración anterior e inicializar los errores para esta iteración.
- Para cada punto de aprendizaje, se calcula su salida y el error cometido y se adaptan los pesos de la red.
- Para cada punto de generalización, se calcula la salida y el error.
- Si el error de generalización ha aumentado, se incrementa el número de iteraciones sin mejora. En caso contrario se reinicializa.
- Se termina visualizando los valores de la iteración actual (errores y tasa).

El método llama a dos métodos privados para evaluar los errores (de manera que se conserve la legibilidad).

```
   public void Lanzar() {
       // Inicialización
       int numIteraciones = 0;
       doble errorTotal = Double.POSITIVE_INFINITY;
       doble antiguoError = Double.POSITIVE_INFINITY;
       doble errorGeneralizacionTotal = Double.POSITIVE_INFINITY;
       doble antiguoErrorGeneralizacion =
Double.POSITIVE_INFINITY;
       int numSobreAprendizaje = 0;

       while (numIteraciones < numIteracionesMax && errorTotal >
```

```
errorMax && numSobreAprendizaje < 5) {
           // Pasa a la siguiente iteración
           antiguoError = errorTotal;
           errorTotal = 0;
           antiguoErrorGeneralizacion = errorGeneralizacionTotal;
           errorGeneralizacionTotal = 0;

           // Evaluación y aprendizaje
           errorTotal = Evaluar();
           errorGeneralizacionTotal = Generalizacion();
           if (errorGeneralizacionTotal >
antiguoErrorGeneralizacion) {
               numSobreAprendizaje++;
           }
           else {
               numSobreAprendizaje = 0;
           }

           // Visualización e incremento
           ihm.MostrarMensaje("Iteración n°" + numIteraciones +
" - Error total: " + errorTotal + " - Generalización:
" + errorGeneralizacionTotal + " - Tasa: " + tasaAprendizaje +
" - Media: " + Math.sqrt(errorTotal /
datos.ptsAprendizaje.length));
           numIteraciones++;
       }
   }

   private doble Evaluar() {
       doble errorTotal = 0;
       for (PuntoND punto: datos.getPtsAprendizaje()) {
           doble[] salidas = red.Evaluar(punto);
           for (int num = 0; num < salidas.length; num++) {
               doble error = punto.salidas[num] - salidas[num];
               errorTotal += (error * error);
           }
           red.AjustarPesos(punto, tasaAprendizaje);
       }
       return errorTotal;
   }

   private doble Generalizacion() {
       doble errorGeneralizacionTotal = 0;
       for (PuntoND punto: datos.getPtsGeneralizacion()) {
           doble[] salidas = red.Evaluar(punto);
           for (int num = 0; num < salidas.length; num++) {
               doble error = punto.salidas[num] - salidas[num];
               errorGeneralizacionTotal += (error * error);
           }
       }
       return errorGeneralizacionTotal;
   }
```

Todo el sistema está preparado, aprendizaje y bucle principal incluidos.

6. Programa principal

La última etapa consiste en crear el programa principal **Aplicacion**. Esto implementa la interfaz `IHM` y por lo tanto, tiene un método `MostrarMensaje()`. También tiene un método `leerArchivo()` que recupera todas las líneas de un archivo indicado.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class Aplicacion implements IHM {

   protected String[] leerArchivo(String nombreArchivo, boolean
quitarEncabezado) {
       try {
           ArrayList<String> lineas = new ArrayList();
           BufferedReader buffer = new BufferedReader(new
FileReader(nombreArchivo));
           String linea;
           while ((linea = buffer.readLine()) != null) {
               lineas.add(linea);
           }
           buffer.close();
           if (quitarEncabezado) {
               lineas.remove(0);
           }
           String[] contenido = lineas.toArray(new
String[lineas.size()]);
           return contenido;
       }
       catch (IOExcepción e) {
           System.err.println(e.toString());
           return null;
       }
   }

   @Override
   public void MostrarMensaje(String msg) {
       System.out.println(msg);
   }
}
```

Para terminar, el método `main()` instancia la clase y llama a su método `Lanzar()`, que se define más adelante en función del problema elegido:

```
   public static void main(String[] args) {
       Aplicacion app = new Aplicacion();
       app.Lanzar();
   }
```

```
protected void Lanzar() {
    // Problema del XOR
    /*String[] contenido = leerArchivo("xor.txt", true);
    Sistema sistema = new Sistema(2, 2, 1, contenido, 1.0, this);
    sistema.Lanzar();*/

    // Problema Abalone
    String[] contenido = leerArchivo("abalone_norm.txt", false);
    Sistema sistema = new Sistema(10, 4, 1, contenido, 0.8, this);
    sistema.setTasaAprendizaje(0.6);
    sistema.setNumIteracionesMax(50000);
    sistema.Lanzar();
}
```

7. Aplicaciones

a. Aplicacion del XOR

El primer problema que utilizamos es el del operador booleano **XOR**, llamado "O exclusivo". Si cambiamos verdadero por 1 y falso por 0, a continuación se muestra la tabla de verdad de XOR:

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

XOR indica que uno de los dos valores vale "verdadero", pero no los dos al mismo tiempo.

Al contrario de lo que sucede con otros operadores booleanos (como el "Y" y el "O"), este no es linealmente separable. Por lo tanto, es una buena prueba para una red de neuronas.

La primera etapa consiste en crear el archivo que contiene los ejemplos, aquí son cuatro. Visto su pequeño número, no habrá conjunto de generalización.

Por lo tanto, el archivo xor.txt contiene el siguiente texto:

```
X   Y       XOR
0   0       0
0   1       1
1   0       1
1   1       0
```

Para entender este problema, se define el método `Lanzar`, de la clase **`Aplicacion`**. Por lo tanto, esta consiste en leer el archivo y después, crear el sistema. Elegimos dos neuronas en la capa oculta. Para terminar, se lanza el aprendizaje.

```
protected void Lanzar() {
    // Problema del XOR
    String[] contenido = leerArchivo("xor.txt", true);
    Sistema sistema = new Sistema(2, 2, 1, contenido, 1.0, this);
    sistema.Lanzar();
}
```

El archivo xor.txt se debe añadir a la raíz del código de manera que se tenga acceso desde el código.

Con los argumentos elegidos, la red de neuronas converge y se para cuando se alcanza el error residual de 0.005. En este momento, la salida es, como media, de 0.97 para las salidas verdaderas y 0.03 para las salidas falsas. Por lo tanto, el error es mínimo y si se redondean las salidas (porque se trata de booleanos), este se hace nulo.

Como aquí no hacemos regresión sino una clasificación entre dos clases, sería necesario cambiar el algoritmo para tener mejores resultados y tomar la función de coste logístico, en lugar del error cuadrático.

Sin embargo, puede permanecer bloqueado en una solución óptima local (que no sucede en nuestras pruebas, pero es una posibilidad) y no sucede más al salir. Por este motivo es interesante volver a lanzar varias veces el aprendizaje, para conservar las mejores redes. En efecto, siendo fuerte la tasa de aprendizaje, el algoritmo puede divergir.

b. Aplicacion en Abalone

El segundo problema es más complejo. Se define en la UCI (*University of California, Irvine*) Machine Learning Repository. Se trata de un banco de datos que contiene muchos datasets, que de esta manera se pueden utilizar para hacer pruebas en el dominio del aprendizaje automático.

Abalone se presenta en la siguiente dirección: https://archive.ics.uci.edu/ml/datasets/Abalone

Este conjunto, presentado en 1995, propone determinar la edad de la oreja de mar (se trata de unas conchas llamadas abalone en inglés), en función de características físicas. Para esto, se dispone de ocho datos:

- el sexo del animal, a elegir entre macho, hembra e hijo,
- la longitud de la concha más larga,
- el diámetro medido perpendicularmente a la longitud,
- la altura,
- el peso total de la concha,
- el peso del animal (sin la concha),
- el peso de las vísceras (por tanto después de haberla desangrado),
- el peso de la concha seca.

La salida buscada es el número de anillos de la concha, que se corresponde con la edad del animal (añadiendo 1.5). Todos tienen entre 1 y 29 años.

Hay disponibles 4.177 ejemplos, lo que permite crear un conjunto de aprendizaje que contiene el 80% de los datos (es decir, 3.341 animales) y el 20% para probar la generalización (es decir, 836).

El archivo .csv que contiene los datos está disponible en el sitio web. Sin embargo, para poder utilizarlo con nuestra red de neuronas, tenemos que sustituir el primer dato (el sexo) por tres datos booleanos: ¿es macho, hembra o un hijo? Por lo tanto, hemos sustituido nuestro dato textual por tres entradas que contienen 0 (falso) o 1 (verdadero), lo que eleva a 10 el número de variables de entrada.

La salida de la red era el resultado de una sigmoidea, por lo que el resultado que se puede obtener es una salida comprendida entre 0 y 1. Como la edad aquí está comprendida entre 1 y 29, se va a normalizar este valor. Para esto, será suficiente con dividir la edad por 60 y agregar 0,25. De esta manera, los datos estarán comprendidos entre 0,25 y 0,75.

Los límites de la sigmoidea son 0 y 1en más o menos el infinito, por lo que es muy difícil generar valores cercanos a 1 o a 0. Por lo tanto, se aconseja tener salidas cercanas a 0.5, lo que facilitará la convergencia. En este caso, también se podría cambiar la función de activación de las neuronas de salida, por una función ReLU, pero esto haría cambiar los cálculos de retro-propagación.

El archivo obtenido después de las modificaciones se puede descargar desde el sitio del fabricante, en la solución NetBeans.

Para resolver este problema, seleccionamos cuatro neuronas ocultas y una tasa de aprendizaje de 0.1. Por lo tanto, el método `Lanzar` de la clase **`Aplicacion`** queda como sigue:

```
protected void Lanzar() {
    String[] contenido = leerArchivo("abalone_norm.txt", false);
    Sistema sistema = new Sistema(10, 4, 1, contenido, 0.8, this);
    sistema.setTasaAprendizaje(0.1);
    sistema.setNumIteracionesMax(50000);
    sistema.Lanzar();
}
```

El archivo abalone_norm.txt también se debe añadir a la raíz de la solución.

Se han lanzado 10 simulaciones con fines estadísticos. Durante la primera generación, el error cuadrático acumulado es de 11.45, y sobre el conjunto de generalización de 1.89.

Cuando el aprendizaje se detiene, solo es 4.06 y un error de generalización de 1.01. Por lo tanto, el error se ha dividido casi por 3 sobre el conjunto de aprendizaje y por 2 sobre la generalización: el aprendizaje ha permitido mejorar los resultados. Esto se corresponde con un error medio de menos de 2 años en cada oreja de mar, que como recordatorio viven entre 1 y 29 años.

Aunque no optimizadas, sin embargo vemos que las redes neuronales llegan a aprender a partir de datos y a ofrecer resultados de calidad.

c. Mejoras posibles

Como hemos visto anteriormente, la primera mejora consistiría en optimizar los diferentes argumentos de aprendizaje:

- el número de neuronas ocultas,
- la tasa de aprendizaje,
- el número máximo de iteraciones,
- los pesos iniciales (para reducirlos).

Además, los resultados se podrán mejorar modificando la estrategia de modificación de la tasa de aprendizaje, o incluso modificando el orden de los ejemplos presentados a la red en cada iteración. En efecto, durante un descenso por gradiente estocástico, es importante mezclar los datos en cada iteración, lo que no hemos hecho aquí.

También se podrían utilizar algoritmos con mejor rendimiento que la retro-propagación del gradiente clásico. Para terminar, se podría utilizar mini-batchs en lugar de un descenso estocástico (es decir, actualizar los pesos de cada x puntos y no de cada punto).

Por lo tanto, los resultados que se presentan aquí son sobre todo para ilustrar el funcionamiento de estas redes, sin buscar resultados óptimos.

Resumen

Las **redes neuronales** se han inspirado en el funcionamiento del cerebro de los seres vivos. En efecto, las sencillas células se limitan a transmitir impulsos eléctricos en función de las entradas que reciben y permiten el conjunto de comportamientos y razonamientos. Su potencia emerge del número de células grises y de sus conexiones.

Su principal utilización se encuentra en la **Machine Learning**. Las principales formas de aprendizaje son el aprendizaje no supervisado (para las tareas de **clustering**) y el aprendizaje supervisado (para la **regresión** o la **clasificación**). Sin embargo, las técnicas puramente matemáticas no permiten la resolución de problemas complejos, en particular no linealmente separables. Las redes neuronales y en particular el Deep Learning, permiten eliminar estos límites.

La neurona artificial, llamada **neurona formal**, combina una **función de agregación**, que permite obtener un valor único a partir del conjunto de entradas, pesos de la neurona y de su sesgo y una **función de activación**, que permite obtener su salida.

La función de agregación normalmente es una suma ponderada. La función de activación es más variable, pero se corresponde con la función sigmoidea o la función ReLU en la mayoría de los casos actuales.

Sin embargo, las redes de una única capa también están limitadas a los problemas **linealmente separables**, motivo por el que las redes más utilizadas son de tipo feed-forward: las entradas pasan de una primera capa de neuronas, completamente conectada a la siguiente y de esta manera sucesivamente hasta la capa de salida.

Sea cual sea el tipo elegido, sin embargo hay que ajustar los pesos y los umbrales para un problema dado. Esta etapa de **aprendizaje** es compleja y casi imposible de hacer "a mano". Existen muchos algoritmos de aprendizaje, sean o **no supervisados**, **por refuerzo** o **supervisados**.

En este último caso y para las redes de una única capa de tipo perceptrón, se puede utilizar el **descenso por gradiente**. Para las redes de tipo feed-forward, se aplica normalmente el **algoritmo de retro-propagación**. Lo que consiste en propagar el error sobre la capa de salida a las capas ocultas, una después de la otra, y en corregir los pesos de cada capa para disminuir el error total.

Sin embargo, la dificultad consiste en evitar el **sobre-aprendizaje**. Para esto, hay que comparar la evolución del error total en la parte de los datos presentada para el aprendizaje y sobre el conjunto de generalización, que no se utiliza. Tan pronto como el error en generalización aumente, hay que detener el aprendizaje.

Estas redes se utilizan en muchos dominios, porque dan buenos resultados, son un esfuerzo de posicionamiento muy bajo. Además, permiten resolver los problemas demasiado complejos para las técnicas más clásicas.

En este capítulo se ha propuesto una posible **implementación** de una red feed-forward, y se ha aplicado al problema sencillo del XOR y a uno más complejo llamado Abalone, consistente en determinar la edad de las conchas a partir de datos físicos. En los dos casos, la red da buenos resultados, sin haber sido optimizado completamente y con una arquitectura muy simple.

¿Por qué una webgrafía?

Esta webgrafía presenta distintos enlaces a aplicaciones de inteligencia artificial. Permite hacerse una idea del uso real que se realiza de una u otra técnica, cada una con su propia sección.

Los artículos mencionados están en francés o en inglés, y se indican con la mención [FR] o bien [EN] a continuación de su título.

Esta lista está lejos de ser exhaustiva, pero presenta aplicaciones muy diferentes.

Sistemas expertos

Applications of Artificial Intelligence for Organic Chemistry [EN], R. Lindsay, B. Buchanan, E. Feigenbaum, J. Lederberg, 1980:

http://profiles.nlm.nih.gov/ps/access/BBALAF.pdf

Este PDF presenta el proyecto Dendral, que es el primer gran sistema experto creado en los años sesenta. Permite reconocer componentes químicos en función de sus características.

MYCIN: A Quick Case Study [EN], A. Cawsey, 1994:

http://cinuresearch.tripod.com/ai/www-cee-hw-ac-uk/_alison/ai3notes/section2_5_5.html

MYCIN es otro sistema experto pionero (desarrollado en los años setenta) y reconocido mundialmente. Permitía identificar las principales enfermedades de la sangre y proponía un tratamiento. Se trata, aquí, de una corta discusión acerca de sus puntos fuertes y débiles.

Clinical decision support systems (CDSSs) [EN], Open Clinical, 2006:

http://www.openclinical.org/dss.html

Esta página presenta varios sistemas expertos utilizados en medicina, así como su funcionamiento general y los principales trabajos publicados.

Automatic classification of glycaemia measurements to enhance data interpretation in an expert system for gestational diabetes [EN], Estefania Caballero-Ruiz y al., Expert Systems with Applications, Volume 63, noviembre 2016

http://www.sciencedirect.com/science/article/pii/S0957417416303645

Este artículo científico presenta un uso de los sistemas expertos en el dominio médico, más concretamente para ayudar a la toma de decisiones relativas a la diabes gestacional.

Expert system for automated bone age determination [EN], Jinwoo Seok et al., Expert Systems with Applications, Volume 50, mayo 2016

http://www.sciencedirect.com/science/article/pii/S0957417415008131

En este artículo, se utiliza un sistema experto para determinar la edad ósea de personas a partir de radios de la mano. La aplicación combina las edades predichas de diferentes zonas de la mano para dar una edad todavía más precisa.

Machine learning for an expert system to predict preterm birth risk [EN], Journal of the American Medical Informatics Association, L Woolery, J Grzymala-Busse, 1994:

http://www.ncbi.nlm.nih.gov/pmc/articles/PMC116227/

Se trata de un artículo científico publicado en la revista Journal of the American Medical Informatics Association que presenta un sistema experto que permite estimar el riesgo de nacimientos prematuros.

An Expert System for Car Failure Diagnosis [EN], Proceedings of World Academy of Science, Engineering and Technology, volume 7, A. Al-Taani, 2005:

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.3377

Este artículo científico, que puede descargarse en PDF, procede también de una revista (Proceedings of World Academy of Science, Engineering and Technology, volume 7, 2005). Propone un sistema experto que permite detectar fallos en los vehículos. También explica las dificultades encontradas durante su implementación.

Expert System in Real World Applications [EN], Generation 5, K. Wai, A. Abdul Rahman, M. Zaiyadi, A. Aziz, 2005:

https://pdfs.semanticscholar.org/94a7/595c51fada8ecfe0b2ecc805a6d84f8517d4.pdf

Se trata de un artículo publicado en el sitio "Generation 5" que ofrece una visión general de las aplicaciones de los sistemas expertos, en particular en agricultura, en educación, en gestión medioambiental y en medicina. Se trata, por tanto, de un muy buen punto de entrada a las principales aplicaciones en estos dominios.

Natural Language Processing With Prolog in the IBM Watson System [EN], Adam Lally, Paul Fodor, 2011

https://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/

Este artículo presenta el uso de Prolog en el sistema de IBM Watson para mejorar y facilitar la comprensión de las Preguntas en el caso del Jeopardy (que Watson ha ganado frente a los mejores competidores del juego televisivo americano).

ClioPatria [EN], 2015

http://www.semantic-web-journal.net/content/cliopatria-swi-prolog-infrastructure-semantic-web

Este framework permite crear sitios web semánticos en Prolog.

Tutorial - Creating Web Applications in SWI-Prolog [EN], Anne Ogborn, 2015

http://www.pathwayslms.com/swipltuts/html/#_introduction

Este tutorial explica todas las etapas para crear un sitio web basándose en las librerías de SWI-Prolog, con ejercicios para realizar.

Prosper: A Framework for Extending Prolog Applications with a Web Interface [EN], 2007

http://prospear.sourceforge.net/

Otro framework para crear sitios web en SWI-Prolog, que data del año 2007.

Ficha de producto "Washing Machines - LG T8018EEP5" [EN], LG, 2014:

https://www.lg.com/in/washing-machines/lg-T7281NDDL

Esta ficha de producto para una lavadora de marca LG refleja el uso de un controlador difuso (Fuzzy) para escoger la cantidad de agua y el tiempo de lavado. La lógica difusa se utiliza como argumento de venta.

Ficha de producto de un termostato Seltron - ST2RDR [EN], Seltron, 2017

https://www.seltron.eu/en/products/room-thermostat-rtd2drn/

Esta ficha de producto presenta un termostato que utiliza la lógica difusa y presenta sus datos técnicos.

Fuzzy Logic and NeuroFuzzy in Appliances, Constantin von Altrock, [EN], 1996

http://www.fuzzytech.com/e/e_a_esa.html

Este artículo presenta los principales usos de la lógica difusa en los dispositivos eléctricos domésticos. El artículo es antiguo pero todavía de actualidad.

Patente "Fuzzy logic control for an electric clothes dryer" [EN], registrada por Whirlpool Corporation, 2001:

https://www.google.com/patents/US6446357

La patente, registrada por la marca de electrodomésticos Whirlpool, describe el funcionamiento de una máquina secadora de ropa que utiliza lógica difusa. Este controlador permite seleccionar el tiempo de secado en función de la carga y de la humedad de la ropa.

Fuzzy Logic in Automotive Engineering [EN], Circuit Cellar Ink, Issue 88, C. von Altrock, 1997:

ftp://ftp.me.psu.ac.th/pub/me/Fuzzy/88constantin.pdf

Esta edición de la revista está dedicada a las aplicaciones de la lógica difusa en la automoción. Los avances posteriores han sido muy importantes, pero los controladores referenciados entonces todavía se utilizan, y en todas las grandes marcas.

Fuzzy logic method and apparatus for battery state of health determination [EN], USPTO Patent Database, registrada por Cadex, 2001:

http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=/netahtml/PTO/srchnum.htm&r=1&f=G&l=50&s1=7,072,871.PN.&OS=PN/7,072,871&RS=PN/7,072,871

Esta patente presenta el uso de lógica difusa para determinar el estado de una pila o de una batería a partir de parámetros electromagnéticos.

The use of Fuzzy Logic for Artificial Intelligence in Games [EN], M. Pirovano, 2012:

http://www.michelepirovano.com/pdf/fuzzy_ai_in_games.pdf

En este artículo, el autor presenta numerosos usos de la lógica difusa en videojuegos tras repasar qué es la inteligencia artificial y, en particular, los conceptos de la lógica difusa. Hay disponibles también muchos vínculos hacia otros artículos para profundizar en este tema.

A Computer Vision System for Color Grading Wood Boards Using Fuzzy Logic [EN], IEEE International Symposium on Industrial Electronics, J. Faria, T. Martins, M. Ferreira, C. Santos, 2008:

http://repositorium.sdum.uminho.pt/bitstream/1822/16874/1/C24-%20A%20Computer%20Vision%20System%20for%20Color%20Grading%20Wood%20Boards%20Using%20Fuzzy%20Logic_2.pdf

Este artículo, publicado tras una conferencia, permite ver una aplicación más atípica de la lógica difusa. En efecto, se utiliza aquí para determinar el color de una madera y poder agrupar las planchas en función del tono (por ejemplo, para construir muebles en los mismos tonos).

Leaf Disease Grading by Machine Vision and Fuzzy Logic [EN], International Journal of Computer Technology and Applications Vol 2 (5), S.Sannakki, V. Rajpurohit, V. Nargund, A. Kumar R, P. Yallur, 2011:

http://ijcta.com/documents/volumes/vol2issue5/ijcta2011020576.pdf

La lógica difusa se aplica, en este caso, a la búsqueda de enfermedades a partir de imágenes de hojas de plantas. El sistema permite determinar, además, la gravedad actual de la enfermedad.

Búsqueda de rutas

Path-Finding Algorithm Applications for Route-Searching in Different Areas of Computer Graphics [EN], New Frontiers in Graph Theory, cap. 8, C. Szabó, B. Sobota, 2012

http://cdn.intechopen.com/pdfs-wm/29857.pdf

Se trata de un capítulo extraído de un libro más completo sobre la teoría de los grafos. Los autores se interesan aquí por los algoritmos de búsqueda de rutas y sus aplicaciones en las imágenes informáticas.

The Bellman-Ford routing algorithm [EN], FortiOS Online Handbook, 2015 :

http://help.fortinet.com/fos50hlp/56/Content/FortiOS/fortigate-networking/fortigate-advanced-routing/Routing_RIP/Background_Concepts.htm

Se trata de una parte del libro sobre FortiOS que presenta los diferentes algoritmos de enrutamiento, y de manera más concreta la implementación del algoritmo de Bellman-Ford para el enrutamiento RIP.

OSPF Background and concepts [EN], FortiOS Online Handbook, 2015

http://help.fortinet.com/fos50hlp/56/Content/FortiOS/fortigate-networking/fortigate-advanced-routing/Routing_OSPF/OSPF_Background_Concepts.htm

Esta sección, que forma parte del mismo libro que el del enlace anterior, explica el protocolo OSPF, que sustituye a RIP. En lugar de utilizar Bellman-Ford para la búsqueda de rutas, que implementa Dijkstra.

Deep Blue, el ordenador con una sola misión: ganar al humano, Pablo Espeso

https://www.xataka.com/otros/deep-blue-el-ordenador-con-una-sola-mision-ganar-al-humano

En este artículo se presenta Deep Blue, la máquina capaz de batir a los mejores jugadores de ajedrez del mundo (entre ellos Kasparov).

Google AI algorithm masters ancient game of Go, Nature, E. Gibney, 2016 [EN]

http://www.nature.com/news/google-ai-algorithm-masters-ancient-game-of-go-1.19234

Este artículo presenta el algoritmo AlphaGo, que ha batido a los dos campeones más grandes Go.

Algoritmos genéticos

Sitio web del laboratorio "Notredame's Lab Comparative Bioinformatics" [EN], C. Notredame:

http://www.tcoffee.org/homepage.html

Este laboratorio, situado en Barcelona, en el centro de regulación genómica, utiliza de forma importante los algoritmos genéticos para resolver problemas biológicos y, en particular, en el análisis y alineación de secuencias de ARN. En su web se presentan los distintos proyectos.

Staples' Evolution [EN], BloombergBusinessweek, Innovation & Design, J. Scanlon, 2008:

https://www.bloomberg.com/news/articles/2008-12-29/staples-evolutionbusinessweek-business-news-stock-market-and-financial-advice

Este artículo, aparecido en una revista económica, estudia la estrategia de marketing de Stables. En efecto, esta empresa utilizó un algoritmo genético para relanzar su marca mejorando el empaquetado de sus paquetes de folios.

A (R)evolution in Crime-fighting [EN], Forensic Magazine, C. Stockdale, 2008:

http://www.forensicmag.com/articles/2008/06/revolution-crime-fighting

Este artículo presenta la dificultad de crear retratos robots a partir de los recuerdos de los testigos. Esta fase puede simplificarse mediante un algoritmo genético que haga evolucionar estos retratos, dejando la elección al usuario, que escogerá el retrato "más cercano" a sus recuerdos.

Automatic timetable conflict resolution with genetic algorithms [EN], Global Railway, D. Abels et J. Balsiger, 2017

https://www.globalrailwayreview.com/article/61578/automatic-timetable-algorithms/

Este artículo explica el uso de los algoritmos genéticos hechos en Suiza para determinar los horarios de los trenes, teniendo en cuenta las diferentes restricciones.

Daewha Kang Design's rainbow publishing HQ celebtrates history in Paju Book City, Korea [EN], D. Kang, DesignBoom, 2017

https://www.designboom.com/architecture/daewha-kang-design-rainbow-publishing-headquarters-paju-book-city-korea-08-03-2017/

Este artículo presenta un edificio en Corea del Sur que contiene una biblioteca que también es la barandilla de una escalera. El diseño de esta biblioteca se ha creado usando un algoritmo genético.

Metaheurísticos

A Comparative Study on Meta Heuristic Algorithms for Solving Multilevel Lot-Sizing Problems [EN], Recent Advances on Meta-Heuristics and Their Application to Real Scenarios, I. Kaku, Y. Xiao, Y. Han, 2013:

http://www.intechopen.com/books/recent-advances-on-meta-heuristics-and-their-application-to-real-scenarios/a-comparative-study-on-meta-heuristic-algorithms-for-solving-multilevel-lot-sizing-problems

Se trata de un capítulo de un libro dedicado a una aplicación industrial de los metaheurísticos, disponible gratuitamente. El problema principal consiste en escoger las cantidades de cada componente que hay que producir. Se comparan varios algoritmos.

A Two-Step Optimization Method for Dynamic Weapon Target Assignment Problem [EN], Recent Advances on Meta-Heuristics and Their Application to Real Scenarios, C. Leboucher, H.-S. Shin, P. Siarry, R. Chelouah, S. Le Ménec, A. Tsourdos, 2013:

http://www.intechopen.com/books/recent-advances-on-meta-heuristics-and-their-application-to-real-scenarios/a-two-step-optimisation-method-for-dynamic-weapon-target-assignment-problem

Los militares encuentran también numerosas aplicaciones a los metaheurísticos. Este capítulo, extraído del mismo libro que el anterior, está dedicado a una aplicación militar que consiste en saber cómo asignar los medios de defensa (o de ataque) en función de las distintas amenazas.

Metaheuristics and applications to optimization problems in telecommunications [EN], Handbook of optimization in telecommunications, S. Martins, C. Ribeiro, 2006:

https://link.springer.com/chapter/10.1007%2F978-0-387-30165-5_4

Las telecomunicaciones son un dominio que exige numerosas optimizaciones. Este libro está dedicado a este tema. Entre todas las técnicas posibles, los metaheurísticos se sitúan en buen lugar, y son objeto de todo el capítulo 1 aquí citado.

Gradient Descent for Machine Learning [EN], J. Brownlee, 2016

http://machinelearningmastery.com/gradient-descent-for-machine-learning/

Este artículo presenta el uso de metaheurísticos (de manera más particular el descente por gradiente), en el dominio de la Machine Learning.

A Tabu Search Algorithm for application placement in computer clustering [EN], Computers & Operations Research, J. van der Gaast and al., 2014

http://is.ieis.tue.nl/staff/yqzhang/wp-content/uploads/2016/02/tabuCOR.pdf

Este artículo presenta una aplicación reciente de los metaheurísticos durante la colocación de aplicaciones informáticas en un cluster informático, limitando al máximo los costes de creación de nodos.

Sistemas multiagentes

MASSIVE [EN], sitio web de la aplicación, 2011:

http://www.massivesoftware.com/

Las películas y los videojuegos utilizan con frecuencia sistemas multiagentes, aunque pueden usarse también en educación, arquitectura o en simulaciones. MASSIVE es una aplicación muy utilizada que permite simular multitudes. Las páginas de referencias son impresionantes.

Ant Colony Optimization - Techniques and applications [EN], edited by H. Barbosa, 2013:

http://www.intechopen.com/books/ant-colony-optimization-techniques-and-applications

Se trata de un libro completo dedicado únicamente a los algoritmos basados en hormigas, en particular a sus aplicaciones en distintos dominios, como la logística, o a variantes/extensiones.

An ant colony optimization algorithm for job shop scheduling problem [EN], E. Flórez, W. Gómez, L. Bautista, 2013:

http://arxiv.org/abs/1309.5110

Este artículo se interesa en el uso de un algoritmo de colonia de hormigas en el dominio de la planificación de operaciones. Se utiliza una variante en estos trabajos.

Train Scheduling using Ant Colony Optimization Technique [EN], Research Journal on Computer Engineering, K. Sankar, 2008:

http://www.academia.edu/1144568/Train_Scheduling_using_Ant_Colony_Optimization_Technique

Tras recordar el comportamiento de las hormigas y el estado de la cuestión, este pequeño artículo presenta la aplicación de las hormigas al problema de la planificación de trenes.

Army to get self-reliant, autonomous robots soon [EN], The Times of India, C. Kumar, 2017

http://timesofindia.indiatimes.com/india/army-to-get-self-reliant-autonomous-robots-soon/articleshow/57465221.cms

Este artículo presenta los últimos avances militares en India, en particular la utilización de sistemas multi-agente que podrían cooperar para garantizar la defensa de un país.

DeepMind’s AI has learnt to become " highly aggressive " when it feels like it’s going to lose [EN], Wired, M. Burgess, 2017

http://www.wired.co.uk/article/artificial-intelligence-social-impact-deepmind

Este artículo presenta los resultados de búsqueda de DeepMind (filial de Google) en sistemas multi-agentes. Se utilizaron dos juegos, uno que pide una estrategia más agresiva y otro que permite la cooperación. La IA ha encontrado y aplicado estas estrategias.

Kiva Systems, an Amazon company [EN], Multi-Robot Systems, J. Durham, 2014

http://multirobotsystems.org/?q=node/87

Este artículo presenta el sistema Kiva que se utiliza en las empresas de Amazon: robots que mueven los palets para ayudar a la preparación de pedidos. Los robots trabajan conjuntamente, porque forman un sistema multi-agentes.

Redes neuronales

An Application of Backpropagation Artificial Neural Network Method for Measuring The Severity of Osteoarthritis [EN], International Journal of Engineering & Technology, Vol. 11, no. 3, D. Pratiwi, D. Santika, B. Pardamean, 2011:

http://arxiv.org/abs/1309.7522

La medicina utiliza numerosas imágenes, y las redes neuronales son una técnica muy eficaz para encontrar patrones en ellas. En este artículo, los autores indican cómo es posible estimar la gravedad de la artrosis a partir de radiografías de la mano y de la muñeca.

Inteligencia artificial en los nuevos P30 y P30 Pro, Marcos Merino, marzo 2019

https://www.xataka.com/inteligencia-artificial/huawei-se-proclama-lider-procesamiento-inteligencia-artificial-moviles-su-gama-p30

Este artículo habla de cómo los dispositivos son capaces de identificar escenas, permitiendo al software sugerir un modo o bien aplicar una previsualización del procesado según lo que identifica.

Reconocimiento facial: la tecnología de IA que pretende superar al cerebro humano, 2018

https://www.infobae.com/america/tecno/2019/03/28/reconocimiento-facial-la-tecnologia-de-ia-que-pretende-superar-al-cerebro-humano/

Los investigadores han utilizado el Deep Learning para entrenar una red que puede reconocer con resultados satisfactorios, a la persona que aparece en una foto, incluso si su cara está parcialmente oculta. Esto es importante para la seguridad en las ciudades.

Google's Artificial Brain Learns to Find Cat Video [EN], L. Clark, Wired IK, junio 2012

https://www.wired.com/2012/06/google-x-neural-network/

Este artículo presenta el primer éxito del proyecto Google Brain, en el que un algoritmo ha podido identificar los gatos presentes en los vídeos de YouTube, sin estar etiquetados por un humano.

AlphaGo

https://deepmind.com/research/alphago/

Este sitio presenta el algoritmo AlphaGo, que viene del proyecto Google Brain, y que ha conseguido batir a los mejores jugadores del mundo de Go en 2016 y 2017. El juego Go está considerado como el juego más difícil, incluso más que el ajedrez.

Instalación de SWI-Prolog

SWI-Prolog es una aplicación que permite utilizar Prolog en los PC Windows o Linux, así como en los Mac. El sitio oficial es: http://www.swi-prolog.org/

La aplicación está disponible para varias plataformas (Windows, Mac OS, incluso archivos fuente para compilar), en la página: http://www.swi-prolog.org/download/stable

La primera etapa consiste en descargar la aplicación:

- Seleccione, en la sección **Binaries**, el programa adaptado a su equipo.
- Guarde el archivo.
- Ejecute la instalación .
- Acepte el mensaje de seguridad si es necesario.

Arranca la instalación. Las diferentes etapas son muy simples:

- En la presentación de la licencia, seleccione **I agree**.
- Seleccione, a continuación, una instalación de tipo **Typical** (por defecto) y **Next**.
- Seleccione la ubicación de instalación o deje la opción por defecto.
- En la siguiente pantalla, deje las opciones por defecto y seleccione **Install**.

Una vez terminada la instalación, basta con hacer clic en **finished** (y seleccionar o no leer el archivo Readme).

Uso de SWI-Prolog en Windows

Tras la ejecución del programa, se presenta en Windows de la siguiente manera:

Vemos la consola Prolog y el prompt, que espera un comando para ejecutar.

Para crear un nuevo proyecto:

- **File** y a continuación **New**.
- Seleccione la ubicación del archivo.

Por defecto, los archivos prolog tienen la extensión .pl. Sin embargo, puede escoger la extensión que desee.

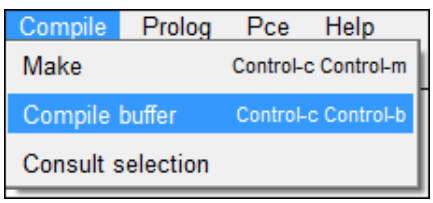
El archivo (vacío inicialmente) que contiene las reglas y los predicados se abre en otra ventana:

En este archivo es donde escribiremos el contenido del motor. Aquí, utilizaremos el ejemplo del capítulo dedicado a los Sistemas expertos:

```
comer(gato, raton).
comer(raton, queso).

piel(gato).
piel(raton).

amigos(X, Y) :-
    comer(X, Z),
    comer(Z, Y).
```

Para poder utilizar los predicados y reglas deseados:

- Escriba el ejemplo anterior o su propio contenido.
- Guarde el archivo (*Safe buffer*).
- Compílelo: menú **Compile** y, a continuación, **Compile buffer**.



En la consola, debe aparecer un mensaje indicando que se ha compilado el archivo. Si no, corrija los errores y repita la operación.

Una vez compilado el código, se carga en la consola y puede utilizarse. Basta con indicar qué se desea en esta última. Tenga la precaución de terminar las líneas con un '.'. Para obtener los resultados siguientes, pulse ';'.

He aquí una posible salida por consola:

```
1 ?- piel(X).
X = gato ;
X = raton.

2 ?- comer(gato,X).
X = raton.

3 ?- amigos(X,Y).
X = gato,
Y = queso ;
false.

4 ?- piel(queso).
false.

5 ?-
```

Preste atención tras cada modificación, asegúrese de guardar el archivo y recompilarlo para que se tengan en cuenta los cambios en la consola.