

INTELIGENCIA ARTIFICIAL

Fundamentos, práctica y aplicaciones

ALBERTO GARCÍA SERRANO

Alfaomega

libros R

INTELIGENCIA ARTIFICIAL

Fundamentos, práctica y aplicaciones

INTELIGENCIA ARTIFICIAL

Fundamentos, práctica y aplicaciones

Alberto García Serrano



Inteligencia Artificial. Fundamentos, práctica y aplicaciones
Alberto García Serrano

ISBN: 978-84-939450-2-2
EAN: 9788493945022
BIC: UYQ

Copyright © 2012 RC Libros
© RC Libros es un sello y marca comercial registrados

Inteligencia Artificial. Fundamentos, práctica y aplicaciones.
Reservados todos los derechos. Ninguna parte de este libro incluida la cubierta puede ser reproducida, su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes intencionadamente reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica, o su transformación, interpretación o ejecución en cualquier tipo de soporte existente o de próxima invención, sin autorización previa y por escrito de los titulares de los derechos de la propiedad intelectual.

RC Libros, el Autor, y cualquier persona o empresa participante en la redacción, edición o producción de este libro, en ningún caso serán responsables de los resultados del uso de su contenido, ni de cualquier violación de patentes o derechos de terceras partes. El objetivo de la obra es proporcionar al lector conocimientos precisos y acreditados sobre el tema tratado pero su venta no supone ninguna forma de asistencia legal, administrativa ni de ningún otro tipo, si se precisase ayuda adicional o experta deberán buscarse los servicios de profesionales competentes. Productos y marcas citados en su contenido estén o no registrados, pertenecen a sus respectivos propietarios.

RC Libros
Calle Mar Mediterráneo, 2. Nave 6
28830 SAN FERNANDO DE HENARES, Madrid
Teléfono: +34 91 677 57 22
Fax: +34 91 677 57 22
Correo electrónico: info@rclibros.es
Internet: www.rclibros.es

Diseño de colección, cubierta y pre-impresión: Grupo RC
Impresión y encuadernación: Villena Artes Gráficas
Depósito Legal: M-11290-2012
Impreso en España

16 15 14 13 12 (1 2 3 4 5 6 7 8 9 10 11 12)

ÍNDICE

PREFACIO ..... **VII**

CAPÍTULO 1. INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL **1**

¿QUÉ ES LA INTELIGENCIA ARTIFICIAL? 1

PERSPECTIVA HISTÓRICA 5

PRESENTE Y FUTURO 7

CAPÍTULO 2. IA Y RESOLUCIÓN DE PROBLEMAS **9**

RESOLVER PROBLEMAS 9

Otros problemas más complejos *11*

ALGUNOS TIPOS DE PROBLEMAS 15

El problema del viajante de comercio *15*

El problema de la satisfacibilidad booleana *19*

El problema de la programación lineal entera *23*

Otros problemas *26*

CAPÍTULO 3. BÚSQUEDA NO INFORMADA **29**

BÚSQUEDA 29

Representación de estados: árboles y grafos *31*

BÚSQUEDA EN AMPLITUD 34

BÚSQUEDA EN PROFUNDIDAD 45

BÚSQUEDA DE COSTE UNIFORME 56

CAPÍTULO 4. BÚSQUEDA INFORMADA **65**

FUNCIÓN HEURÍSTICA 65

Búsqueda con vuelta atrás (backtracking) 69
Algoritmo A* 73
Búsqueda local 83
Algoritmos constructivos voraces *86*
El algoritmo de Dijkstra 88
Algoritmo de Clarke y Wright 94
Hill climbing *102*
Simulated annealing *111*
Búsqueda tabú *117*
Algoritmos genéticos *126*
CAPÍTULO 5. JUEGOS 141
Inteligencia Artificial y juegos 141
El algoritmo minimax *143*
Poda alfa-beta *155*
Otros tipos de juegos *165*
CAPÍTULO 6. RAZONAMIENTO 171
Introducción 171
sistemas expertos 172
Sistemas difusos *176*
Conjuntos difusos *177*
Inferencia difusa *183*
CAPÍTULO 7. APRENDIZAJE 193
Introducción 193
Clasificación probabilística *194*
Un poco de probabilidad *194*
Clasificación bayesiana ingenua *200*
Redes neuronales Artificiales *208*
El perceptrón simple *210*
Redes neuronales multicapa *220*
Red de Hopfield *233*
APÉNDICE. EL LENGUAJE PYTHON 243
Introducción 243
El intérprete interactivo *244*
El primer programa *245*
Cadenas de caracteres *250*
Estructuras de control *255*
Secuencias *263*
Funciones *270*
Clases y objetos *274*
Módulos *280*
Paquetes *280*
ÍNDICE ALFABÉTICO 283

© RC Libros

Prefacio

La Inteligencia Artificial (o simplemente IA) es quizás unas de las disciplinas que más despiertan la imaginación de los que oyen hablar de ella por primera vez. Tal vez debido a que la ciencia ficción se ha encargado de crear ese halo de misterio que la envuelve y que nos invita a soñar con máquinas capaces de razonar, como el maravilloso HAL 9000 (*Heuristically Programmed Algoritmic Computer*) imaginado por Kubrick o incluso a sentir, como los replicantes de *Blade Runner*.

Si bien es cierto que las expectativas iniciales pueden hacer que alguien se decepcione con el estado del arte actual de la IA, no lo es menos que en las últimas décadas ha habido nuevos avances y se ha reactivado el interés por investigar en este campo. Sin duda, el auge de Internet y la necesidad de aplicar cierta "inteligencia" para poder manejar cantidades ingentes de información, como es el caso de buscadores como Google, han sido un buen catalizador.

Aunque a veces no sea demasiado evidente, la Inteligencia Artificial está presente cada vez más en nuestro día a día, ayudándonos casi sin darnos cuenta de ello. Cada vez que usamos una cámara fotográfica, un algoritmo de IA está identificando las caras de la imagen y enfocándolas. Algunas incluso reconocen que la persona ha sonreído para hacer el disparo. Tampoco es extraño ver cómo es posible controlar algunas funciones de nuestro vehículo o de nuestro teléfono móvil hablándoles directamente. Un complejo algoritmo de reconocimiento de voz está detrás de toda esta tecnología para facilitarnos la vida diaria.

No es probable que en pocos años veamos robots como R2D2 ayudándonos en casa, pero sí que podemos soñar con que no está lejos el día en que los vehículos nos

llevarán solos a nuestro destino, evitando las numerosas muertes que hoy provoca el tráfico por carretera. Tampoco es ninguna locura pensar que en pocos años dispondremos de asistentes personales para personas con movilidad reducida.

En definitiva, pocas áreas de la ciencia y la ingeniería invitan a imaginar el futuro tanto como la IA, y lo mejor de todo es que es ahora cuando se está inventando (y a veces reinventando) esta nueva ciencia, por lo que invito al lector a subirse a este tren cuyo destino es desconocido, pero sin duda apasionante.

Sobre este libro

Existe multitud de literatura en torno a la Inteligencia Artificial, la hay variada y a veces muy buena, sin embargo, la mayoría de los libros que tratan el tema son extremadamente teóricos. El nivel exigido para seguirlos y entenderlos suele ser muy alto. Además, casi toda la información se encuentra en inglés. Desde el primer momento, este libro ha sido concebido como un texto de introducción a la IA, pero desde una perspectiva práctica. No tiene sentido conocer toda la teoría que hay tras las redes neuronales si no somos capaces de hacer uso de todo ese conocimiento para aplicarlo en un problema concreto.

Con ese espíritu, se han implementado casi todos los algoritmos descritos en el libro y cuando no se ha hecho ha sido porque el código hubiera sido demasiado largo, en cuyo caso se ha mostrado paso a paso su funcionamiento con ejemplos. Se decidió usar el lenguaje Python en los ejemplos debido a su gran sencillez para ser leído, y por su gran expresividad, que hace que los programas sean más cortos ahorrando bastante espacio. Con la idea de que este libro sea lo más autocontenido posible, se ha añadido un apéndice con una introducción al lenguaje Python para facilitar la comprensión de los ejemplos.

A diferencia de otros libros sobre IA, se ha tratado de dejar las matemáticas reducidas a la mínima expresión; sin embargo, en una materia como esta no siempre es posible (ni conveniente) eliminarlas completamente. La dificultad no está más allá de un nivel de bachillerato, y en todo caso, muchas de las fórmulas y explicaciones matemáticas pueden obviarse sin peligro de no entender los algoritmos presentados. Siempre que ha sido necesario se ha hecho un breve repaso de las matemáticas involucradas, como es el caso del capítulo 7, donde se hace un breve repaso a la Probabilidad, necesaria para entender los clasificadores bayesianos.

El libro se ha dividido en siete capítulos que reúnen una serie de técnicas y algoritmos usados en IA. En el primer capítulo se hace una introducción a la IA para situarla en un contexto histórico y entender qué es y cómo se ha llegado a la

© RC Libros

situación actual. En el segundo se presenta el enfoque que seguiremos durante el resto del libro, definiendo qué es un problema en IA y cómo los podemos representar para que un ordenador pueda resolverlos.

En el tercer capítulo se introducen las técnicas clásicas de búsqueda para, seguidamente en el cuarto, introducir las técnicas heurísticas y de búsqueda local más utilizadas, desde las más clásicas a las más modernas, como los algoritmos genéticos.

El capítulo quinto se centra en los juegos de ordenador y los algoritmos que permiten hacer que las máquinas jueguen a juegos como el a*jedrez* o las d*amas*. En el sexto se hace una introducción al razonamiento mediante sistemas expertos y la lógica difusa. Y finalmente, el séptimo capítulo se dedica a las técnicas de aprendizaje, en concreto se introducen los métodos probabilísticos como los clasificadores bayesianos. También se hace una introducción a las redes neuronales y su implementación práctica.

Sobre el autor

El autor se inició en el mundo de la informática a una temprana edad cuando comenzó a "jugar" con el BASIC del Commodore 64. Desde entonces se ha dedicado al mundo de la informática y al del desarrollo de aplicaciones.

Alberto García es Ingeniero Técnico en Informática y autor de varios libros sobre informática y programación. Durante años impartió clases como docente en numerosos cursos y másteres sobre tecnología, informática y programación. Ha desarrollado la mayor parte de su trabajo en el área de la consultoría y el I+D, lo que le ha brindado la oportunidad de involucrarse en proyectos muy novedosos para grandes empresas de este país, desarrollando y poniendo en práctica parte de los conocimientos en IA que se recogen en este libro. Además de su labor profesional, escribe el blog http://divertimentosinformaticos.blogspot.com y su página personal es http://www.agserrano.com.

Agradecimientos

Quisiera agradecer a mi mujer Carmen su paciencia y su apoyo mientras escribía este libro por ayudarme con la lectura y correcciones del texto. También a mi hijo Diego por el tiempo que no he podido dedicarle mientras tecleaba estas páginas, pero que espero recuperar lo antes posible.

1 Introducción a la Inteligencia Artificial

¿QUÉ ES LA INTELIGENCIA ARTIFICIAL?

Por extraño que pueda parecer, lo cierto es que no hay un consenso entre los científicos e ingenieros sobre lo que es la Inteligencia Artificial, y mucho menos se ha llegado a una definición exacta y concisa que nos permita dirimir qué programas son o no inteligentes. El problema es que ni siquiera tenemos la certeza de que seamos capaces de definir qué es la inteligencia (no artificial).

Se dice que los humanos son una especie inteligente: saben hablar, resuelven problemas matemáticos, han llegado a la Luna… pero ¿acaso no son inteligentes el resto de animales que habitan el planeta? En efecto, un perro muestra comportamientos inteligentes e incluso es capaz de expresar sentimientos. ¿Dónde poner el límite de lo que es o no inteligente entonces?

El primer intento de definir la Inteligencia Artificial lo hizo el matemático Alan Turing, que es considerado como el padre de la computación. Este científico inglés es más conocido por su máquina de Turing: una máquina conceptual que utilizó para formalizar los conceptos del modelo computacional que seguimos utilizando hoy día. En concreto demostró que con las operaciones básicas que podía desarrollar su máquina podía codificarse cualquier algoritmo, y que toda máquina capaz de computar tendría las mismas operaciones básicas que su máquina o un superconjunto de estas.

En 1950 Turing publicó un artículo llamado *Computing machinery and intelligence* donde argumentaba que si una máquina puede actuar como un humano, entonces podremos decir que es inteligente. En el artículo proponía una prueba, llamada Test de Turing, que permitiría afirmar si una máquina es o no inteligente. Para llegar a esa conclusión, un ser humano se comunicaría a través de un terminal informático con una entidad que se hallaría en una habitación contigua. Esta entidad podría ser un humano o una máquina inteligente. Si tras una conversación la persona no es capaz de distinguir si lo que hay en la otra habitación es un humano o una máquina, entonces, en caso de ser una máquina, la podemos considerar inteligente.

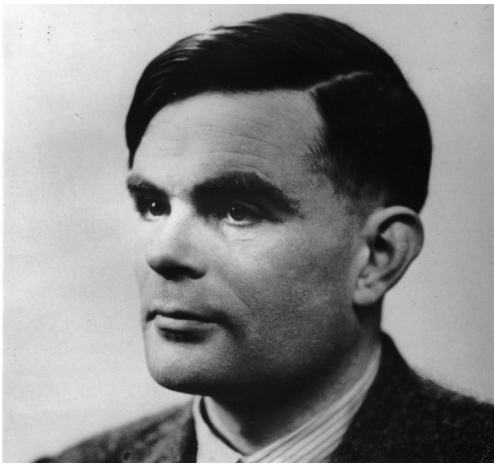


Fig. 1-1 Retrato de Alan Turing.

El Test de Turing, pese a los años que han pasado, tiene una gran importancia, ya que exige una serie de capacidades a la máquina inteligente cuyo conjunto conforma, a grandes rasgos, lo que es la Inteligencia Artificial hoy día. En efecto, una máquina que sea capaz de pasar el Test de Turing ha de tener las siguientes capacidades.

- Reconocimiento del lenguaje natural.
- Razonamiento.
- Aprendizaje.
- Representación del conocimiento.

Además, existe una prueba llamada Test de Turing Total en la que la terminal informática que permite la comunicación dispone de cámara de vídeo e imagen, por

© RC Libros

lo que la comunicación se produce como si fuera una videoconferencia. También se permite el paso de objetos a través de una compuerta. Para pasar esta prueba, una máquina ha de tener dos capacidades adicionales.

- Visión.
- Robótica.

Efectivamente, la máquina tiene que ser capaz de reconocer el lenguaje natural en el que hablamos los humanos. El habla se asocia a una inteligencia superior, y para que una máquina sea capaz de reconocerla y también de construir frases tiene que ser capaz de realizar complejos análisis morfológicos, sintácticos, semánticos y contextuales de la información que recibe y de las frases que genera. En la actualidad, el **procesamiento del lenguaje natural** o **NLP** (*Natural Language Processing*) es una rama de la Inteligencia Artificial que se ocupa de las capacidades de comunicación de los ordenadores con los humanos utilizando su propio lenguaje. Es un área cuyas aplicaciones son múltiples y variadas, como la traducción automática o el reconocimiento y comprensión del lenguaje humano entre otros.

La prueba propuesta por Turing exige también una capacidad de **razonamiento automático**. Los humanos somos capaces de llegar a conclusiones a partir de una serie de premisas. Por ejemplo, un humano puede ser capaz de llegar a la conclusión de que si está lloviendo, el suelo estará mojado y en consecuencia es muy probable que esté resbaladizo. Un primer intento de conseguir que las máquinas razonaran fue llevado a la práctica mediante los llamados sistemas expertos. Estos tratan de llegar a conclusiones lógicas a partir de hechos o premisas introducidas a priori en el sistema. Actualmente, se utilizan otras técnicas más versátiles como las redes probabilísticas, que permiten hacer predicciones y llegar a conclusiones incluso cuando hay cierto nivel de incertidumbre en las premisas.

El **aprendizaje automático** es también condición necesaria para que un ente artificial pueda ser considerado inteligente. Si una máquina no es capaz de aprender cosas nuevas, difícilmente será capaz de adaptarse al medio, condición exigible a cualquier ser dotado de inteligencia. En Inteligencia Artificial, las líneas de investigación actuales buscan hacer que las máquinas sean capaces de hacer generalizaciones a partir de ejemplos sacados del entorno. Por ejemplo, un niño aprende desde edad muy temprana que si se cae al suelo puede hacerse daño, y para llegar a esta generalización, primero tiene que caerse varias veces (es lo que consideramos ejemplos en IA). Actualmente, se utilizan técnicas basadas en redes y métodos probabilísticos como las redes bayesianas o de Markov y también se trata de simular el funcionamiento del cerebro humano a través de las redes neuronales.

© RC Libros

Evidentemente, para tener capacidad de razonamiento y aprendizaje, la computadora ha de ser capaz de almacenar y recuperar de forma eficiente la información que va obteniendo o infiriendo autónomamente, es decir, necesita mecanismos de **representación del conocimiento**. Por sí sola, esta es una rama de la Inteligencia Artificial que investiga las técnicas de almacenamiento de información de forma que sea fácilmente accesible y, sobre todo, utilizable por los sistemas inteligentes. De nada sirve almacenar datos si luego los sistemas no pueden acceder a ellos de forma que sean capaces de usarlos para sacar conclusiones u obtener nueva información que no poseían de forma directa.

Para pasar el Test de Turing Total, la máquina tiene que tener capacidades de **visión artificial** y de manipulación de objetos, lo que en Inteligencia Artificial se denomina **robótica**. A través de una cámara, un ordenador puede ver el mundo que le rodea, pero una cosa es obtener imágenes y otra entenderlo. Un humano puede ser capaz de leer un texto separando las letras, palabras y frases que lo componen y atribuyéndoles un significado. La visión artificial busca que los sistemas inteligentes sean capaces de interpretar el entorno que les rodea a partir de imágenes.

Por otro lado, la capacidad de manipular objetos debe hacer uso de la visión artificial (o, alternativamente, otro tipo de sensores) que permitan saber al ordenador dónde está el objeto, qué forma tiene, hacerse una idea del peso del mismo, etc. Además, debe ser capaz de aplicar la presión exacta para no dañar el objeto y saber qué movimientos ha de hacer para trasladarlo a su nuevo destino. Tareas nada sencillas para un ordenador.

Frente a la idea que proponía Turing de que una máquina será inteligente si actúa como un humano, otros investigadores y autores proponen nuevos paradigmas. Uno de ellos afirma que si una máquina piensa como un humano, entonces será inteligente. Otros, sin embargo, defienden la idea de que una máquina será inteligente si piensa o actúa de forma racional.

La afirmación de que una máquina es inteligente si piensa como un humano puede parecer muy similar a la idea que tenía Turing sobre la inteligencia, pero existe una gran diferencia. Comportarse como un humano no significa necesariamente que las máquinas recreen internamente el mismo proceso mental que ocurre en el cerebro humano; sin embargo, pensar como un humano implica que primero debemos saber cómo piensa realmente un humano. Las ciencias cognitivas tratan de descifrar cómo pensamos y cómo procesamos la información que llegan a nuestros sentidos. Lo cierto es que aún estamos lejos de comprender todo el mecanismo cerebral, por lo que este enfoque se antoja complicado. A pesar de ello, con lo que ya

© RC Libros

sabemos, es posible mejorar y crear nuevos algoritmos capaces de ampliar las fronteras de la Inteligencia Artificial.

Finalmente, los que proponen el modelo racional, es decir, consideran que una máquina es inteligente si piensa o se comporta racionalmente, basan sus técnicas en la lógica y en el concepto de **agentes**. Según este enfoque, con una gran aceptación en la actualidad, los actos de un agente inteligente deben basarse en el razonamiento y en las conclusiones obtenidas a partir de la información que se posee. Estos agentes tomarán la decisión más conveniente a la vista de esos datos y del tiempo del que disponen: no es lo mismo tomar una decisión disponiendo de toda la información y todo el tiempo necesario que tomarla con información incompleta (incertidumbre) y poco tiempo para procesarla.

Actualmente, las principales líneas de investigación trabajan sobre el concepto de agente inteligente y lo hacen extensivo a grupos de agentes, que pueden tener capacidades diferentes, y que trabajan de forma conjunta colaborando entre ellos para resolver un problema. Son los llamados **sistemas multiagente**.

El autor del presente texto no cree que exista un paradigma definitivo que sea capaz de conseguir el objetivo final de la Inteligencia Artificial, que es construir entidades inteligentes. Si bien, ninguno ha demostrado ser capaz de dotar de inteligencia completa a una máquina, todos han aportado información y técnicas valiosas.

En este libro tomaremos un enfoque mucho más pragmático, tal y como se espera de un texto que pretende ser introductorio y, sobre todo, práctico. Consideraremos la Inteligencia Artificial como un conjunto de técnicas, algoritmos y herramientas que nos permiten resolver problemas para los que, a priori, es necesario cierto grado de inteligencia, en el sentido de que son problemas que suponen un desafío incluso para el cerebro humano.

PERSPECTIVA HISTÓRICA

Cualquier ciencia incipiente hunde sus raíces en otras materias, y en este caso, filósofos como Aristóteles con sus silogismos o el español Ramón Llull que en 1315 ya hablaba de máquinas capaces de razonar como las personas, pusieron los cimientos sobre los que construir esta nueva disciplina.

© RC Libros

Situar el principio de la Inteligencia Artificial es bastante complicado; sin embargo, parece haber cierto consenso en que Warren McCulloch y Walter Pitts dieron el pistoletazo de salida a esta joven ciencia en 1943 gracias a sus trabajos en los que propusieron el primer modelo de red neuronal artificial. Era un modelo bastante simple, pero McCulloch y Pitts demostraron que era capaz de aprender y resolver funciones lógicas. Curiosamente, el estudio de las redes neuronales sufrió un buen parón en los años siguientes hasta que a mediados de los 80 se retomó la investigación en este campo gracias a los avances y éxitos que tuvieron diversos grupos usando redes de retropropagación.

Sin embargo, la Inteligencia Artificial como tal no era muy reconocida en ambientes universitarios ni grandes grupos de investigación. Más bien era residual el esfuerzo dedicado en este campo, hasta que en 1950, Alan Turing publicó su artículo *Computing machinery and intelligence*. Años después Turing sería coautor del primer programa capaz de jugar al ajedrez. A partir de ese momento comenzó a consolidarse el interés que iba creciendo junto a los avances que se hacían en el campo de las computadoras.

A pesar de que se considera a Turing como el padre de esta disciplina, el término Inteligencia Artificial fue acuñado en 1958 por John McCarthy, que a la postre inventaría el lenguaje LISP, considerado el lenguaje de la Inteligencia Artificial. McCarthy, con la ayuda de científicos como Minsky, Claude, Shannon o Newell, convocaron una conferencia de dos meses en *Dartmounth College*, donde se reunieron los mejores investigadores en este campo, dando lugar a la era moderna de la IA y a alcanzar el estatus de ciencia a esta disciplina.

Desgraciadamente, el impulso inicial y la euforia que supuso esta conferencia se fueron desinflando poco a poco al constatarse que ninguna de las expectativas puestas en la IA se iba cumpliendo. Algunas máquinas eran capaces de jugar al ajedrez, de resolver problemas fáciles o hacer razonamientos sencillos, pero nada hacía vislumbrar el nacimiento de una máquina pensante. Los trabajos e investigaciones fueron decayendo y, salvo algunos trabajos aislados, como los primeros trabajos con algoritmos genéticos a finales de los 50, pocas universidades invertían tiempo y dinero.

A principio de los 80, los sistemas expertos encendieron de nuevo una pequeña llama de esperanza, sobre todo porque tenían una aplicación directa en la industria. La industria americana invirtió millones de dólares en crear sistemas expertos de todo tipo, pero de nuevo, las expectativas parecían desvanecerse de nuevo, hasta que a mediados de los 80 las redes neuronales resurgieron para dar un nuevo giro de timón a la disciplina.

© RC Libros

Ya en los 90, y bien asentada como ciencia y con bastantes líneas de investigación abiertas, nuevos avances como las redes ocultas de Markov, las redes probabilísticas como las redes bayesianas, los agentes inteligentes y otras técnicas novedosas están llevando a la Inteligencia Artificial a un nuevo nivel en el que sus aplicaciones comienzan a estar cada vez más presentes en las vidas cotidianas de las personas.

PRESENTE Y FUTURO

En la actualidad, son muchas las universidades que investigan en este campo, pero cada vez más empresas como Google, gobiernos y otras instituciones invierten grandes cantidades de dinero. Mucha culpa de todo esto la tiene internet, cuya gran cantidad de información facilita el acceso a grandes cantidades de datos analizables y a su vez demanda de nuevas técnicas que permitan manejar tales cantidades de información. Google es un ejemplo claro, cuyo buscador es capaz de ir aprendiendo de los *clics* de los usuarios para mejorar los resultados de las búsquedas. También usa la gran información que maneja para ofrecernos un traductor muy potente que también aprende.

Otro ejemplo de la utilidad de la IA en Internet es la lucha contra el *spam* o correo basura. En efecto, el uso de clasificadores bayesianos y otras técnicas probabilísticas se muestran muy eficaces contra este problema.

El mercado de consumo tampoco deja escapar el tren y actualmente los teléfonos inteligentes (*smartphones*) comienzan a hacer uso del reconocimiento facial como es el caso de los teléfonos Android o el reconocimiento del habla del que hace uso Apple en su iPhone con su sistema *siri*.

La empresa automovilística también trabaja actualmente en coches autónomos que conducen solos o que son capaces de prever situaciones complicadas y actuar en consecuencia. También están incorporando el reconocimiento del habla, lo que nos permitirá en un futuro no muy lejano hablar con el coche de forma natural.

Los usos militares, aunque la mayoría de las veces permanecen en secreto, son visibles, sobre todo en robótica, donde aviones no tripulados y robots que desactivan minas son solo la punta de iceberg.

En la actualidad, ya nadie pone en duda la supremacía de las máquinas sobre los humanos en juegos como el ajedrez, y miles de empresas utilizan sistemas de planificación para organizar los repartos de mercancías o planificar los procesos industriales. Sin duda, el gran ahorro que suponen compensan las inversiones necesarias.

Podríamos seguir poniendo ejemplos de usos de la IA, pero seguro que es posible encontrar tantos que nos dejaríamos muchos en el tintero. Tampoco es seguro saber qué nos deparará el futuro, ya que esa tarea tiene más que ver con la literatura de ciencia ficción.

Empresas como Numenta exploran nuevos campos y nuevas aproximaciones al desarrollo de sistemas inteligentes. En concreto uno de sus fundadores, Jeff Hawkins, propone un nuevo modelo de máquina inteligente basada en el funcionamiento del neocórtex humano. Si este modelo dará o no sus frutos en un futuro, solo el tiempo lo dirá.

© RC Libros

2 IA y resolución de problemas

RESOLVER PROBLEMAS

En el capítulo anterior hemos visto que, a pesar de ser una ciencia relativamente joven y aún en fase temprana en comparación con otros campos, la IA es ya hoy una herramienta que nos ayuda a afrontar la resolución de problemas usando un enfoque muy distinto al de la computación clásica. No obstante, no puede verse la IA como la receta mágica capaz de resolver cualquier problema, sino como un conjunto de técnicas que por sí solas o en combinación con otras nos ayudarán a encontrar una solución (no necesariamente la mejor) a un problema cuya resolución es compleja e incluso inabordable por una persona humana.

No vamos a definir lo que es un "Problema", ya que esto entra más dentro del campo de la Filosofía que de la Informática. Aun así, vamos a quedarnos con la idea intuitiva de que un problema es una cuestión difícil de solucionar, es decir, que no tiene una solución trivial.

Todos recordamos de nuestro paso por el colegio aquellos ejercicios que teníamos que resolver en clase de Matemáticas. Vamos a proponer un par de problemas de nivel de primaria e infantil.

Problema 1: Juan tiene 5 caramelos. De camino a casa pierde 1 caramelo que cae del bolsillo. Cuando sube las escaleras de su portal pierde de nuevo otro caramelo. Finalmente, al llegar, se encuentra 3 caramelos que se ha dejado olvidados sobre la mesa su hermano Víctor. ¿Cuántos caramelos tiene Juan?

Problema 2: Ordene en orden descendente la siguiente lista de números: 3,6,1,3,9,4,2

Son dos problemas muy simples, pero ¿cómo se enfrenta nuestro cerebro a ellos? Si ha intentado buscar la solución de ambos, habrá observado que las técnicas que ha usado para resolverlos son bien distintas a pesar de ser de una dificultad similar.

En el primer problema, seguramente, ha tenido que contextualizar las situaciones. Seguramente se ha imaginado a Juan perdiendo un caramelo mientras subía la escalera. Finalmente, ha reducido el enunciado a una simple operación con sumas y restas. En definitiva, todo el párrafo que describía el problema ha quedado reducido a la operación:

Caramelos = 5-1-1+3 = 6

Evidentemente, una vez reducido el problema a una operación matemática habrá tenido que aplicar unas herramientas de las que ya disponía desde la infancia, como son las operaciones matemáticas.

El acercamiento al segundo problema ha debido ser algo diferente. En este caso no ha tenido que hacer operaciones matemáticas (al menos directamente), sino que la tarea consistía en comparar los valores de dos números para saber cuál debía colocar en primer lugar. Repitiendo esta operación, habrá acabado ordenando todos los números.

El cerebro, en este segundo caso, ha usado unas herramientas diferentes a las utilizadas para el primero. En todo caso, al igual que en el primero se conceptualizó todo el enunciado en una sola operación matemática, en el segundo también se ha reducido un problema complejo (ordenar una lista de números), a una secuencia de operaciones más simples (ordenar sucesivamente un par de números hasta tener toda la lista ordenada).

En ambos casos, el cerebro ha recogido la información que necesitaba dejando de lado todo aquello que le era superfluo (no importa si el caramelo se cayó subiendo las escaleras o en la calle) y lo ha traducido a un modelo mental que sí es capaz de manejar.

Sin embargo, no todos los problemas son tan sencillos. Vamos a plantear algunos problemas algo más complejos y que nos van a servir de base para seguir indagando en la resolución de problemas.

© RC Libros

Otros problemas más complejos

Vamos a enunciar algunos problemas que nos servirán de modelo a seguir durante el resto del libro.

Si ha hecho algún viaje largo alguna vez en avión, seguramente ha intentado viajar al destino directamente y sin escalas. Este es un problema clásico y con una gran aplicación práctica. El problema podría enunciarse de la siguiente manera: Dada una ciudad de origen y otra de destino, encontrar una combinación de vuelos que nos permita cubrir la distancia con el mínimo número de trasbordos. Por ahora vamos a obviar tiempos de vuelo, distancias, horarios de los vuelos y todo lo demás (esto lo trataremos más adelante).

Vamos a enunciar otro problema similar. En esta ocasión se trata de ir de una ciudad a otra pero esta vez por carretera y tratando de realizar el menor número de kilómetros posible. Parece que es un problema similar al de los aviones, solo que en lugar de pasar por aeropuertos pasamos por diversas ciudades en el camino que va del origen al destino. Sin embargo, hay un pequeño detalle que lo hace un problema muy diferente. Y es que los objetivos son distintos.

En el primer problema teníamos un objetivo: minimizar el número de trasbordos. En el segundo problema lo que tratamos de minimizar es la distancia por carretera. Veámoslo con un ejemplo.

En la figura 2-1 vemos un esquema con algunos aeropuertos españoles y unas conexiones ficticias a través de vuelos para un día concreto.

Si quisiéramos viajar desde Málaga a Santiago, podríamos optar por las siguientes rutas:

Málaga > Barcelona > Santiago (3 aeropuertos)

Málaga > Madrid > Santander > Santiago (4 aeropuertos)

Málaga > Madrid > Sevilla > Santiago (4 aeropuertos)

© RC Libros

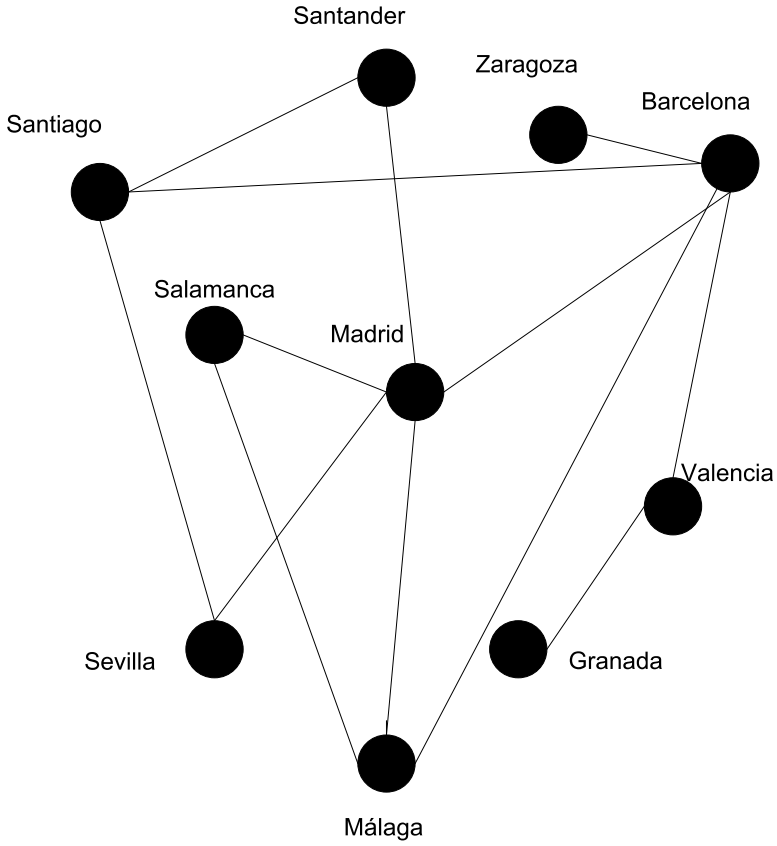


Fig. 2-1 Conexiones de vuelos

En este caso la mejor opción es la primera (Málaga > Barcelona > Santiago), según nuestro criterio de realizar el mínimo número de trasbordos.

Si quisiéramos viajar desde Málaga a Granada, a pesar de que las ciudades son cercanas, necesitamos visitar nada menos que cuatro aeropuertos:

Málaga > Barcelona > Valencia > Granada

Pero ¿qué ocurre si nos planteamos esos mismos viajes por carretera y con el objetivo de minimizar la distancia recorrida?

En el siguiente esquema vemos las mismas ciudades y sus conexiones por carretera junto con sus distancias.

© RC Libros

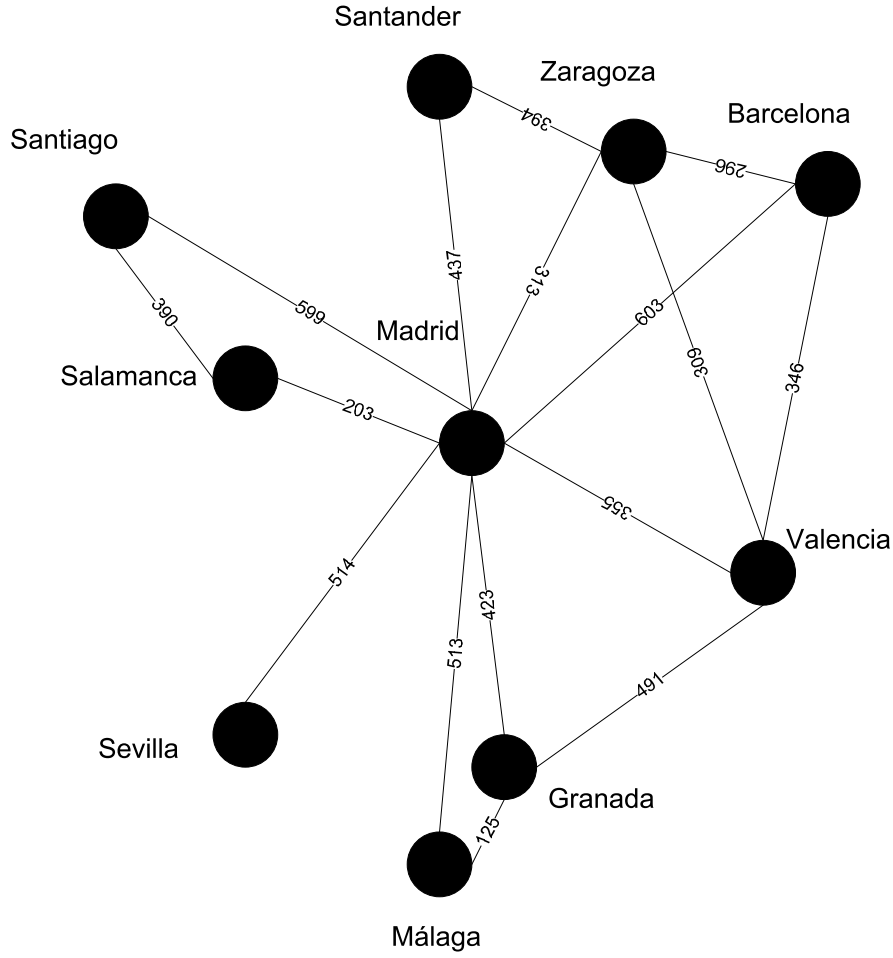


Fig. 2-2 Mapa de carreteras

Supongamos de nuevo que queremos ir de Málaga a Santiago. Estas son las posibles opciones:

Málaga > Madrid > Santiago (1.112 km)

Málaga > Madrid > Salamanca > Santiago (1.139 km)

¿Cuál es el mejor recorrido? Hemos de definir un criterio que nos diga cómo de bueno es un recorrido. En este caso hemos escogido la suma total de kilómetros, que arroja que el mejor recorrido es el primero porque permite llegar al destino cubriendo menos distancia.

© RC Libros

A la vista del breve análisis de los problemas que hemos planteado, podemos intuir que hay tres conceptos importantes que definen un problema. Por un lado, hemos visto cómo necesitamos crear un **modelo** simplificado que represente al problema real. Por ejemplo, reduciendo el enunciado del problema de Juan y los caramelos a simples operaciones matemáticas.

También hemos llegado a la conclusión de que hay que tener muy claro el **objetivo** que buscamos. Es decir, ¿cuándo damos el problema por resuelto? ¿Cuáles serán las soluciones válidas? En los ejemplos del viaje en avión y por carretera, ¿cuáles eran los objetivos? Para el primero, encontrar las conexiones con el número de trasbordos, y para el segundo, encontrar el recorrido con el menor número de kilómetros.

Finalmente, hemos tenido que definir un criterio que valore cómo de buena es cada una de las soluciones del problema, para poder compararla con las demás y quedarnos con la mejor. En el caso de los viajes en avión hemos tomado el criterio del número de aeropuertos visitados. En el problema de los viajes por carretera nuestro criterio ha sido la suma total de kilómetros. Este criterio que nos da información de la calidad de una solución lo llamaremos a partir de ahora **función de evaluación**.

Por lo tanto, para definir un problema necesitamos un modelo, un objetivo y una función de evaluación.

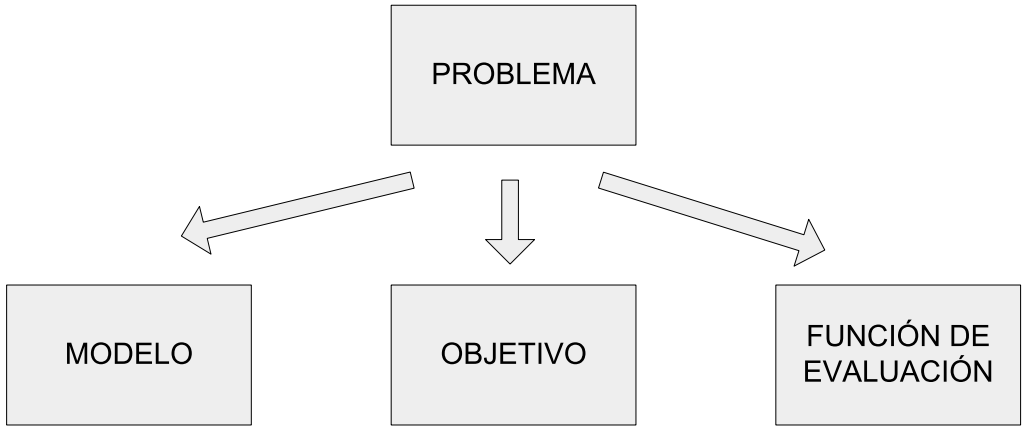


Fig. 2-3 Definición de un problema

© RC Libros

ALGUNOS TIPOS DE PROBLEMAS

Antes de comenzar a plantear una solución a un problema, hemos visto en el apartado anterior que es necesario conceptualizarlo, eliminar información redundante y utilizar estructuras que sabemos manejar (como operaciones matemáticas) para modelarlo. El modelo, para ser útil, debe ser una simplificación del modelo real.

Hay que tener muy claro que al construir un modelo, lo que vamos a conseguir es una solución al mismo, y no al problema real. Por lo tanto, cuanto mejor describa el modelo al problema que queremos resolver, tanto mejor se ajustará esa solución a la del problema real.

Cuando hacemos una conceptualización del problema, necesitamos una forma de representar la información de la que disponemos. Cuando finalmente hagamos la implementación en el ordenador, este modelo tendrá que ser traducido a estructuras de datos y operaciones que el ordenador sea capaz de manejar.

Al igual que en la sección anterior, vamos a utilizar unos ejemplos para después sacar algunas conclusiones útiles.

El problema del viajante de comercio

El primer problema que vamos a tratar es bastante conocido dentro de la IA y es conocido como “El problema del viajante de comercio” o también por sus siglas en inglés **TSP** (*Travelling Salesman Problem*). La formulación del problema es sencilla: Sean N ciudades, todas conectadas por carretera entre sí y cuyas distancias entre ellas es conocida. Nuestro objetivo es encontrar una ruta que, empezando en una ciudad y acabando en la misma, pase exactamente una vez por cada una de las ciudades (excepto por la primera, que será visitada dos veces) y minimice la distancia recorrida.

Tras este simple enunciado se esconde un problema tremendamente difícil. Tanto que, incluso con los ordenadores más potentes que existen hoy, somos incapaces de resolverlo. Incluso para un número de ciudades moderado. Aun así, podemos conseguir aproximaciones muy buenas utilizando diferentes técnicas de IA, algunas de las cuales van a ser estudiadas en los siguientes capítulos.

Vamos a comenzar por construir un modelo que el ordenador pueda manejar. Empecemos por el mapa en sí. Hay varias opciones. Por ejemplo, podemos optar por representar las ciudades mediante un dígrafo ponderado. En este caso, cada nodo

del grafo representará una ciudad, y cada arista corresponderá a una carretera con su distancia (peso) asociada. Debe tenerse en cuenta que para el caso del TSP, la distancia a cubrir desde una ciudad A hasta otra B no tiene por qué ser la misma que de B hasta A. De todas formas, para simplificar el ejemplo vamos a suponer que las distancias entre dos ciudades son iguales con independencia del sentido en que vayamos.

En la figura 2-4 se ha representado un esquema de un hipotético mapa de carreteras para cinco ciudades, y su representación como grafo.

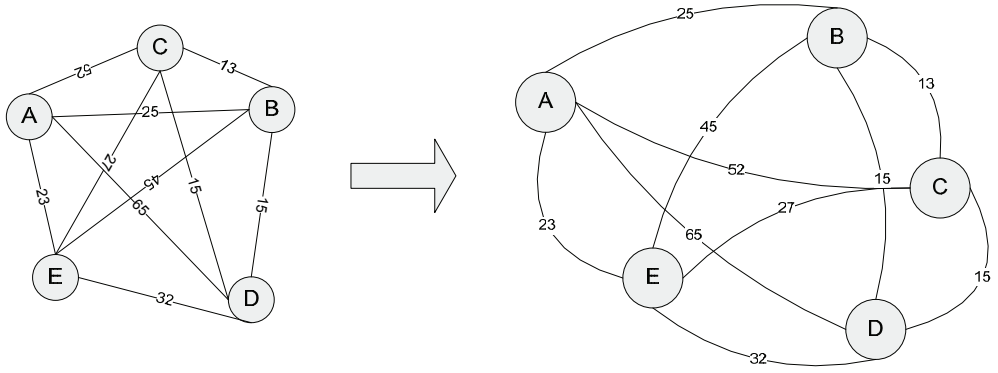


Fig. 2-4 Transformación del mapa en un grafo

Otra posible representación es utilizar una tabla (también llamada matriz de adyacencia) como la siguiente.

	A	B	C	D	E
A	0	25	52	25	23
B	25	0	13	15	45
C	52	13	0	15	27
D	25	15	15	0	32
E	23	45	27	32	0

© RC Libros

Siempre que nos sea posible elegiremos la representación más simple e intuitiva, a no ser que en términos de eficiencia sea más conveniente utilizar otra estructura de datos más compleja. En el caso que nos ocupa, parece que la tabla es más interesante, ya que puede ser modelada más fácilmente con un array de dos dimensiones. La implementación en forma de grafo no aporta nada, ya que se trata de una estructura de datos estática que no cambia durante la ejecución. Si la información fuera cambiante durante el tiempo de ejecución (por ejemplo, un problema en el que fuéramos descubriendo nuevas ciudades según avanza la ejecución), entonces sí sería interesante usar el grafo como estructura de datos dinámica.

Una vez modelada la información del problema, necesitamos una forma de representar la solución, es decir, nuestro objetivo. Una forma simple y directa de representarlo es como una lista de ciudades que cumpla las siguientes premisas:

- La lista comienza y termina por la misma ciudad.
- En la lista aparecen todas las ciudades.
- No se repite ninguna ciudad, excepto la primera y la última.

Cualquier lista de ciudades que cumpla las tres premisas anteriores es una posible solución. Sin embargo, sabemos que no todas las soluciones son iguales. De hecho buscamos la ruta con menor distancia. Necesitamos pues una función de evaluación que nos informe de la calidad de una solución para poder compararla con otras soluciones. Para el TSP usaremos la suma de las distancias de cada una de las ciudades por las que vamos pasando. Por ejemplo, estas son posibles soluciones y el valor obtenido por la función de evaluación.

A – B – E – D – C – A (función de evaluación: 169)

A – E – D – C – B – A (función de evaluación: 108)

A – C – D – E – B – A (función de evaluación: 169)

¿Cómo encontrar la mejor solución? En teoría podríamos proceder de la siguiente manera:

- Generamos todas las posibles soluciones.
- Comparamos unas con otras y nos quedamos con la que arroje un valor menor de la función de evaluación.

© RC Libros

Analicemos con detalle el primer paso. ¿Cómo generamos todas las posibles soluciones? ¿Cuántas soluciones posibles hay?

Una forma de generar las posibles soluciones es partir de una lista de ciudades cualquiera e ir permutándolas entre ellas. Cada permutación será una posible solución al problema. Vamos a suponer que la lista de ciudades es la siguiente:

A – B – E – D – C – A (función de evaluación: 169)

Si hacemos el cambio entre las ciudades B y E obtenemos:

A – E – B – D – C – A (función de evaluación: 150)

Si ahora efectuamos el cambio de E y D sobre la primera ruta, obtenemos:

A – D – E – B – C – A (función de evaluación: 207)

Y así sucesivamente hasta obtener todas las combinaciones posibles. ¿Pero cuántas hay? Si tenemos N ciudades, tendremos N! combinaciones posibles. N! (leído N factorial) es igual a:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

A cada posible solución la llamaremos **estado**, y al conjunto de todas las posibles soluciones lo denominaremos **espacio de estados**. En nuestro ejemplo con cinco ciudades tenemos 120 posibles soluciones o estados. Un número manejable por cualquier ordenador personal.

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Con 7 ciudades tendríamos 5.040 posibles soluciones.

Con 10 ciudades tendríamos 3.628.800 posibles soluciones.

Con 20 ciudades tendríamos 2.432.902.008.176.640.000 posibles soluciones.

Para hacernos una idea, si cada permutación le ocupara al ordenador 1 microsegundo, con 20 ciudades necesitaríamos 77.147 años para generar todo el espacio de estados. En la figura 2-5 podemos observar cómo de rápido es el crecimiento del espacio de estados según el número de ciudades.

© RC Libros

A este tipo de problemas los denominamos **intratables** y decimos que pertenecen a la **clase NP**, que sin entrar en detalle diremos que son una clase de problemas cuya complejidad hace que no sean resolubles en un tiempo razonable por un ordenador.

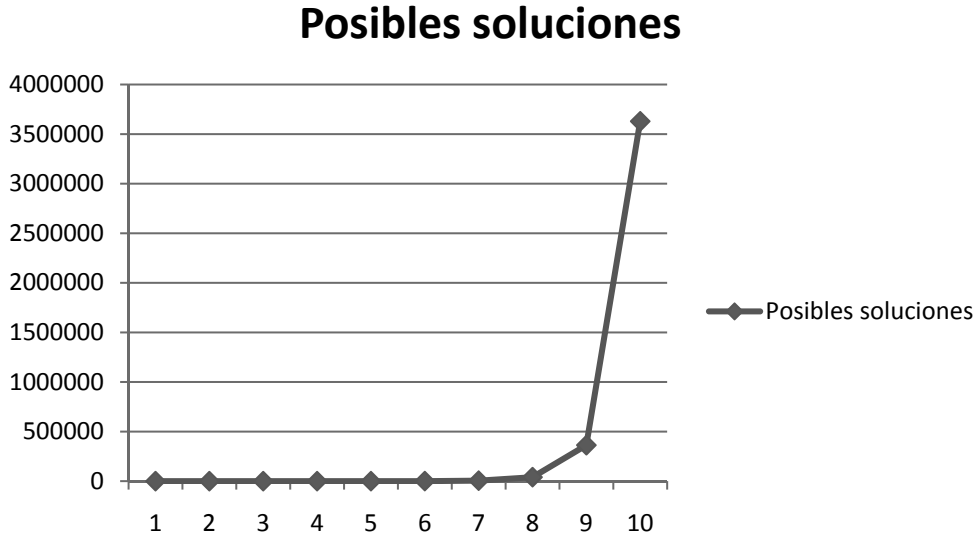


Fig. 2-5 Soluciones posibles según el número de ciudades

Por lo tanto, queda claro que con métodos computacionales clásicos no es posible resolver este problema. Precisamente la IA trata de cómo abordar este tipo de problemas a través de diferentes técnicas que serán tratadas en sucesivos capítulos. Quizás no podamos crear un programa que nos asegure el mejor resultado posible, pero como veremos, sí podremos ofrecer una solución cuasi-óptima o al menos muy buena en un tiempo razonable.

El problema de la satisfacibilidad booleana

Tras este largo y estrambótico nombre se encuentra otro problema clásico. El problema de la satisfacibilidad booleana o, para abreviar, **SAT** (Boolean satisfiability problem) tiene el honor de ser el primer problema conocido de la clase NP. Aun así, fue solo hace 4 décadas, en 1971, cuando se demostró que SAT era un problema intratable.

© RC Libros

Antes de enunciar el problema recordemos algunos conceptos básicos de lógica booleana. Una variable booleana es aquella que puede tomar dos valores diferentes: verdadero y falso. También se pueden representar como 0 y 1 o como V y F.

Sobre estas variables podemos realizar varias operaciones, pero nos centraremos en las tres principales: conjunción, disyunción y negación.

La conjunción, también conocida como operación AND y representada por el símbolo ^, es una operación binaria (en el sentido de que opera dos variables) y que devuelve el valor verdadero solo cuando ambos operadores son verdaderos. La siguiente tabla (llamada tabla de verdad) muestra el resultado de operar con la operación conjuntiva (AND).

OP1	OP2	OP1^OP2
F	F	**F**
F	V	**F**
V	F	**F**
V	V	**V**

La disyunción u operador OR se representa por el símbolo ˅, y devuelve el valor verdadero si alguno de los operadores es verdadero, según la tabla siguiente.

OP1	OP2	OP1˅OP2
F	F	**F**
F	V	**V**
V	F	**V**
V	V	**V**

© RC Libros

Por último, el operador unario (opera sobre una sola variable) de negación u operador NOT se representa por el símbolo ¬ delante de la variable. Su tabla de verdad se muestra en la siguiente tabla.

OP	¬OP
F	**V**
V	**F**

Podemos construir funciones booleanas cuyo resultado depende de las variables que la componen. Por ejemplo, sea la función

$$f(x_{1,}x_{2,}x_{3,})=(x_2 \wedge x_1) \vee \neg x_3$$

Si representamos el valor verdadero como T y falso como F, las siguientes asignaciones de variables darían como resultado los siguientes valores.

$$f(F,V,V)=(V \wedge F) \vee \neg V = F$$

$$f(F,V,F)=(V \wedge F) \vee \neg F = V$$

$$f(V,V,V)=(V \wedge V) \vee \neg V = V$$

Las funciones booleanas suelen expresarse de forma normalizada para que sea más fácil operar con ellas. Las dos formas habituales son la **forma normal conjuntiva (FNC)** y la **forma normal disyuntiva (FND)**. La FNC se expresa como agrupaciones de disyunciones, llamadas **cláusulas**, unidas por conjunciones de la siguiente manera.

$$(x_2 \vee x_4) \wedge (x_3 \vee x_1) \wedge (x_2 \vee \neg x_1)$$

En cambio, la FND tiene el siguiente aspecto:

$$(x_2 \wedge x_4) \vee (x_3 \wedge x_1) \vee (x_2 \wedge \neg x_1)$$

Ambas formas son igualmente válidas, pero nosotros usaremos preferentemente la FNC en este libro.

© RC Libros

Una cláusula puede tener un número cualquiera de variables, y a su vez, una función booleana en FNC o FND puede tener tantas cláusulas como sea necesario.

Finalizado este breve recordatorio de lógica booleana, estamos listos para enunciar el problema SAT.

Siendo f una función booleana en FNC con n variables, queremos saber si existe una asignación para las variables tal que haga verdadera a la función f. Por ejemplo, consideremos la función f siguiente:

$$f(x_1,x_2,x_3,x_4)=(x_2 \vee x_4)\wedge(x_3 \vee x_1)\wedge(x_2 \vee \neg x_1)$$

¿Existe una asignación para las variables x_1,x_2,x_3,x_4 que haga verdadera la función f? En este ejemplo podemos encontrar una asignación de variables sin tener que pensar demasiado.

Por ejemplo, la asignación $x_1 = F, x_2, = V, x_3 = V, x_4 = F$ satisface las restricciones del problema.

Si ahora quisiéramos resolver el problema SAT con una función de tres variables por cláusula (problema conocido como 3-SAT) y 100 variables booleanas, la cosa se complica un poco. Vamos a plantear un modelo, un objetivo y una función de evaluación.

En este caso el modelo es bastante directo. Podemos representar las asignaciones de las variables como cadenas de valores 0 y 1. Por ejemplo, $x_{1=F},x_{2=V},x_{3=V},x_{4=F}$ estaría representado por la cadena 0110. Si tenemos 100 variables tendríamos una cadena con 100 dígitos binarios.

Una cadena cualquiera de n dígitos binarios es una candidata a ser una solución y, por lo tanto, tendremos que ir evaluándolas con una función de evaluación. Para 100 variables, nuestro espacio de estado (número de posibles soluciones) es de 2^{100}, es decir, un número astronómicamente alto. De nuevo nos encontramos con un problema intratable de la clase NP.

Además, encontrar una función de evaluación que nos permita comparar diferentes soluciones tampoco es fácil. Si evaluamos la función, nos devolverá un valor verdadero o falso, pero no tenemos información de cuántas variables son correctas y cuántas no (si lo supiéramos entonces significa que conocemos la solución). Si comparamos dos cadenas diferentes que devuelven el valor *false*, ¿cómo sabemos cuál de las dos es mejor?

© RC Libros

Al ser una función en FNC, sabemos que la evaluación de todas las cláusulas ha de arrojar un valor verdadero. Si hay uno que no lo es, la función no puede ser verdadera. Por lo tanto, una función de evaluación que nos daría una medida de la calidad de una asignación es aquella que devuelve el número de cláusulas que son verdaderas.

El problema de la programación lineal entera

El problema de la programación lineal entera (**PLE**) consiste en la optimización de una función en la que intervienen variables enteras y, además, hay unas restricciones impuestas. Por ejemplo, sea la función:

$$f(x_1, x_2) = 21x_1 + 11x_2$$

Donde x_1 y x_2 son dos variables enteras, y sean también las siguientes restricciones:

$$7x_1 + 4x_2 \leq 13$$
$$x_1 \geq 0$$
$$x_2 \geq 0$$

Es decir, buscamos una asignación de valores enteros para las dos variables que maximice (o minimice) el valor de la función f y que a su vez cumplan las restricciones impuestas de ser mayores o iguales que cero y que la suma de $7x_1$ y $4x_2$ sea menor que 13.

La programación lineal entera es un caso particular de la programación lineal (PL) en la que se exige que todas o algunas variables sean enteras. En el caso de la PL las variables son reales. Pese a lo que pueda parecer a primera vista, la PLE es un problema más complejo de resolver que la PL.

Para hacernos una idea de la utilidad que tiene poder resolver este tipo de problemas, vamos a plantear un ejemplo algo más elaborado.

Una empresa fabrica 2 tipos de prendas de vestir: camisetas y pantalones. El tiempo necesario para fabricar un pantalón es de 3 horas y necesita 4 metros de tela. En el caso de la camiseta se tardan 2 horas y son necesarios 3 metros de tela. También sabemos que la tela disponible para una semana son 160 metros y con los empleados que hay contratados se puede invertir 150 horas/semana de trabajo. El dueño de la empresa nos ha pedido que a partir del coste de producción y del precio

de venta de las prendas estimemos cuántas camisetas y cuántos pantalones tiene que fabricar para obtener el máximo de ganancias.

En la siguiente tabla hay un resumen de los datos.

	Pantalón	Camiseta
Horas producción por prenda	7	4
Metros por prenda	6	5
Coste de producción	6	4
Precio de venta	12	8

Comenzaremos buscando un modelo que represente el problema. Para ello, vamos a utilizar dos variables. Llamaremos x_1 al número de pantalones y x_2 al número de camisetas. Lo que buscamos es el máximo beneficio con los recursos de los que disponemos, así que definiremos primero el beneficio según la siguiente fórmula.

$$beneficio = (precio\ venta - \ costo\ producción) \times unidades\ vendidas$$

Como el beneficio total será la suma de los beneficios de ambas prendas, llegamos a la siguiente ecuación que representa la función de beneficio para nuestro caso concreto.

$$f(x_1, x_2) = (12 - 6)x_1 + (8 - 4)x_2$$

Por otro lado, vamos a imponer una serie de restricciones. La primera y más obvia es que ambas variables han de ser mayores o iguales a 0. No podemos fabricar un número negativo de prendas.

$$x_1 \geq 0$$
$$x_2 \geq 0$$

También sabemos que disponemos de una capacidad de trabajo semanal de 150 horas. Como fabricar un pantalón se invierten 3 horas y 2 en una camiseta, hemos de imponer la siguiente restricción al problema.

$$7x_1 + 4x_2 \leq 150$$

© RC Libros

Lo mismo sucede con la cantidad de metros de tela disponibles semanalmente, que son 160 metros. Como un pantalón necesita 4 metros de tela y una camiseta 3, impondremos la restricción

$$6x_1 + 5x_2 \leq 160$$

En resumen, nuestro modelo para el problema de PLE es:

$$f(x_1, x_2) = (12 - 6)x_1 + (8 - 4)x_2$$

Restricciones:

$$7x_1 + 4x_2 \leq 150$$

$$6x_1 + 5x_2 \leq 160$$

$$x_1 \geq 0$$
$$x_2 \geq 0$$

Nuestro objetivo pues es encontrar una asignación de valores enteros para las variables x_1 y x_2 que, cumpliendo con las restricciones impuestas, hagan que la función f tome el máximo valor posible.

Se puede comprobar fácilmente que la variable x_1 está acotada entre 0 y 50, mientras que X_2 lo está entre 0 y 75. Por lo tanto, nuestro espacio de estados constará de $50 \times 75 = 3.750$ posibilidades, que no es un número muy elevado. Es perfectamente tratable por un ordenador. Sin embargo, en este problema solo entran en juego dos variables y su rango de valores posibles es muy limitado. Puede demostrarse que para un mayor número de variables y rango de valores el problema es intratable.

La función de evaluación para nuestro problema es muy evidente. Podemos usar la misma función que queremos maximizar para comparar las diferentes asignaciones de variables. A mayor valor de la función de evaluación, mejor será la calidad de los valores de las variables.

Un problema relacionado con la PLE es el de la programación no lineal (**PNL**) en el que la función o alguna de las restricciones es una función no lineal y las variables toman valores reales.

Otros problemas

Los problemas propuestos hasta ahora no han sido escogidos al azar. Son problemas bien conocidos y estudiados, ya que son problemas canónicos; es decir, cada uno de estos problemas representan y se resuelven de la misma manera que una multitud de otros problemas diferentes. En definitiva, son un modelo que nos permiten atacar la resolución de otros similares. Por ejemplo, el TSP es la base de un problema con mucha mayor aplicación práctica: el **problema de las rutas de vehículos o vehicle routing problem (VRP)**. Las distribuidoras de mercancías o las empresas de reparto tratan siempre de optimizar las rutas de reparto, ya que ello trae consigo un gran beneficio económico por el ahorro en recursos: tiempo, combustible, vehículos y personal. El problema consiste en, dado un número arbitrario de clientes a los que hay que servir mercancía, calcular las rutas que minimicen el número de vehículos necesarios y el número de kilómetros recorridos por cada vehículo. Al resolver el problema de la ruta de vehículos hay que tener presente que las rutas pueden tener asociadas restricciones como:

- Carga máxima de los vehículos: dependiendo de la cantidad, peso o volumen de los pedidos de cada cliente, un vehículo podrá atender a más o menos clientes en un viaje.
- Horario de los clientes: las rutas tienen que diseñarse de acuerdo con los horarios de los clientes. De nada sirve que la ruta sea óptima si el transportista, que suele cobrar por horas, tiene que esperar una hora a que abra un comercio para poder servir la mercancía.
- Horario del transportista: los transportistas pueden tener limitaciones legales en cuanto a la distancia recorrida o tiempo de trabajo que pueden realizar sin descansar.
- Preferencia: algunos clientes pueden ser más importantes que otros y por lo tanto es posible que nos interese servirles antes que a los demás.
- Número de almacenes: no es lo mismo tener un solo almacén que varios. Si hay varios almacenes hay que tener en cuenta qué almacenes servirán a cada cliente en concreto siguiendo criterios de cercanía o de existencias de stock en cada almacén.
- Recarga: otro condicionante es si los vehículos cargan la mercancía una vez o pueden volver al almacén a recargar mercancía para seguir haciendo entregas. En este caso hay que decidir qué clientes se sirven en la primera carga y cuáles en la segunda y siguientes.

Evidentemente, si se puede prescindir de un vehículo, la empresa se ahorra un transportista y a menor número de kilómetros, mayor ahorro en combustible.

© RC Libros

A lo largo del libro presentaremos técnicas que nos permitirán lidiar con la complejidad implícita de estos problemas y a resolverlos, o al menos, conseguir aproximaciones lo suficientemente buenas como para ser aceptables.

Pero la IA trata de resolver muchos más problemas de muy distinto tipo. Estos son algunos de los campos donde se aplican técnicas de IA con un resultado prometedor.

- Minería de datos.
- Reconocimiento del habla.
- Reconocimiento de la escritura.
- Visión artificial.
- Diagnóstico médico.
- Robótica.
- Entornos industriales.
- Planificación y control.
- Videojuegos.
- Seguridad.
- Predicciones financieras.
- Predicción de catástrofes.
- Marketing especializado.

© RC Libros

3 Búsqueda no informada

BÚSQUEDA

Las técnicas que vamos a presentar a continuación son conocidas de forma genérica como **búsqueda**, ya que pretenden encontrar una solución válida dentro del espacio de estados. En concreto, las que analizaremos en este capítulo se denominan técnicas de **búsqueda no informada** debido a que el problema que queremos resolver no nos ofrece ninguna información adicional que nos ayude a encontrar una solución de forma más rápida, más allá de la que proporciona el propio enunciado.

Retomemos el problema de encontrar la ruta más corta por carretera entre dos ciudades. El problema nos proporciona información sobre el progreso de la búsqueda; es decir, su función de evaluación nos permite conocer la calidad de un estado en comparación con otros. Esto nos permite cierto margen de maniobra en toma de decisiones durante la búsqueda para poder mejorar el proceso. Por ejemplo, supongamos que durante el proceso de búsqueda tenemos a medio construir una ruta que en el momento actual tiene 1.120 kilómetros (todavía no es una solución completa porque la ruta no llega al destino). Sin embargo, en una de las rutas analizadas anteriormente habíamos encontrado una ruta válida con 1.112 kilómetros. En este caso, podemos abortar la búsqueda actual, ya que sabemos que será peor que otra que ya habíamos evaluado anteriormente. Cuando tenemos información que nos ayuda a guiar la búsqueda usaremos técnicas de **búsqueda informada**, que serán tratadas en el capítulo 4.

Como ejemplo para la búsqueda no informada imaginaremos que hemos perdido las llaves del coche. Sabemos que están en una habitación de la casa, pero no sabemos nada más. En este caso no nos queda más remedio que realizar la búsqueda habitación por habitación hasta dar con ellas. Ningún dato nos permite guiar la búsqueda. En este caso usaremos técnicas de búsqueda no informada, también conocidas como búsqueda a ciegas.

Partiremos de un ejemplo sencillo que nos ayudará a presentar algunos conceptos importantes. Un puzle lineal es un puzle para niños con grandes piezas. Se llama puzle lineal porque sus piezas solo pueden unirse en una línea. La figura 3-1 representa un puzle lineal. Se han numerado las piezas según su orden.



Fig. 3-1 Puzle lineal

Realmente podríamos sacar ventaja de algunas cuestiones, como por ejemplo, la forma de las piezas para probar alguna de las técnicas de búsqueda informada que veremos en el próximo capítulo. Sin embargo, en el ejemplo que estamos presentando vamos a suponer que todas las piezas son exactamente cuadradas y encajan unas con otra sea cual sea el orden en el que las distribuyamos.

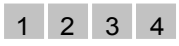
Antes de abordar la solución, vamos a hacernos algunas preguntas que nos ayuden a definir el problema.

¿Cuál es el espacio de estados del problema?

¿Cuál es el estado inicial y el final (objetivo)?

¿Qué operación nos permite pasar de un estado a otro?

El estado inicial puede ser cualquier ordenación posible de las cuatro piezas, mientras que el estado final u objetivo es el siguiente:



En cuanto al espacio de estados, un estado puede ser cualquier ordenación posible de las cuatro piezas, así que tendremos $4! = 24$ posibles estados. Con solo

© RC Libros

cuatro piezas vemos que el problema puede ser resuelto fácilmente examinando todos sus posibles estados, aunque como advertimos en el anterior capítulo, un número mayor de piezas lo volvería intratable.

Necesitamos definir una **operación** que nos permita ir explorando los posibles estados. Una operación básica que nos permite generarlos es intercambiar dos piezas continuas. Para el caso del puzle de 4 piezas, tendríamos 3 operaciones posibles:

- Intercambiar las dos piezas de la derecha: la llamaremos operación **D**.
- Intercambiar las dos piezas centrales: la llamaremos operación **C**.
- Intercambiar las dos piezas de la izquierda: la llamaremos operación **I**.

Por último, necesitamos una función de evaluación. La más lógica sería una función que evaluara cuántas piezas hay bien colocadas, pero como queremos mostrar las técnicas de búsqueda no informada, vamos a obviar ese dato y nuestra función de evaluación solo comprobará si hemos llegado al estado objetivo o no. En el siguiente capítulo retomaremos este mismo problema para resolverlo usando técnicas de búsqueda informada.

Representación de estados: árboles y grafos

Ahora que sabemos algo más sobre el problema, necesitamos plantearnos otras cuestiones referentes a la implementación para una solución en el ordenador. La primera y más básica es: ¿cómo representamos el puzle y sus estados en un programa? Parece una buena idea representar las piezas del puzle como un array de números enteros con dimensión 4. Usando esta representación algunos estados posibles podrían ser:

[2,1,4,3]

[1,2,4,3]

[4,2,3,4]

Cualquiera de ellos es un estado válido. Si aplicamos a cada estado uno de los operadores que hemos definido (I, C, D), los arrays se transformarán de la siguiente manera:

© RC Libros

[2,1,4,3] ⟶ [1,2,4,3] (Operador I)

[1,2,4,3] ⟶ [1,4,2,3] (Operador C)

[4,2,3,4] ⟶ [4,2,4,3] (Operador D)

La mejor forma de representar los estados y el resultado de aplicar un operador es usar un árbol. En la figura 3-2 podemos ver un árbol que muestra cómo evolucionan los estados al ir aplicando los operadores que hemos definido.

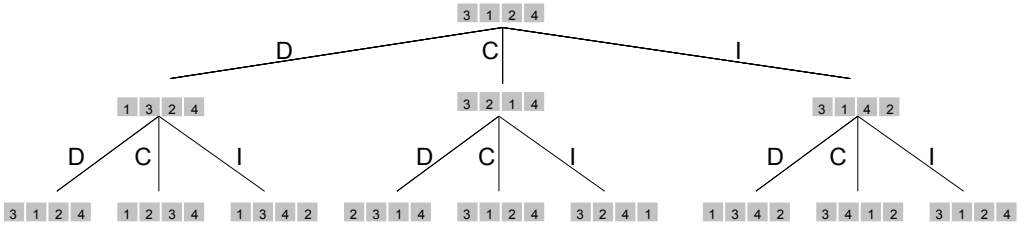


Fig. 3-2 Árbol de estados

Dentro de la nomenclatura usada en teoría de árboles y grafos, llamaremos **nodo** a cada uno de los estados. A los nodos resultantes de aplicar los operadores los llamamos nodos **hijos** y al nodo que ha generado a los hijos los llamamos nodo **padre**. Todos los nodos hijos de un mismo padre se denominan **hermanos**. El nodo superior, que es el único que no tiene padre, lo llamamos nodo **raíz**, en este caso se corresponde con el estado inicial [3,1,2,4]. A un nodo que no tiene hijos lo llamamos **hoja**. Un nodo que tenga padre y, al menos un hijo, lo llamaremos **rama**.

Como tenemos tres operadores, al aplicarlo a un nodo obtendremos tres hijos. En este caso decimos que el árbol tiene un **factor de ramificación** de 3. Además, cada vez que aplicamos un operador decimos que hemos descendido un **nivel** en la jerarquía del árbol.

Si observamos el desarrollo del árbol, puede comprobarse que hay estados que son iguales. Por ejemplo, si al estado inicial aplicamos el operador D y, seguidamente volvemos a aplicarlo, obtendremos de nuevo el estado inicial tal y como se observa en la figura 3-3. En este caso podemos modelar los estados como un grafo, que es una estructura similar al árbol pero que tiene ciclos. Los grafos suelen presentarse uniendo sus nodos con líneas o flechas curvas. Si podemos pasar de un estado a otro

© RC Libros

en ambos sentidos usaremos una línea; sin embargo, si la transición de un estado a otro solo se puede dar en un sentido, usaremos una flecha para indicarlo. En la figura 2-4 del anterior capítulo ya presentamos un grafo que mostraba cómo estaban unidas las ciudades.

Cuando en el desarrollo del árbol se encuentran ciclos, como nos ha pasado en este caso, la búsqueda de la solución podría no encontrarse nunca, ya que el programa podría quedar indefinidamente en esos ciclos, por lo que tendremos que evitarlos. Una forma habitual de hacerlo es "recordar" qué nodos hemos visitado almacenándolos en memoria y cada vez que lleguemos a un nuevo nodo, comprobar que no lo hemos visitado ya con anterioridad.

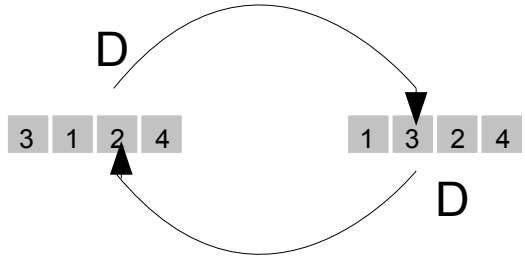


Fig. 3-3 Grafo

Encontrar una solución a un problema consistirá en hacer una búsqueda en el árbol de estados o en un grafo para encontrar un nodo que contenga un estado objetivo. En general, el recorrido de un árbol (o grafo) se compone de los siguientes pasos:

1. Seleccionamos el nodo raíz y lo almacenamos en una lista que contendrá aquellos nodos que están pendientes de visitar. A esta lista la llamaremos **frontera** (también se conoce como **lista abierta**).
2. Cogemos un nodo de la lista de nodos frontera y comprobamos si es un nodo objetivo. Si lo es, hemos terminado.
3. Generamos todos los hijos del nodo seleccionado en el paso 2 aplicando los operadores que hemos definido. Esto es lo que se llama **expandir un nodo**. Añadimos los nuevos nodos hijo a la lista de nodos frontera.
4. Volvemos al paso 2 hasta que la lista de nodos frontera esté vacía.

Si en lugar de un árbol estamos trabajando con un grafo, el proceso cambia ligeramente:

© RC Libros

1. Seleccionamos el nodo raíz y lo almacenamos en la lista de nodos frontera.
2. Cogemos un nodo de la lista de nodos frontera y comprobamos si es un nodo objetivo. Si lo es, hemos terminado. Además, almacenamos el nodo en una lista llamada **visitados**, que contiene todos los nodos visitados hasta el momento.
3. Generamos todos los hijos del nodo seleccionado en el paso 2 aplicando los operadores que hemos definido. Para cada hijo comprobamos que no está en la lista de nodos visitados, y si no está, la añadimos a la lista de nodos frontera.
4. Volvemos al paso 2 hasta que la lista de nodos frontera esté vacía.

BÚSQUEDA EN AMPLITUD

El proceso de búsqueda que acabamos de presentar es un procedimiento genérico que puede dar lugar a variaciones. La primera que vamos a presentar se llama **búsqueda en amplitud** o ***Breadth First Search*** (BFS). Esta búsqueda recorre el árbol por niveles. Es decir:

- Primero se visita el nodo raíz.
- Seguidamente se visitan todos sus hijos.
- Para cada hijo en el paso anterior se visitan todos sus hijos, y así sucesivamente.

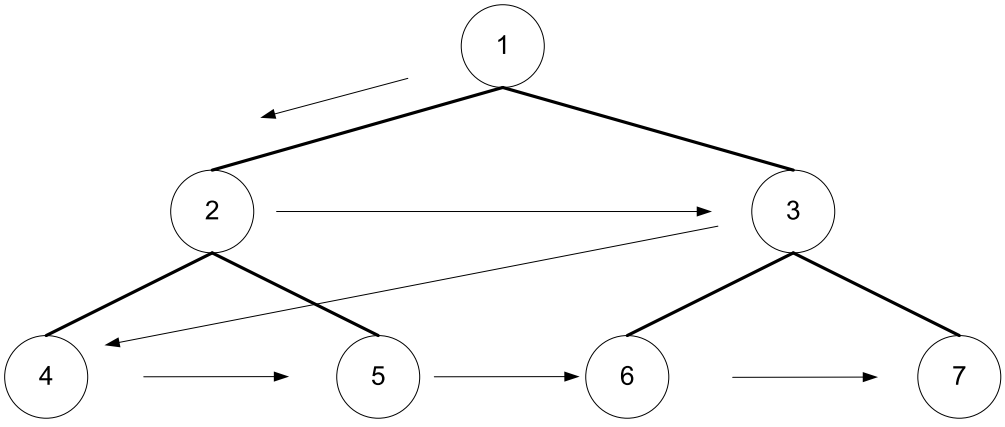


Fig. 3-4 Búsqueda en amplitud

© RC Libros

Este recorrido se implementa en la práctica usando una cola FIFO (First In First Out) para la lista de nodos frontera. Una cola FIFO es una estructura de datos en la que podemos hacer dos operaciones principales: almacenar un dato y sacar un dato. La particularidad de las colas FIFO es que cuando sacamos un dato, este se extrae en el mismo orden en el que se almacenó. A continuación, se muestra el algoritmo de búsqueda en amplitud en pseudocódigo.

```
nodo_inicial = estado inicial
nodos_frontera = Cola FIFO
nodos_visitados = Lista
almacenar nodo_inicial en nodos_frontera
mientras nodos_frontera no vacío:
    nodo_actual = extraer un nodo de nodos_frontera
    si nodo_actual == solución:
           salir con solución

    introducir nodo_actual en nodos_visitados
    por cada operador:
           nodo_hijo = operador(nodo_actual)
           si nodo_hijo no en nodos_visitados ni nodos_frontera:
                  introducir nodo_hijo en nodos_frontera
```

Listado. 3-1 Pseudocódigo búsqueda en amplitud

Vamos a llevar a la práctica el algoritmo haciendo un programa para resolver el problema del puzle lineal. A lo largo de este capítulo, en los programas de ejemplo, vamos a hacer uso de la clase Nodo del listado 3-2. Es una sencilla implementación de la estructura de datos árbol de la que hemos hablado en la sección anterior. El archivo que contiene a la clase se llama *arbol.py*.

```
class Nodo:
  def __init__(self, datos, hijos=None):
    self.datos = datos
    self.hijos = None
    self.padre = None
```

© RC Libros

```
      self.coste= None
      self.set_hijos(hijos)

    def set_hijos(self, hijos):
      self.hijos=hijos
      if self.hijos != None:
        for h in self.hijos:
          h.padre = self

    def get_hijos(self):
      return self.hijos

    def get_padre(self):
      return self.padre

    def set_padre(self, padre):
      self.padre = padre

    def set_datos(self, datos):
      self.datos = datos

    def get_datos(self):
      return self.datos

    def set_coste(self, coste):
      self.coste = coste

    def get_coste(self):
      return self.coste

    def igual(self, nodo):
      if self.get_datos() == nodo.get_datos():
        return True
      else:
```


© RC Libros

```
        return False

    def en_lista(self, lista_nodos):
      en_la_lista=False
      for n in lista_nodos:
        if self.igual(n):
          en_la_lista=True
      return en_la_lista

    def __str__(self):
      return str(self.get_datos())
```

Listado. 3-2 arbol.py

El uso de la clase Nodo es sencillo. El constructor acepta como primer parámetro un dato que será almacenado en el nodo (en principio de cualquier tipo). Opcionalmente acepta un segundo parámetro con una lista de hijos.

También es posible usar los siguientes métodos:

set_hijos(hijos) – Asigna al nodo la lista de hijos que son pasados como parámetro.

get_hijos() – Retorna una lista con los hijos del nodo.

get_padre() – Retorna el nodo padre.

set_padre(padre) – Asigna el nodo padre de este nodo.

set_datos(dato) – Asigna un dato al nodo.

get_datos() – Devuelve el dato almacenado en el nodo.

set_peso() – Asigna un peso al nodo dentro del árbol.

get_peso() – Devuelve el peso del nodo dentro del árbol.

Igual(nodo) – Devuelve True si el dato contenido en el nodo es igual al nodo pasado como parámetro.

© RC Libros

En_lista(lista_nodos) – Devuelve True si el dato contenido en el nodo es igual a alguno de los nodos contenidos en la lista de nodos pasada como parámetro.

El código del programa que resuelve el problema del puzle lineal es el siguiente:

```
# Puzle Lineal con busqueda en amplitud
from arbol import Nodo

def buscar_solucion_BFS(estado_inicial, solucion):
  solucionado=False
  nodos_visitados=[]
  nodos_frontera=[]
  nodoInicial = Nodo(estado_inicial)
  nodos_frontera.append(nodoInicial)
  while (not solucionado) and len(nodos_frontera)!=0:
    nodo=nodos_frontera.pop(0)
    # extraer nodo y añadirlo a visitados
    nodos_visitados.append(nodo)
    if nodo.get_datos() == solucion:
      # solucion encontrada
      solucionado=True
      return nodo
    else:
      # expandir nodos hijo
      dato_nodo = nodo.get_datos()

      # operador izquierdo
      hijo=[dato_nodo[1], dato_nodo[0], dato_nodo[2], dato_nodo[3]]
      hijo_izquierdo = Nodo(hijo)
      if not hijo_izquierdo.en_lista(nodos_visitados) \
      and not hijo_izquierdo.en_lista(nodos_frontera):
        nodos_frontera.append(hijo_izquierdo)
      # operador central
      hijo=[dato_nodo[0], dato_nodo[2], dato_nodo[1], dato_nodo[3]]
```


© RC Libros

```
        hijo_central = Nodo(hijo)
        if not hijo_central.en_lista(nodos_visitados) \
        and not hijo_central.en_lista(nodos_frontera):
          nodos_frontera.append(hijo_central)
        # operador derecho
        hijo=[dato_nodo[0], dato_nodo[1], dato_nodo[3], dato_nodo[2]]
        hijo_derecho = Nodo(hijo)
        if not hijo_derecho.en_lista(nodos_visitados) \
        and not hijo_derecho.en_lista(nodos_frontera):
          nodos_frontera.append(hijo_derecho)

        nodo.set_hijos([hijo_izquierdo, hijo_central, hijo_derecho])

  if __name__ == "__main__":
    estado_inicial=[4,2,3,1]
    solucion=[1,2,3,4]
    nodo_solucion = buscar_solucion_BFS(estado_inicial, solucion)
    # mostrar resultado
    resultado=[]
    nodo=nodo_solucion
    while nodo.get_padre() != None:
      resultado.append(nodo.get_datos())
      nodo = nodo.get_padre()
    resultado.append(estado_inicial)
    resultado.reverse()
    print resultado
```

Listado. 3-3 puzlelineal_bfs.py

La ejecución del programa anterior nos mostrará el siguiente resultado:

[[4, 2, 3, 1], [2, 4, 3, 1], [2, 3, 4, 1], [2, 3, 1, 4], [2, 1, 3, 4], [1, 2, 3, 4]]

Que es la secuencia de movimientos que realiza el programa para resolver el puzle.

© RC Libros

La función encargada de implementar la búsqueda en amplitud se llama *buscar_solucion_BFS()* que es invocada con dos parámetros. El primero es el estado inicial, que en nuestro ejemplo es [4,2,3,1]. El segundo parámetro es el estado objetivo, es decir, la solución: [1,2,3,4]. La función retorna con el nodo solución si lo ha encontrado. Para conocer el camino concreto desde el nodo raíz al nodo objetivo, solo hay que recorrer el camino inverso desde el nodo objetivo. Como cada nodo tiene asignado un padre, solo tenemos que usar la función *get_padre()* para recorrer el camino.

Dentro de la función *buscar_solucion_BFS()* lo primero que se hace es inicializar las listas de nodos frontera y nodos visitados. También se crea el nodo inicial con el valor del primer parámetro de la función y se añade a la lista de nodos frontera.

La siguiente porción del código es un bucle y se repite mientras haya elementos en la lista de nodos frontera o no se haya alcanzado una solución. El primer paso es extraer el primer nodo pendiente de la cola de nodos frontera e insertarlo en la lista de elementos visitados. Si el nodo es una solución, salimos del bucle devolviendo el nodo. Si no, expandimos el nodo aplicando los operadores I, C y D, y generamos sus hijos. Si los hijos no están en la lista de nodos visitados, los añadimos a la cola de nodos frontera.

La búsqueda en amplitud tiene algunas propiedades que vale la pena señalar. Cabe preguntarse si la búsqueda en amplitud encontrará siempre una solución. En efecto, es fácil comprobar que si existe una solución, la búsqueda en amplitud acabará encontrándola (otra cuestión diferente es en cuánto tiempo). Cuando de un algoritmo sabemos que siempre encontrará una solución si se le deja el tiempo necesario, decimos que el algoritmo es **completo** y nos referiremos a esta característica como **completitud** de un algoritmo. Como veremos no todos los algoritmos que trataremos en este libro son completos.

Sabiendo que siempre encontraremos una solución, otra cuestión que podemos plantearnos es si la solución que encontremos será la mejor posible. Si lo es, decimos que el algoritmo es **óptimo** y nos referiremos a esta característica como **optimalidad** (que es una mala traducción de la palabra inglesa *optimality*). Pero ¿es óptima la búsqueda en amplitud?

Primero analicemos el significado de la palabra óptimo, que depende del problema que estemos tratando y del objetivo que estemos buscando. Recordemos del capítulo anterior el ejemplo del viaje en avión. Nuestro objetivo era conseguir el mínimo número de trasbordos posibles. Por lo tanto, diremos que la solución es óptima si no es posible encontrar otra solución diferente con un número de trasbordos menor.

© RC Libros

Sin embargo, en el ejemplo del viaje por carretera, nuestro objetivo era otro: reducir el número de kilómetros. Por lo tanto, una solución será óptima si no es posible encontrar una ruta con menor número de kilómetros.

En el caso del puzle lineal, podríamos plantearnos que una solución será óptima si no es posible encontrar otra diferente que resuelva el puzle en menor número de movimientos. Si nos fijamos en el árbol de la figura 3-2, observamos que por cada movimiento bajamos un nivel en el árbol. Así pues, a cuanta menos profundidad se encuentre la solución, menos movimientos habrán sido necesarios para llegar a ella. Como la búsqueda en amplitud recorre el árbol por niveles, queda claro que, en este caso, la solución que encontremos será óptima.

Con el problema de los trasbordos del viaje en avión ocurre algo parecido. Cada trasbordo se traduce en un descenso en el nivel del árbol, por lo que la solución que encontremos con la búsqueda en amplitud también será óptima.

¿Y con el problema del viaje por carretera? ¿Ocurre igual? Imaginemos una red de carreteras ficticia como la siguiente.

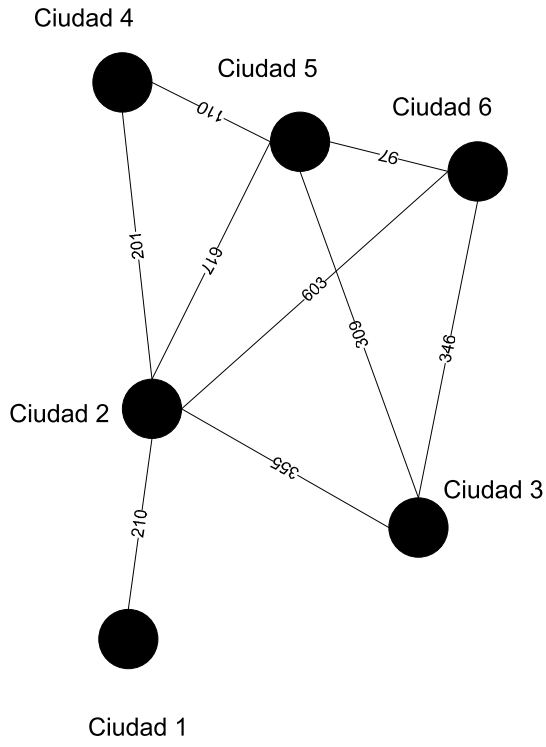


Fig. 3-5 Red de carreteras

Nuestro objetivo es llegar desde la ciudad 1 a la ciudad 5 cubriendo la menor distancia posible. De nuevo, usando la búsqueda en amplitud, cada paso por una ciudad supone bajar un nivel en el árbol, por lo que parece que tiene sentido que a menor número de ciudades recorridas, mejor solución. Sin embargo, observemos estas dos soluciones:

Ciudad 1 > Ciudad 5 (827 kilómetros)

Ciudad 1 > Ciudad 4 > Ciudad 5 (521 kilómetros)

Usando una búsqueda en amplitud hubiéramos llegado a la solución primera que atraviesa dos ciudades con 827 kilómetros (dos niveles del árbol). Sin embargo, si hubiéramos permitido que el algoritmo descendiera un nivel más, hubiera encontrado una solución mejor con 521 kilómetros, aunque atraviese una ciudad más. Por lo tanto, en el caso de la ruta por carretera, el algoritmo de búsqueda en amplitud no es óptimo.

En general, podemos decir que es óptimo si el coste de la función de evaluación es un valor no decreciente directamente relacionado con la profundidad dentro del recorrido del árbol.

A la hora de decidirnos por un algoritmo u otro, es conveniente conocer su comportamiento en cuanto al tiempo de ejecución y cantidad de memoria necesaria. El tiempo de ejecución necesario depende directamente del número de nodos generados. Es lo que llamamos **complejidad temporal**. Si el árbol tiene un factor de ramificación b, del nodo raíz saldrán b hijos. En el siguiente nivel tendremos b^2, y así sucesivamente. En general, para una profundidad d tendremos:

$$número\ de\ nodos = b + b^2 + b^3 + \cdots + b^d$$

Generalmente, se utiliza una notación especial para expresar la complejidad de un algoritmo: la cota superior asintótica. Más comúnmente conocida como notación O (notación O grande). Sin entrar en detalle diremos que se corresponde con una función que sirve de cota superior de la función que estamos analizando cuando el argumento tiende a infinito. Para este caso diremos que el algoritmo de búsqueda en amplitud tiene una complejidad temporal $O(b^d)$.

En cuanto al uso de memoria, que llamaremos **complejidad espacial**, la búsqueda en amplitud mantiene en memoria cada uno de los nodos expandidos. Además, al usar dos estructuras de datos tendremos una complejidad $O(b^d)$ para almacenar los nodos frontera y $O(b^{d-1})$ para los nodos visitados.

© RC Libros

La notación O es muy útil para comparar la complejidad de dos algoritmos diferentes.

Antes de seguir avanzando resolveremos el problema de los vuelos del capítulo anterior usando la búsqueda en amplitud.

```
# Vuelos con búsqueda en amplitud
from arbol import Nodo

def buscar_solucion_BFS(conexiones, estado_inicial, solucion):
  solucionado=False
  nodos_visitados=[]
  nodos_frontera=[]
  nodoInicial = Nodo(estado_inicial)
  nodos_frontera.append(nodoInicial)
  while (not solucionado) and len(nodos_frontera)!=0:
    nodo=nodos_frontera[0]
    # extraer nodo y añadirlo a visitados
    nodos_visitados.append(nodos_frontera.pop(0))
    if nodo.get_datos() == solucion:
      # solución encontrada
      solucionado=True
      return nodo
    else:
      # expandir nodos hijo (ciudades con conexión)
      dato_nodo = nodo.get_datos()
      lista_hijos=[]
      for un_hijo in conexiones[dato_nodo]:
        hijo=Nodo(un_hijo)
        lista_hijos.append(hijo)
        if not hijo.en_lista(nodos_visitados) \
        and not hijo.en_lista(nodos_frontera):
          nodos_frontera.append(hijo)

      nodo.set_hijos(lista_hijos)
```

```
  if __name__ == "__main__":
      conexiones = {
      'Malaga':{'Salamanca', 'Madrid', 'Barcelona'},
      'Sevilla':{'Santiago', 'Madrid'},
      'Granada':{'Valencia'},
      'Valencia':{'Barcelona'},
      'Madrid':{'Salamanca', 'Sevilla', 'Malaga', \
      'Barcelona','Santander'},
      'Salamanca':{'Malaga', 'Madrid'},
      'Santiago':{'Sevilla', 'Santander', 'Barcelona'},
      'Santander':{'Santiago', 'Madrid'},
      'Zaragoza':{'Barcelona'},
      'Barcelona':{'Zaragoza', 'Santiago', 'Madrid', 'Malaga', \
      'Valencia'}
      }
    estado_inicial='Malaga'
    solucion='Santiago'
    nodo_solucion = buscar_solucion_BFS(conexiones, estado_inicial,
solucion)
    # mostrar resultado
    resultado=[]
    nodo=nodo_solucion
    while nodo.get_padre() != None:
      resultado.append(nodo.get_datos())
      nodo = nodo.get_padre()
    resultado.append(estado_inicial)
    resultado.reverse()
    print resultado
```

Listado. 3-4 vuelos_bfs.py

© RC Libros

Al programa le hemos pedido que nos encuentre la mejor secuencia de trasbordos para ir de Málaga a Santiago, y esta es la propuesta que nos ha hecho:

['Malaga', 'Barcelona', 'Santiago']

Vemos que es una solución óptima (según el objetivo que hemos definido), ya que aunque con tres conexiones hay otras soluciones posibles, no podemos encontrar ninguna con menos de tres.

Para representar las conexiones, hemos usado la estructura de datos diccionario de python, que nos permite de manera sencilla representar todas las conexiones. A la hora de generar los nodos hijo solo tenemos que obtener los elementos almacenados en el diccionario que corresponde con la ciudad que estamos examinando.

BÚSQUEDA EN PROFUNDIDAD

La **búsqueda en profundidad** o **Depth First Search** (DFS) recorre el árbol/grafo de forma diferente a la búsqueda en amplitud. En lugar de ir visitando todos los nodos de un mismo nivel, va descendiendo hasta la profundidad máxima de la rama y cuando llega al nodo más profundo continúa con la siguiente.

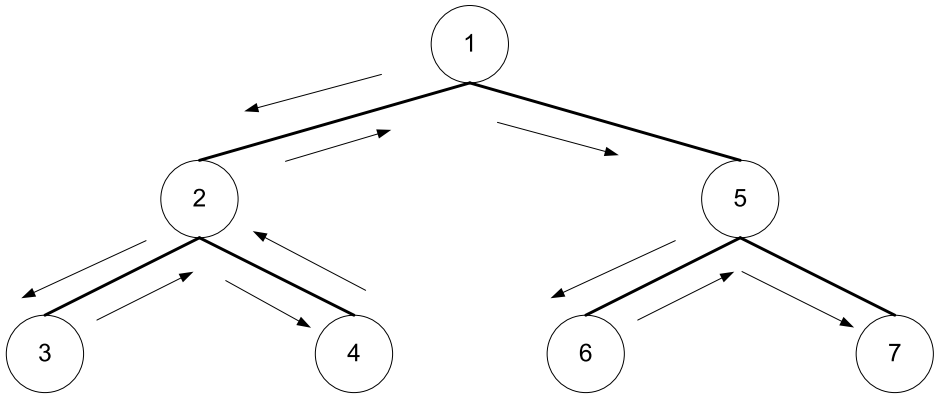


Fig. 3-6 Búsqueda en profundidad

Desde el punto de vista de la implementación, la búsqueda en profundidad se diferencia de la búsqueda en amplitud en que utiliza una pila LIFO (Last In Firts Out) en vez de una cola FIFO. Una pila LIFO tiene dos operaciones: una para introducir un dato en la pila y otra para extraer un dato de la pila. A diferencia de la cola, el dato que se extrae de la pila es el último que se introdujo. Se puede imaginar como una pila de platos. Cada vez que amontonamos uno quedará en la parte superior del montón. Para coger uno siempre tendremos que coger el que haya en la parte superior del montón, es decir, el último que se agregó al montón.

El algoritmo en pseudocódigo de la búsqueda en profundidad es el siguiente:

```
nodo_inicial = estado inicial
nodos_frontera = Pila LIFO
nodos_visitados = Lista
almacenar nodo_inicial en nodos_frontera
mientras nodos_frontera no vacío:
    nodo_actual = extraer un nodo de nodos_frontera
    si nodo_actual == solución:
           salir con solución

    introducir nodo_actual en nodos_visitados
    por cada operador:
           nodo_hijo = operador(nodo_actual)
           si nodo_hijo no en nodos_visitados ni nodos_frontera:
                  introducir nodo_hijo en nodos_frontera
```

Listado. 3-5 Pseudocódigo búsqueda en profundidad

Como puede observarse, el algoritmo es muy similar al de la búsqueda en amplitud, con la salvedad de la estructura de datos usada para almacenar los nodos frontera. La implementación del algoritmo para resolver el problema del puzle lineal sería la siguiente:

```
# Puzle Lineal con búsqueda en profundidad
from arbol import Nodo
```


© RC Libros

```
def buscar_solucion_DFS(estado_inicial, solucion):
  solucionado=False
  nodos_visitados=[]
  nodos_frontera=[]
  nodoInicial = Nodo(estado_inicial)
  nodos_frontera.append(nodoInicial)
  while (not solucionado) and len(nodos_frontera)!=0:
    nodo=nodos_frontera.pop()
    # extraer nodo y añadirlo a visitados
    nodos_visitados.append(nodo)
    if nodo.get_datos() == solucion:
      # solución encontrada
      solucionado=True
      return nodo
    else:
      # expandir nodos hijo
      dato_nodo = nodo.get_datos()

      # operador izquierdo
      hijo=[dato_nodo[1], dato_nodo[0], dato_nodo[2], dato_nodo[3]]
      hijo_izquierdo = Nodo(hijo)
      if not hijo_izquierdo.en_lista(nodos_visitados) \
      and not hijo_izquierdo.en_lista(nodos_frontera):
        nodos_frontera.append(hijo_izquierdo)
      # operador central
      hijo=[dato_nodo[0], dato_nodo[2], dato_nodo[1], dato_nodo[3]]
      hijo_central = Nodo(hijo)
      if not hijo_central.en_lista(nodos_visitados) \
      and not hijo_central.en_lista(nodos_frontera):
        nodos_frontera.append(hijo_central)
      # operador derecho
      hijo=[dato_nodo[0], dato_nodo[1], dato_nodo[3], dato_nodo[2]]
      hijo_derecho = Nodo(hijo)
      if not hijo_derecho.en_lista(nodos_visitados) \
```

```
        and not hijo_derecho.en_lista(nodos_frontera):
          nodos_frontera.append(hijo_derecho)

        nodo.set_hijos([hijo_izquierdo, hijo_central, hijo_derecho])

if __name__ == "__main__":
  estado_inicial=[4,2,3,1]
  solucion=[1,2,3,4]
  nodo_solucion = buscar_solucion_DFS(estado_inicial, solucion)
  # mostrar resultado
  resultado=[]
  nodo=nodo_solucion
  while nodo.get_padre() != None:
    resultado.append(nodo.get_datos())
    nodo = nodo.get_padre()
  resultado.append(estado_inicial)
  resultado.reverse()
  print resultado
```

Listado. 3-6 puzlelineal_dfs.py

Tras la ejecución obtenemos como resultado:

[[4, 2, 3, 1], [4, 2, 1, 3], [4, 1, 2, 3], [4, 1, 3, 2], [4, 3, 1, 2], [3, 4, 1, 2], [3, 4, 2, 1], [3, 2, 4, 1], [3, 2, 1, 4], [3, 1, 2, 4], [1, 3, 2, 4], [1, 2, 3, 4]]

Que es un resultado visiblemente peor que el obtenido con la búsqueda en amplitud. Esto nos indica de entrada que esta búsqueda no es óptima como en el caso de la búsqueda en amplitud. Decimos que es **subóptima**. Otro problema que presenta el algoritmo es que si el árbol de búsqueda no está acotado en profundidad, es decir, es infinitamente profundo, nunca saldremos de la primera rama. En cuanto a la complejidad temporal, un árbol con profundidad p tiene una complejidad $O(b^p)$. Recordemos que la complejidad de la búsqueda en amplitud era $O(b^d)$, donde d es la profundidad a la que se encuentra la solución. Como en la mayoría de los casos p>d, podemos afirmar que la complejidad de la búsqueda en profundidad será, para la mayoría de los casos, mayor que la de la búsqueda en amplitud (o en el mejor de los casos igual).

© RC Libros

¿Por qué presentamos entonces la búsqueda en profundidad en este capítulo si parece no tener ninguna ventaja sobre la búsqueda en amplitud? Deliberadamente no hemos hablado aún de la complejidad espacial.

En los pseudocódigos de ambas búsquedas hemos usado una lista para almacenar los nodos visitados. Lo hacíamos así para no perder generalidad y que el algoritmo fuera usable tanto en árboles como en grafos. Ahora bien, si nuestro espacio de búsqueda está descrito por un árbol y no por un grafo, podemos prescindir de la lista de nodos visitados. En la búsqueda en profundidad, si el espacio de búsqueda está representado por un árbol, la complejidad espacial se reduce a O(bm), lo que supone un coste de almacenamiento muy inferior al usado en la búsqueda en amplitud. Esto es debido a que cuando exploramos una rama, sus nodos quedan eliminados de la lista de nodos frontera, mientras que en la búsqueda en amplitud era necesario mantener almacenados los nodos frontera de todas las ramas a la vez.

Para hacernos una idea tomemos como ejemplo el puzle lineal. Si suponemos que un nodo necesita 1 kilobyte de almacenamiento (incluyendo las estructuras propias de python) la exploración de un árbol (es decir, sin lista de nodos visitados) hasta el nivel 10 necesita unos 60 megabytes de memoria RAM para la búsqueda en amplitud, mientras que la misma búsqueda usando el algoritmo en profundidad necesita unos 90 kilobytes. La diferencia es más que apreciable.

Si la búsqueda la realizamos usando la lista de visitados, la búsqueda es también completa. En caso de prescindir de almacenar los nodos visitados en aras de preservar el uso de memoria, el algoritmo no es completo, ya que podría entrar en bucles durante la búsqueda.

Por las características propias de la búsqueda en profundidad, es posible una implementación alternativa usando recursividad. El siguiente pseudocódigo muestra la implementación de forma general.

```
Función DFS_rec(nodo, solución, visitados):
  Añadir nodo a visitados
  Si nodo == solución:
    Salir con solución
  Si no:
    Por cada operador:
      nodo_hijo = operador(nodo)
      Si nodo_hijo no en visitados:
```

© RC Libros

```
          s = DFS_rec(nodo_hijo, solución, visitados)
          Salir con s

  nodo = estado inicial
  visitados = Lista
  solucion = solución
  s = DFS_rec(nodo, solucion, visitados)
```

Listado. 3-7 Pseudocódigo búsqueda en profundidad recursiva

El esquema expresado en el pseudocódigo podemos llevarlo a la implementación concreta para resolver el puzle lineal.

```
# Puzle Lineal con búsqueda en profundidad recursiva
from arbol import Nodo

def buscar_solucion_DFS_Rec(nodo_inicial, solucion, visitados):
  visitados.append(nodo_inicial.get_datos())
  if nodo_inicial.get_datos() == solucion:
    return nodo_inicial
  else:
    # expandir nodos sucesores (hijos)
    dato_nodo = nodo_inicial.get_datos()
    hijo=[dato_nodo[1], dato_nodo[0], dato_nodo[2], dato_nodo[3]]
    hijo_izquierdo = Nodo(hijo)
    hijo=[dato_nodo[0], dato_nodo[2], dato_nodo[1], dato_nodo[3]]
    hijo_central = Nodo(hijo)
    hijo=[dato_nodo[0], dato_nodo[1], dato_nodo[3], dato_nodo[2]]
    hijo_derecho = Nodo(hijo)
    nodo_inicial.set_hijos([hijo_izquierdo, hijo_central, \
    hijo_derecho])

    for nodo_hijo in nodo_inicial.get_hijos():
```


© RC Libros

```
        if not nodo_hijo.get_datos() in visitados:
          # llamada recursiva
          sol = buscar_solucion_DFS_Rec(nodo_hijo, solucion, \
          visitados)
          if sol != None:
            return sol

      return None


if __name__ == "__main__":
  estado_inicial=[4,2,3,1]
  solucion=[1,2,3,4]
  nodo_solucion = None
  visitados=[]
  nodo_inicial = Nodo(estado_inicial)
  nodo = buscar_solucion_DFS_Rec(nodo_inicial, solucion, visitados)

  # mostrar resultado
  resultado=[]
  while nodo.get_padre() != None:
      resultado.append(nodo.get_datos())
      nodo = nodo.get_padre()
  resultado.append(estado_inicial)
  resultado.reverse()
  print resultado
```

Listado. 3-8 puzlelineal_bfs_rec.py

Visto que la búsqueda en profundidad es ventajosa en cuanto al uso de memoria, podemos intentar mejorar el resto de aspectos para intentar quedarnos con lo mejor de la búsqueda en profundidad y la búsqueda en amplitud.

Uno de los problemas de la búsqueda en profundidad se da cuando el espacio de estados es infinito. En este escenario no es posible que el algoritmo dé con una

solución (a no ser que tengamos la tremenda suerte de que esta se encuentre en la primera rama del árbol). Además, el algoritmo no terminaría nunca. Para evitar esto, se puede hacer una modificación al algoritmo para evitar que descienda más de un número prefijado de niveles. Así evitamos que quede en ejecución indefinidamente. Se trata de la **búsqueda en profundidad limitada**.

```
Función DFS_limitado(nodo, solución, visitados, límite):
  Añadir nodo a visitados
  Si nodo == solución:
    Salir con solución
  Si no si limite == 0:
    Salir con corte_límite
  Si no:
    Por cada operador:
      nodo_hijo = operador(nodo)
      Si nodo_hijo no en visitados:
        s = DFS_rec(nodo_hijo, solución, visitados, límite-1)
        Salir con s

nodo = estado inicial
visitados = Lista
solucion = solución
límite = límite_niveles
s = DFS_limitado(nodo, solución, visitados, límite)
```

Listado. 3-9 Pseudocódigo búsqueda en profundidad limitada

Partiendo de esta mejora podemos conseguir, además, que el algoritmo sea óptimo. La técnica se llama **búsqueda con profundidad iterativa**. La idea es ir repitiendo la búsqueda de forma iterativa pero incrementando el nivel de profundidad en cada iteración.

```
Función DFS_prof_iter(nodo, solución):
  para límite de 1 a ∞:
```


© RC Libros

```
      visitados = Lista
      s = DFS_limitado(nodo, solución, visitados, límite)
      si s ≠ corte_límite:
        salir con s

nodo = estado inicial
solucion = solución
s = DFS_prof_iter(nodo, solución)
```

Listado. 3-10 Pseudocódigo búsqueda con profundidad iterativa

Aunque parezca un proceso pesado y costoso, en realidad no lo es tanto. Hay que tener en cuenta que la operación más costosa es generar los nodos más profundos (hojas). En comparación, generar los niveles anteriores es un proceso liviano. Además, si encontramos la solución en el nivel d del árbol, los nodos de este nivel solo se habrán generado una vez, los del nivel d-1 dos veces y así sucesivamente. Es decir, el número de nodos total generados en un árbol con factor de ramificación b y profundidad d será:

$$número\ de\ nodos = b(d) + b^2(d-1) + b^3(d-2) + \cdots + b^d$$

Por lo que su complejidad es $O(b^d)$, que es exactamente la misma que la de la búsqueda en amplitud. Es por ello, que si no conocemos la profundidad a la que se encuentra la solución y el espacio de búsqueda es grande, este algoritmo será preferible a las otras dos búsquedas.

Como ejemplo presentamos una solución al problema de los vuelos resuelto con la búsqueda con profundidad iterativa.

```
# Vuelos con búsqueda con profundidad iterativa
from arbol import Nodo

def DFS_prof_iter(nodo, solucion):
  for limite in range(0,100):
    visitados=[]
    sol = buscar_solucion_DFS_Rec(nodo, solucion, visitados, limite)
    if sol!=None:
```

```
        return sol

  def buscar_solucion_DFS_Rec(nodo, solucion, visitados, limite):
    if limite > 0:
      visitados.append(nodo)
      if nodo.get_datos() == solucion:
        return nodo
      else:
        # expandir nodos hijo (ciudades con conexión)
        dato_nodo = nodo.get_datos()
        lista_hijos=[]
        for un_hijo in conexiones[dato_nodo]:
          hijo=Nodo(un_hijo)
          if not hijo.en_lista(visitados):
            lista_hijos.append(hijo)

        nodo.set_hijos(lista_hijos)

        for nodo_hijo in nodo.get_hijos():
          if not nodo_hijo.get_datos() in visitados:
            # llamada recursiva
            sol = buscar_solucion_DFS_Rec(nodo_hijo, solucion, \
            visitados, limite-1)
            if sol != None:
              return sol

      return None


  if __name__ == "__main__":
    conexiones = {
      'Malaga':{'Salamanca', 'Madrid', 'Barcelona'},
      'Sevilla':{'Santiago', 'Madrid'},
      'Granada':{'Valencia'},
```


© RC Libros

```
    'Valencia':{'Barcelona'},
    'Madrid':{'Salamanca', 'Sevilla', 'Malaga', \
    'Barcelona','Santander'},
    'Salamanca':{'Malaga', 'Madrid'},
    'Santiago':{'Sevilla', 'Santander', 'Barcelona'},
    'Santander':{'Santiago', 'Madrid'},
    'Zaragoza':{'Barcelona'},
    'Barcelona':{'Zaragoza', 'Santiago', 'Madrid', 'Malaga', \
    'Valencia'}
    }
estado_inicial='Malaga'
solucion='Santiago'
nodo_inicial = Nodo(estado_inicial)
nodo = DFS_prof_iter(nodo_inicial, solucion)

# mostrar resultado
if nodo != None:
  resultado=[]
  while nodo.get_padre() != None:
      resultado.append(nodo.get_datos())
      nodo = nodo.get_padre()
  resultado.append(estado_inicial)
  resultado.reverse()
  print resultado
else:
  print "solución no encontrada"
```

Listado. 3-11 vuelos_bpi.py

La solución ofrecida por el programa es la misma que la búsqueda en amplitud, y por lo tanto una solución óptima.

['Málaga', 'Barcelona', 'Santiago']

BÚSQUEDA DE COSTE UNIFORME

Los métodos de búsqueda analizados hasta ahora nos permiten enfrentarnos a un espectro de problemas relativamente amplio, pero no todos. Pongamos como ejemplo el problema de las rutas por carretera. Con los algoritmos que ya conocemos podemos abordar el problema hasta cierto punto. Siempre podremos encontrar una solución, pero como ya vimos con la búsqueda en amplitud, para este caso particular la solución no es óptima. Y este es precisamente uno de los requerimientos del problema. ¿Cómo obtener el camino óptimo de menor distancia? La **búsqueda de coste uniforme** o ***Uniform-Cost Search*** (UCS) nos permite realizar una búsqueda en el espacio de estados teniendo en cuenta un nuevo factor: el factor **coste**.

En esta búsqueda se asigna a cada nodo un coste. Definimos la función g(n) como el coste de recorrer el camino que va desde el nodo raíz al nodo que estamos evaluando. Por lo tanto, el coste del camino será la suma de los costes de cada nodo del recorrido.

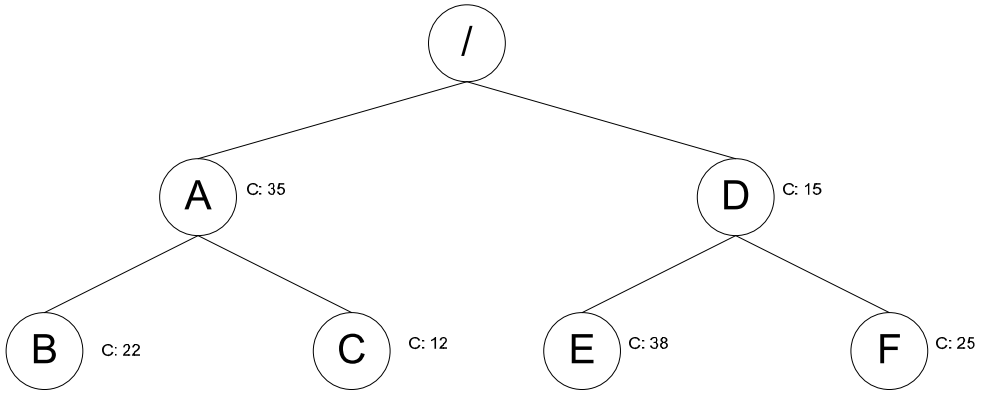


Fig. 3-7 Árbol con costes asociados (ponderado)

La figura 3-7 muestra un árbol de estados donde se indica el coste asociado a cada nodo. Algunos ejemplos de función de coste g(n) son:

$g(B) = 35 + 22 = 57$

$g(D) = 15$

$g(F) = 15 + 25 = 40$

© RC Libros

La búsqueda de coste uniforme utiliza una cola con prioridad en vez de una cola FIFO o una pila LIFO para almacenar los nodos frontera. Una cola con prioridad es una cola que se mantiene ordenada. En este caso, la cola se ordena según el coste de cada nodo. También es posible usar una cola normal y ordenarla en cada paso, o alternativamente, extraer el menor de los valores de la cola.

Se muestra seguidamente el algoritmo en pseudocódigo de la búsqueda de coste uniforme.

```
nodo_inicial = estado inicial
nodos_frontera = Cola con prioridad
nodos_visitados = Lista
almacenar nodo_inicial en nodos_frontera
mientras nodos_frontera no vacío:
    ordenar la lista de nodos_frontera según el coste
    nodo_actual = extraer el primer nodo de nodos_frontera
    si nodo_actual == solución:
           salir con solución

    introducir nodo_actual en nodos_visitados
    por cada operador:
        nodo_hijo = operador(nodo_actual)
        si nodo_hijo no en nodos_visitados:
           si nodo_hijo en nodos_frontera:
                  si coste de nodo_hijo < nodo en nodos_frontera:
                      sustituir nodo_hijo en nodos_frontera
           si no:
                  introducir nodo_hijo en nodos_frontera
```

Listado. 3-12 Pseudocódigo búsqueda coste uniforme

Por lo tanto, cada vez que seleccionamos un nodo para expandir, nos quedamos con aquel que tiene menor coste.

La búsqueda de coste uniforme es óptima en todo caso, ya que siempre escogemos el nodo de menor coste acumulado de la lista de nodos frontera. Cuando encontramos un nodo solución estamos seguros de que no había una solución de coste menor, ya que si no la habríamos encontrado antes. La búsqueda será completa siempre que la profundidad del árbol sea finita y los costes sean positivos.

La complejidad, sin embargo, no es tan obvia. Hay que tener en cuenta que no podemos caracterizarla en función de la profundidad del árbol, ya que no sabemos a priori por qué ramas descenderá ni a qué profundidad. El factor de ramificación por sí solo tampoco nos permite encontrar una medida de la complejidad de la búsqueda. En este caso vamos a intentar acotarla a partir del coste de la solución.

Suponiendo que la solución óptima se encuentra en el nodo N, que los costes de todos los nodos son iguales o superiores a un valor c y que el factor de ramificación es b, la complejidad espacial y temporal del algoritmo es $O(b^{1+(g(N)/c)})$. La búsqueda en amplitud puede caracterizarse como un caso particular de la búsqueda de coste uniforme en el que todos los costes son iguales; sin embargo, en condiciones no favorables, el coste puede llegar a ser bastante superior al de la búsqueda en amplitud $O(b^d)$.

Retomando el problema del viaje por carretera, vamos a ver cómo aplicar este algoritmo con un ejemplo. Dado el mapa de la figura 3-8, queremos encontrar el camino más corto desde Málaga a Santiago. Veamos paso por paso cómo el algoritmo va seleccionando los nodos. Al lado de cada nodo indicamos el coste acumulado. En la lista de nodos frontera ponemos en negrita aquellos nodos que se añaden nuevos o que cambian su valor g(n).

Paso 1.

Nodos frontera: **Málaga(0)**

Seleccionamos: Málaga(0)

Hijos no visitados: Madrid(513), Granada(125)

Paso 2.

Nodos frontera: **Granada(125), Madrid(513)**

Seleccionamos: Granada(125)

Hijos no visitados: Madrid(548), Valencia(616)

© RC Libros

Paso 3.

Nodos frontera: Madrid(513), **Valencia(616)**

Seleccionamos: Madrid(513)

Hijos no visitados: Valencia(868), Barcelona(1116), Zaragoza(826), Santander(950), Santiago(1112), Salamanca(716), Sevilla(1027)

Paso 4.

Nodos frontera: Valencia(616), **Salamanca(716), Zaragoza(826), Santander(950), Santiago(1112), Barcelona(1116), Sevilla(1027)**

Seleccionamos: Valencia(616)

Hijos no visitados: Barcelona(962), Zaragoza(925)

Paso 5.

Nodos frontera: Salamanca(716), Zaragoza(826), Santander(950), **Barcelona(962)**, Santiago(1112), Sevilla(1027)

Seleccionamos: Salamanca(716)

Hijos no visitados: Santiago(1106)

Paso 6.

Nodos frontera: Zaragoza(826), Santander(950), Barcelona(962), **Santiago(1106)**, Sevilla(1027)

Seleccionamos: Zaragoza(826)

Hijos no visitados: Barcelona(1122), Santander(1220)

Paso 7.

Nodos frontera: Santander(950), Barcelona(962), Santiago(1106), Sevilla(1027)

Seleccionamos: Santander(950)

Hijos no visitados: Ninguno

© RC Libros

Paso 8.

Nodos frontera: Barcelona(962), Santiago(1106), Sevilla(1027)

Seleccionamos: Barcelona(962)

Hijos no visitados: Ninguno

Paso 9.

Nodos frontera: Santiago(1106), Sevilla(1027)

Seleccionamos: Santiago(1106)

En este punto hemos encontrado el nodo objetivo y por lo tanto la solución que buscábamos con un coste de 1.106 kilómetros. Hemos necesitado 9 pasos para encontrar la solución óptima.

En el listado 3-13 tenemos una posible implementación del algoritmo en python. Por simplicidad no hemos utilizado ninguna librería adicional para poder usar una cola con prioridad (que no incluye python en su instalación por defecto). En su defecto hemos usado la función *sorted()* para ordenar la lista de nodos frontera. El problema con el que nos enfrentamos es que *sorted()* no sabe cómo ordenar objetos de la clase Nodo (queremos ordenar la lista por el coste acumulado g(n)). Afortunadamente, esta función permite especificar como parámetro una función externa para indicarle cómo hacer la comparación.

```
nodos_frontera = sorted(nodos_frontera, cmp=compara)
```

Aquí le decimos que utilice la función *compara(x, y)* para decidir cómo ordenar los nodos. Esta función recibe dos objetos x e y para ser comparados, y se espera que devuelva un valor negativo si x<y, un valor positivo si x>y y el valor 0 si x=y.

© RC Libros

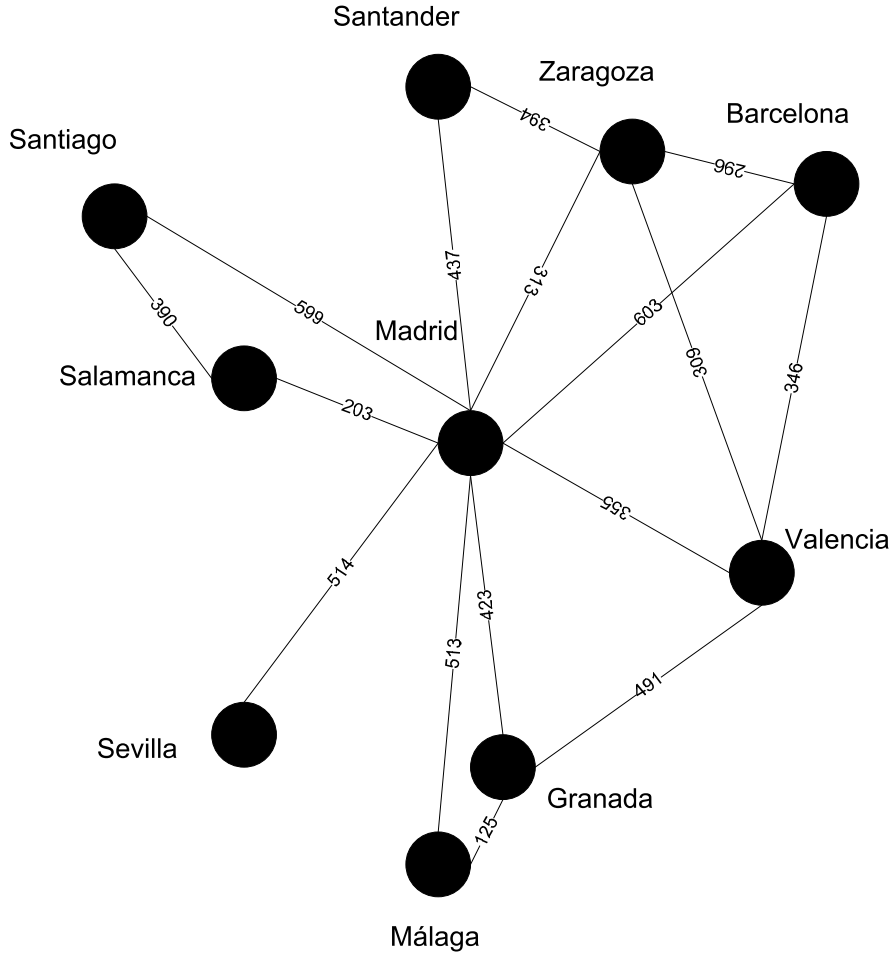


Fig. 3-8 Mapa de carreteras

```
# Viaje por carretera con búsqueda de coste uniforme
from arbol import Nodo

def compara(x, y):
        return x.get_coste() - y.get_coste()

def buscar_solucion_UCS(conexiones, estado_inicial, solucion):
  solucionado=False
  nodos_visitados=[]
```

```
    nodos_frontera=[]
    nodo_inicial = Nodo(estado_inicial)
    nodo_inicial.set_coste(0)
    nodos_frontera.append(nodo_inicial)
    while (not solucionado) and len(nodos_frontera)!=0:
      # ordenar la lista de nodos frontera
      nodos_frontera = sorted(nodos_frontera, cmp=compara)
      nodo=nodos_frontera[0]
      # extraer nodo y añadirlo a visitados
      nodos_visitados.append(nodos_frontera.pop(0))
      if nodo.get_datos() == solucion:
        # solución encontrada
        solucionado=True
        return nodo
      else:
        # expandir nodos hijo (ciudades con conexión)
        dato_nodo = nodo.get_datos()
        lista_hijos=[]
        for un_hijo in conexiones[dato_nodo]:
          hijo=Nodo(un_hijo)
          coste = conexiones[dato_nodo][un_hijo]
          hijo.set_coste(nodo.get_coste() + coste)
          lista_hijos.append(hijo)
          if not hijo.en_lista(nodos_visitados):
            # si está en la lista lo sustituimos con
            # el nuevo valor de coste si es menor
            if hijo.en_lista(nodos_frontera):
              for n in nodos_frontera:
                if n.igual(hijo) and n.get_coste()>hijo.get_coste():
                  nodos_frontera.remove(n)
                  nodos_frontera.append(hijo)
            else :
              nodos_frontera.append(hijo)
```


© RC Libros

```
        nodo.set_hijos(lista_hijos)

  if __name__ == "__main__":
    conexiones = {
      'Malaga':{'Granada':125, 'Madrid':513},
      'Sevilla':{'Madrid':514},
      'Granada':{'Malaga':125, 'Madrid':423, 'Valencia':491},
      'Valencia':{'Granada':491, 'Madrid':356, 'Zaragoza':309, \
      'Barcelona':346},
      'Madrid':{'Salamanca':203, 'Sevilla':514, 'Malaga':513, \
      'Granada':423, 'Barcelona':603, 'Santander':437, 'Valencia':356,
\
      'Zaragoza':313, 'Santander':437, 'Santiago':599},
      'Salamanca':{'Santiago':390, 'Madrid':203},
      'Santiago':{'Salamanca':390, 'Madrid':599},
      'Santander':{'Madrid':437, 'Zaragoza':394},
      'Zaragoza':{'Barcelona':296, 'Valencia':309, 'Madrid':313},
      'Barcelona':{'Zaragoza':296, 'Madrid':603, 'Valencia':346}
      }
    estado_inicial='Malaga'
    solucion='Santiago'
    nodo_solucion = buscar_solucion_UCS(conexiones, estado_inicial, \
    solucion)
    # mostrar resultado
    resultado=[]
    nodo=nodo_solucion
    while nodo.get_padre() != None:
      resultado.append(nodo.get_datos())
      nodo = nodo.get_padre()
    resultado.append(estado_inicial)
    resultado.reverse()
    print resultado
    print 'Coste: ' + str(nodo_solucion.get_coste())
```

Listado. 3-13 carretera_ucs.py

© RC Libros

El resultado de la ejecución es:

['Málaga', 'Madrid', 'Salamanca', 'Santiago']

Coste: 1.106

Este es el camino que necesita menor número de kilómetros. Hay caminos que aun pasando por menos ciudades cubren mayor distancia, como el siguiente:

['Málaga', 'Madrid', 'Santiago']

Coste: 1.112

© RC Libros

4 Búsqueda informada

FUNCIÓN HEURÍSTICA

En el capítulo anterior hemos analizado la búsqueda en profundidad y la búsqueda en amplitud, que utilizaban como criterio de búsqueda solamente la aplicación de los operadores que se habían definido para construir el árbol de estados. Ambas búsquedas se diferenciaban en la estrategia utilizada para recorrer el árbol. También se ha tratado la búsqueda de coste uniforme, en la que introducíamos una nueva componente: el coste. Recordemos que habíamos definido una función g(x) llamada coste que indicaba el coste total de un camino desde la raíz a un nodo cualquiera. Gracias a esta función, la búsqueda de coste uniforme es capaz de encontrar una solución óptima para un problema en el que el coste de tomar una decisión u otra no es igual en todos los casos, ya sea en términos de distancia, tiempo, dinero, dificultad o cualquier otra magnitud.

Algunos problemas, por su naturaleza, ofrecen información que puede ser útil a la hora de resolverlos. Gracias a esta información podemos tratar de mejorar la búsqueda para que nos lleve más directamente a la solución buscada primando el recorrido del árbol por ramas que parecen más prometedoras en detrimento de otras que lo son menos. Recordemos del anterior capítulo cómo la búsqueda de coste uniforme exploraba ramas que, aun teniendo un coste g(n) inferior a otras, luego eran descartadas por no conducir a una solución óptima o directamente no conducir a ninguna solución.

La función **heurística**, que llamaremos h(n), trata de guiar la búsqueda para llegar de forma más rápida a la solución. A partir de la información disponible, la función heurística intenta estimar el coste del mejor camino (menor coste) desde el nodo n hasta el nodo objetivo. Por lo tanto, la elección de una buena función heurística es crucial para obtener una buena solución.

Hay que hacer notar que mientras la función de coste g(n) se calcula a partir del camino que va desde el nodo raíz al nodo actual, la función heurística h(n) depende únicamente del nodo que se está analizando en ese momento.

Tomemos como ejemplo el problema del puzle lineal. Necesitamos un criterio que nos indique si un movimiento nos acerca o nos aleja de la solución. Este criterio nos permitirá comparar el nodo padre con los nodos hijo. Esta comparación la haremos en los términos de una función heurística h(n) que estimará la distancia desde un nodo cualquiera al nodo objetivo. Para el caso del puzle lineal, una función que nos puede dar una estimación de la distancia al nodo objetivo es el número de piezas mal colocadas, es decir, más piezas bien colocadas indican que estamos más cerca de la solución. De esta forma, un nodo hijo que tiene más piezas descolocadas que su padre nos dice que el movimiento empeora la solución. Dicho de otra manera, el nodo hijo es de peor calidad que el nodo padre.

Alternativamente, podríamos usar otras funciones heurísticas h(n); por ejemplo, podemos medir la calidad de un nodo por el número de movimientos que lo separan de un nodo objetivo en vez de por el número de piezas mal colocadas. Esta es probablemente una mejor heurística, pero por sencillez en la implementación nos quedaremos con la primera.

```
# Puzle Lineal con heurística
from arbol import Nodo


def buscar_solucion_heuristica(nodo_inicial, solucion, visitados):
  visitados.append(nodo_inicial.get_datos())
  if nodo_inicial.get_datos() == solucion:
    return nodo_inicial
  else:
    # expandir nodos sucesores (hijos)
    dato_nodo = nodo_inicial.get_datos()
    hijo=[dato_nodo[1], dato_nodo[0], dato_nodo[2], dato_nodo[3]]
```


© RC Libros

```
    hijo_izquierdo = Nodo(hijo)
    hijo=[dato_nodo[0], dato_nodo[2], dato_nodo[1], dato_nodo[3]]
    hijo_central = Nodo(hijo)
    hijo=[dato_nodo[0], dato_nodo[1], dato_nodo[3], dato_nodo[2]]
    hijo_derecho = Nodo(hijo)
    nodo_inicial.set_hijos([hijo_izquierdo, hijo_central, \
    hijo_derecho])

    for nodo_hijo in nodo_inicial.get_hijos():
      if not nodo_hijo.get_datos() in visitados \
      and mejora(nodo_inicial, nodo_hijo):
        # llamada recursiva
        sol = buscar_solucion_heuristica(nodo_hijo, solucion, \
        visitados)
        if sol != None:
          return sol

    return None

def mejora(nodo_padre, nodo_hijo):
    calidad_padre=0
    calidad_hijo=0
    dato_padre = nodo_padre.get_datos()
    dato_hijo = nodo_hijo.get_datos()
    for n in range(1,len(dato_padre)):
        if (dato_padre[n]>dato_padre[n-1]):
            calidad_padre = calidad_padre + 1;
        if (dato_hijo[n]>dato_hijo[n-1]):
            calidad_hijo = calidad_hijo + 1;

    if calidad_hijo>=calidad_padre:
        return True
    else:
        return False
```

```
if __name__ == "__main__":
  estado_inicial=[4,2,3,1]
  solucion=[1,2,3,4]
  nodo_solucion = None
  visitados=[]
  nodo_inicial = Nodo(estado_inicial)
  nodo = buscar_solucion_heuristica(nodo_inicial, solucion, \
  visitados)

  # mostrar resultado
  resultado=[]
  while nodo.get_padre() != None:
      resultado.append(nodo.get_datos())
      nodo = nodo.get_padre()
  resultado.append(estado_inicial)
  resultado.reverse()
  print resultado
```

Listado. 4-1 puzlelinealheu.py

La ejecución del programa nos ofrece una solución similar a la que encontramos con la búsqueda en amplitud, pero usando una búsqueda en profundidad y, por lo tanto, menos costosa en cuanto al uso de memoria:

[[4, 2, 3, 1], [2, 4, 3, 1], [2, 3, 4, 1], [2, 3, 1, 4], [2, 1, 3, 4], [1, 2, 3, 4]]

Para el ejemplo hemos utilizado una función llamada *mejora()* que indica si el nodo hijo mejora respecto al nodo padre en los términos de calidad que hemos definido más arriba. Es decir, es una función booleana que devuelve el valor verdadero si el nodo hijo tiene más piezas en el lugar correcto que el padre, o al menos las mismas.

© RC Libros

BÚSQUEDA CON VUELTA ATRÁS (BACKTRACKING)

Una técnica comúnmente usada para mejorar el rendimiento temporal de la búsqueda en profundidad usando una función heurística es la llamada **búsqueda con vuelta atrás o backtracking**. Cuando exploramos una rama del árbol, hay situaciones que sabemos que no nos van a llevar a una solución o en el mejor de los casos, empeorarán la solución que se está analizando en ese momento. En otras palabras, que el nodo hijo empeora la situación respecto al nodo padre y por lo tanto nos aleja de la solución en vez de acercarnos a ella. Si un movimiento nos lleva a una rama en la que sabemos que no llegaremos a una solución, decimos que el camino (la solución) que estamos analizando no es **completable**. Como no tiene sentido seguir explorando una rama que es no completable, lo que hacemos es que dejamos de explorar y volvemos hacia atrás (de ahí el nombre del algoritmo).

```
Función DFS_backtracking(nodo, solución, visitados):
  Añadir nodo a visitados
  Si nodo == solución:
    Salir con solución
  Si no:
    Por cada operador:
      nodo_hijo = operador(nodo)
      Si nodo_hijo no en visitados y es_completable(nodo_hijo):
        s = DFS_backtracking(nodo_hijo, solución, visitados)
        Salir con s

nodo = estado inicial
visitados = Lista
solucion = solución
s = DFS_backtracking(nodo, solucion, visitados)
```

Listado. 4-2 Pseudocódigo búsqueda con vuelta atrás

Retomemos el problema de la PLE que propusimos en el capítulo 2. Recordemos que se nos pedía maximizar una función que representaba el beneficio resultante de la fabricación de x_1 camisetas y x_2 pantalones.

$$f(x_1, x_2) = (12 - 6)x_1 + (8 - 4)x_2$$

Con las siguientes restricciones:

$$7x_1 + 4x_2 \leq 150$$

$$6x_1 + 5x_2 \leq 160$$

$$x_1 \geq 0$$
$$x_2 \geq 0$$

Nuestro objetivo es encontrar una asignación de valores enteros para las variables x_1 y x_2 que, cumpliendo con las restricciones impuestas, hagan que la función f tome el máximo valor posible. Recordemos también que habíamos llegado a la conclusión de que x_1 estaba acotado entre 0 y 50 y x_2 entre 0 y 75.

Al explorar el árbol y encontrar asignaciones que hacen que se incumplan las restricciones, no tiene sentido seguir explorando la rama, por lo que podemos seguir con la siguiente. El listado que sigue demuestra cómo afrontar el problema de la PLE usando técnicas de backtracking. En este ejemplo no vamos a usar la clase Nodo ni a construir el árbol como hemos hecho en otros ejemplos, ya que, a diferencia de los otros problemas tratados, en este no nos importan los pasos intermedios necesarios para llegar a una solución. Solo el resultado final. En el problema del camino por carretera sí necesitamos conocer por qué ciudades transcurre el camino, pero aquí solo nos interesan los valores finales y óptimos de x_1 y x_2. Por lo tanto, sí que vamos a realizar una búsqueda en un árbol, pero vamos a ir generando los nodos según los vamos a ir necesitando.

Para nuestro problema, que es bastante sencillo ya que solo tiene dos variables, vamos a necesitar hacer una búsqueda en un árbol de solo dos niveles de profundidad. En el primer nivel (sin contar el nodo raíz) vamos a generar los posibles valores de X_1, es decir, de 0 a 50. En el segundo nivel, por cada posible valor de x_1 vamos a generar todos los posibles valores de x_2 que son los valores entre 0 y 75. En total tendremos que examinar $50 \times 75 = 3750$ nodos. Sin embargo, ¿es necesario examinarlos todos? Supongamos que x_1=25 y recordemos que una de las restricciones era:

$$7x_1 + 4x_2 \leq 150$$

© RC Libros

Como $7x_1 = 7 \times 25 = 175$ es mayor que 150, queda claro que esta asignación incumple la restricción sea cual sea el valor de x_2. En este escenario no tiene sentido desplegar los 75 nodos hijo para las asignaciones de x_2.

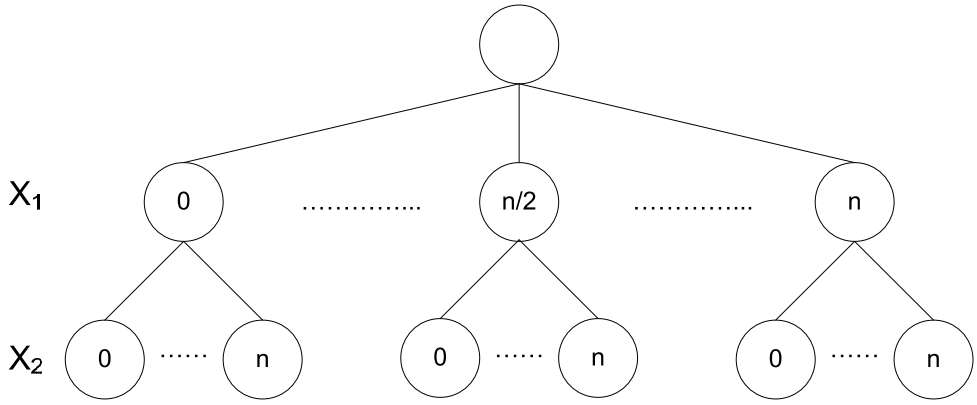


Fig. 4-1 Despliegue del árbol para el problema PLE

En nuestro ejemplo tenemos solo dos variables y 3.750 nodos por examinar, pero cualquier problema mediano puede tener bastantes variables y un número de nodos mucho mayor, por lo que nos va a resultar muy interesante aplicar esta técnica.

El listado siguiente muestra la implementación del algoritmo de backtracking para resolver nuestro problema de fabricación de camisetas y pantalones.

```
# PLE con backtracking
def backtracking(variables, rango_variables, optimo, profundidad):
  min=rango_variables[profundidad][0]
  max=rango_variables[profundidad][1]
  for v in range(min, max):
    variables[profundidad]=v
    if profundidad < len(variables)-1:
      # Es completable si no incumple ninguna restricción
      if es_completable(variables):
        optimo = backtracking(variables[:], rango_variables, \
```

```
          optimo, profundidad+1)
      else:
        # estamos en una hoja. Comprobamos solución.
        sol=evalua_solucion(variables)
        if sol>evalua_solucion(optimo) and es_completable(variables):
          optimo = (variables[0], variables[1])

    return optimo


def evalua_solucion(variables):
    x1=variables[0]
    x2=variables[1]
    val = (12-6)*x1+(8-4)*x2
    return val


def es_completable(variables):
    x1=variables[0]
    x2=variables[1]
    val1=7*x1+4*x2
    val2=6*x1+5*x2
    if val1<=150 and val2<=160:
      return True
    else:
      return False


if __name__ == "__main__":
    # valores de las variables x1 y x2
    variables=[0,0]
    # rangos de las variables x1 y x2
    rango_variables=[(0,51),(0,76)]
    # mejor solución encontrada
```


© RC Libros

```
optimo=(0,0)
sol=backtracking(variables[:], rango_variables, optimo, 0)
print "Mejor solución:"
print str(sol[0])+" Pantalones."
print str(sol[1])+" Camisetas."
print "Beneficio: "+str(evalua_solucion(sol))
```

Listado. 4-3 ple_backtracking.py

La ejecución del programa nos indica que la mejor opción es fabricar 10 pantalones y 20 camisetas, lo que nos dejará un beneficio de 140 euros.

ALGORITMO A*

Utilizando el concepto de coste, la búsqueda de coste uniforme nos permite llegar a una solución óptima. Sin embargo, hay que pagar un precio por ello. En condiciones desfavorables, la complejidad temporal y espacial del algoritmo puede crecer bastante. El algoritmo A* (A estrella) trata de paliar la situación intentando que la convergencia hasta la solución sea más rápida. Para ello, hace uso de una función heurística h(n) que trata de estimar el coste desde el nodo n hasta el nodo objetivo.

Recordemos que la búsqueda de coste uniforme hacía uso de la función g(n) que llamábamos coste (coste acumulado desde el nodo raíz hasta el nodo n), de esta forma, a cada nodo se le hacía corresponder un valor según la siguiente función de evaluación:

$$f(n) = g(n)$$

El algoritmo A* se basa en el mismo concepto, pero usa la siguiente función f(n) como función de evaluación.

$$f(n) = g(n) + h(n)$$

Es decir, la suma del coste acumulado desde el nodo raíz más el coste estimado hasta el nodo solución. En cada iteración del algoritmo iremos seleccionando aquellos nodos con mayor valor f(n). Visto de esta manera, la búsqueda de coste uniforme es un caso especial del algoritmo A* en el que h(n) toma el valor cero.

© RC Libros

Hay que insistir en que h(n) es una estimación; es decir, tendremos que basarnos en la información de la que disponemos para hacer la estimación. La elección de una buena función heurística h(n) es muy importante para que el algoritmo llegue a una solución rápida en pocos pasos.

```
nodo_inicial = estado inicial
nodos_frontera = Cola con prioridad
nodos_visitados = Lista
almacenar nodo_inicial en nodos_frontera
mientras nodos_frontera no vacío:
    calcular el valor f(n)=g(n)+h(n) para cada nodo frontera
ordenar la lista de nodos_frontera según f(n)
    nodo_actual = extraer el primer nodo de nodos_frontera
    si nodo_actual == solución:
           salir con solución

    introducir nodo_actual en nodos_visitados
    por cada operador:
        nodo_hijo = operador(nodo_actual)
        si nodo_hijo no en nodos_visitados:
           si nodo_hijo en nodos_frontera:
                  si coste de nodo_hijo < nodo en nodos_frontera:
                      sustituir nodo_hijo en nodos_frontera
           si no:
                  introducir nodo_hijo en nodos_frontera
```

*Listado. 4-4 Pseudocódigo algoritmo A**

Retomemos el ejemplo del cálculo de la mejor ruta para el viaje por carretera. Cuando analizamos este problema bajo el prisma de la búsqueda de coste uniforme, utilizamos como función de coste g(n) la distancia recorrida desde la ciudad inicial hasta la ciudad representada por el nodo frontera que se evaluaba en cada caso. ¿Qué función heurística h(n) podemos usar para este mismo problema? Parece que una buena estimación podría ser la distancia en línea recta desde el nodo que estamos analizando hasta el nodo de la ciudad de destino. Siempre y cuando

© RC Libros

podamos obtener este dato. Supongamos que disponemos de la siguiente tabla con las coordenadas geográficas de cada una de las ciudades.

Ciudad	Latitud	Longitud
Santander	43.28 N	3.48 O
Santiago de Compostela	42.52 N	8.33 O
Salamanca	40.57 N	5.40 O
Madrid	40.24 N	3.41 O
Zaragoza	41.39 N	0.52 O
Barcelona	41.23 N	2.11 E
Sevilla	37.23 N	5.59 O
Málaga	36.43 N	4.25 O
Granada	37.11 N	3.35 O
Valencia	39.28 N	0.22 O

A partir de las coordenadas de las ciudades, podemos obtener su distancia en línea recta medida en kilómetros. Son las llamadas distancias geodésicas. En el ecuador los meridianos de longitud separados por un grado se encuentran a una distancia de 111,32km, mientras que en los polos los meridianos convergen. Cada grado de longitud y de latitud se divide en 60 minutos y cada minuto en 60 segundos. De este modo se puede asignar una localización precisa a cualquier lugar de la Tierra. Los detalles para el cálculo de distancias geodésicas escapan del ámbito de este libro, así que nos limitaremos a decir que para hacer el cálculo usaremos la siguiente función.

```
def geodist(lat1, lon1, lat2, lon2):
  grad_rad = 0.01745329
  rad_grad = 57.29577951
```

```
    longitud = long1-long2
    val = (sin(lat1*grad_rad)*sin(lat2*grad_rad)) \
    + (cos(lat1*grad_rad)*cos(lat2*grad_rad)*cos(longitud*grad_rad))
    return (acos(val)*rad_grad)*111.32
```

La función *geodist()* toma como parámetros la latitud y longitud de las dos coordenadas sobre las que queremos conocer la distancia y nos devuelve la distancia en kilómetros. La función devuelve un número real, por lo que habrá que hacer un casting a entero para poder usar su valor con la función *sorted()*. El código del programa completo es el siguiente.

```
# Viaje por carretera con búsqueda A*
from arbol import Nodo
from math import sin, cos, acos

def compara(x, y):
  # g(n)+h(n) para ciudad x
  lat1=coord[x.get_datos()][0]
  lon1=coord[x.get_datos()][1]
  lat2=coord[solucion][0]
  lon2=coord[solucion][1]
  d=int(geodist(lat1, lon1, lat2, lon2))
  c1=x.get_coste()+d
  # g(n)+h(n) para ciudad y
  lat1=coord[y.get_datos()][0]
  lon1=coord[y.get_datos()][1]
  lat2=coord[solucion][0]
  lon2=coord[solucion][1]
  d=int(geodist(lat1, lon1, lat2, lon2))
  c2=y.get_coste()+d
  return c1-c2


def geodist(lat1, lon1, lat2, lon2):
  grad_rad = 0.01745329
```


© RC Libros

```
    rad_grad = 57.29577951
    longitud = lon1-lon2
    val = (sin(lat1*grad_rad)*sin(lat2*grad_rad)) \
    + (cos(lat1*grad_rad)*cos(lat2*grad_rad)*cos(longitud*grad_rad))
    return (acos(val)*rad_grad)*111.32


def buscar_solucion_UCS(conexiones, estado_inicial, solucion):
    solucionado=False
    nodos_visitados=[]
    nodos_frontera=[]
    nodo_inicial = Nodo(estado_inicial)
    nodo_inicial.set_coste(0)
    nodos_frontera.append(nodo_inicial)
    while (not solucionado) and len(nodos_frontera)!=0:
      # ordenar la lista de nodos frontera
      nodos_frontera = sorted(nodos_frontera, cmp=compara)
      nodo=nodos_frontera[0]
      # extraer nodo y añadirlo a visitados
      nodos_visitados.append(nodos_frontera.pop(0))
      if nodo.get_datos() == solucion:
        # solución encontrada
        solucionado=True
        return nodo
      else:
        # expandir nodos hijo (ciudades con conexión)
        dato_nodo = nodo.get_datos()
        lista_hijos=[]
        for un_hijo in conexiones[dato_nodo]:
          hijo=Nodo(un_hijo)
          # cálculo g(n): coste acumulado
          coste = conexiones[dato_nodo][un_hijo]
          hijo.set_coste(nodo.get_coste() + coste)
          lista_hijos.append(hijo)
```

```
          if not hijo.en_lista(nodos_visitados):
            # si está en la lista lo sustituimos con
            # el nuevo valor de coste si es menor
            if hijo.en_lista(nodos_frontera):
              for n in nodos_frontera:
                if n.igual(hijo) and n.get_coste()>hijo.get_coste():
                  nodos_frontera.remove(n)
                  nodos_frontera.append(hijo)
            else :
              nodos_frontera.append(hijo)

        nodo.set_hijos(lista_hijos)


if __name__ == "__main__":
  conexiones = {
    'Malaga':{'Granada':125, 'Madrid':513},
    'Sevilla':{'Madrid':514},
    'Granada':{'Malaga':125, 'Madrid':423, 'Valencia':491},
    'Valencia':{'Granada':491, 'Madrid':356, 'Zaragoza':309, \
    'Barcelona':346},
    'Madrid':{'Salamanca':203, 'Sevilla':514, 'Malaga':513, \
    'Granada':423,'Barcelona':603, 'Santander':437, 'Valencia':356,\
    'Zaragoza':313, 'Santander':437, 'Santiago':599},
    'Salamanca':{'Santiago':390, 'Madrid':203},
    'Santiago':{'Salamanca':390, 'Madrid':599},
    'Santander':{'Madrid':437, 'Zaragoza':394},
    'Zaragoza':{'Barcelona':296, 'Valencia':309, 'Madrid':313},
    'Barcelona':{'Zaragoza':296, 'Madrid':603, 'Valencia':346}
    }

  coord = {
    'Malaga':(36.43, -4.24),
    'Sevilla':(37.23, -5.59),
```


© RC Libros

```
    'Granada':(37.11, -3.35),
    'Valencia':(39.28, -0.22),
    'Madrid':(40.24, -3.41),
    'Salamanca':(40.57, -5.40),
    'Santiago':(42.52, -8.33),
    'Santander':(43.28, -3.48),
    'Zaragoza':(41.39, -0.52),
    'Barcelona':(41.23, +2.11)
    }

estado_inicial='Malaga'
solucion='Santiago'
nodo_solucion = buscar_solucion_UCS(conexiones, estado_inicial, \
solucion)
# mostrar resultado
resultado=[]
nodo=nodo_solucion
while nodo.get_padre() != None:
  resultado.append(nodo.get_datos())
  nodo = nodo.get_padre()
resultado.append(estado_inicial)
resultado.reverse()
print resultado
print 'Coste: ' + str(nodo_solucion.get_coste())
```

Listado. 4-5 carreteraAstar.py

Recordemos que la búsqueda de coste uniforme necesitó 9 pasos para llegar a una solución. Tal y como hicimos en el capítulo anterior, vamos a hacer un seguimiento paso por paso del algoritmo para poder comparar su efectividad. Entre paréntesis se indica el coste g(n) más el valor de h(n) correspondiente a la distancia en línea recta. Se adjunta también la tabla de distancias en línea recta desde el nodo objetivo (Santiago) hasta el resto de ciudades.

© RC Libros

Ciudad	Distancia en línea recta hasta Santiago
Zaragoza	658
Santiago	0
Santander	404
Málaga	763
Salamanca	326
Madrid	482
Valencia	771
Sevilla	633
Barcelona	876
Granada	737

Paso 1.

Nodos frontera: **Málaga(**0 + 763 = 763**)**

Seleccionamos: Málaga(0 + 763 = 763)

Hijos no visitados: Madrid(513 + 482 = 995), Granada(125 + 737 = 862)

Paso 2.

Nodos frontera: **Granada(**125 + 737 = 862**), Madrid(**513 + 482 = 995**)**

Seleccionamos: Granada(125 + 737 = 862)

Hijos no visitados: Valencia(616 + 771 = 1387)

Paso 3.

Nodos frontera: Madrid(513 + 482 = 995), **Valencia(**616 + 771 = 1387**)**

© RC Libros

Seleccionamos: Madrid(513 + 482 = 995)

Hijos no visitados: Valencia(616 + 771 = 1387), Barcelona(1116 + 876 = 1992), Zaragoza(826 + 658 = 1484), Santander(950 + 404 = 1354), Santiago(1112 + 0 = 1112), Salamanca(716 + 326 = 1042), Sevilla(1027 + 633 = 1660)

Paso 4.

Nodos frontera: **Salamanca**(716 + 326 = 1042), **Santiago**(1112 + 0 = 1112), **Santander**(950 + 404 = 1354), Valencia(616 + 771 = 1387), **Zaragoza**(826 + 658 = 1484), **Sevilla**(1027 + 633 = 1660), **Barcelona**(1116 + 876 = 1992)

Seleccionamos: **Salamanca**(716 + 326 = 1042)

Hijos no visitados: Santiago(1112 + 0 = 1112)

Paso 5.

Nodos frontera: Santiago(1112 + 0 = 1112), Santander(950 + 404 = 1354), Valencia(616 + 771 = 1387), Zaragoza(826 + 658 = 1484), Sevilla(1027 + 633 = 1660), Barcelona(1116 + 876 = 1992)

Seleccionamos: Santiago(1112 + 0 = 1112)

Con esto hemos llegado al nodo objetivo en solo 5 pasos contra los 9 de la búsqueda de coste uniforme. Una mejora de casi el 50%. Con árboles más grandes la mejora es incluso mayor.

Al seleccionar una función heurística h(n) hay que tener en cuenta que se deben cumplir las siguientes condiciones para que el algoritmo A* funcione correctamente.

- $h(n) \geq 0$ para todo n.
- $h(n) = 0$ si n es nodo objetivo.
- h(n) Ha de ser admisible.

Una función heurística **admisible** es aquella que nunca sobreestime el coste para llegar al nodo objetivo. Dicho de otra forma, la estimación que haga h(n) siempre ha de estar por debajo del coste real para llegar de n al nodo objetivo. En el ejemplo del viaje por carretera está claro que nunca vamos a sobreestimar el coste, ya que no hay un camino más corto que la línea recta. En este sentido, decimos que una heurística admisible es **optimista**, ya que su valor siempre está por debajo del valor del coste real.

© RC Libros

Para dejar claro el concepto vamos a ver otro ejemplo de heurística admisible: Una empresa fabricante de vehículos monta 4 tipos de ruedas en sus modelos de vehículos. Los tipos son: Tipo T para vehículos de gama baja, tipo H para vehículos de gama media, tipo V para vehículos de gama alta y tipo W para vehículos de lujo. A su vez, el fabricante trabaja con cuatro distribuidores de rueda distintos. Los cuatro distribuidores pueden servir los cuatro tipos de rueda, pero cada uno tiene precios diferentes. Por razones comerciales interesa trabajar con las cuatro distribuidoras, por lo que se ha decidido comprar un tipo de rueda a cada fabricante. Los precios en euros por fabricante y tipo de rueda son los siguientes:

	Tipo T	Tipo H	Tipo V	Tipo W
Empresa 1	20	30	20	40
Empresa 2	50	50	40	50
Empresa 3	60	55	50	60
Empresa 4	100	80	60	70

La empresa fabricante nos ha pedido que hagamos un estudio para seleccionar qué rueda comprar a cada fabricante de forma que el precio sea el más ventajoso posible.

El problema puede atacarse con cualquiera de los tipos de búsqueda que hemos analizado, ya que las soluciones pueden representarse en forma de árbol. En el primer nivel tendríamos cuatro nodos representando al fabricante seleccionado para la rueda tipo 1. En el siguiente nivel tendríamos 3 nodos por cada uno de los cuatro nodos del nivel 1 representando al fabricante para el tipo de rueda 2 y así sucesivamente. En este escenario, la elección de la función g(n) es directa. El coste en euros de cada tipo de rueda. Sin embargo, la elección de una función h(n) que además sea una heurística admisible no es tan evidente.

Una posible función heurística admisible podría ser la siguiente: Dado un estado n, la función h(n) será la suma de los precios más bajos de los tipos de rueda aún no asignados, teniendo solo en cuenta aquellas empresas no adjudicadas aún.

Por ejemplo, supongamos que hemos adjudicado el tipo T a la empresa 1 y el nodo actual n evalúa la posibilidad de asignar el tipo H a la empresa 2. Las funciones g(n) y h(n) serían:

© RC Libros

$$g(n) = 20 + 50 = 70$$

$$h(n) = 50 + 60 = 110$$

El valor 50 es el correspondiente al precio de la rueda tipo V de la empresa 3 y el valor 60 es el de la rueda tipo W también de la empresa 3. Como nos queda por escoger un precio para las ruedas tipo V y W de entre las empresas 3 y 4, queda claro que, sea cual sea la selección que hagamos, el precio será igual o superior a 110. Por lo tanto, la función h(n) es admisible. Se deja al lector como ejercicio la implementación de una solución a este problema usando el algoritmo A*, que es muy similar a la del viaje por carretera.

BÚSQUEDA LOCAL

Los algoritmos de búsqueda local tratan de mejorar una solución de forma iterativa, haciendo pequeñas modificaciones sobre una solución inicial en cada iteración, siempre y cuando no se viole ninguna de las restricciones del problema. Si esa modificación mejora a la original, nos quedamos con ella; si no, la descartamos. Hay toda una familia de algoritmos de búsqueda local, también llamados **voraces** (***greedy*** en inglés), de los que analizaremos los más importantes, que son los algoritmos de hill climbing, simulated annealing, búsqueda tabú y algoritmos genéticos.

La búsqueda local se utiliza en problemas en los que nos interesa el resultado y no las acciones necesarias para llegar a él, por lo tanto, no necesitamos almacenar todo el árbol de estados en memoria. En las búsquedas locales normalmente trabajamos sobre un solo estado, al que aplicamos alguna modificación con la esperanza de que el nuevo estado resultante se acerque más a la solución. Al estado resultante tras aplicarle el cambio lo denominamos **estado vecino**.

Retomemos el problema SAT que se planteó en el capítulo 2. Supongamos que tenemos la siguiente función booleana:

$$f(x_1, x_2, x_3, x_4) = (x_2 \vee x_4) \wedge (x_3 \vee x_1) \wedge (x_2 \vee \neg x_1)$$

Queremos encontrar una asignación de valores para las variables en la que la función f sea verdadera. Imaginamos la siguiente asignación de variables $x_1 = F, x_2, = F, x_3 = V, x_4 = F$. Esta asignación la podemos representar como un número

binario en el que la primera cifra se corresponde con la primera variable y así sucesivamente. En este caso, esta asignación podría representarse como 0010. ¿Cuáles serían los vecinos de este estado? Sería otra secuencia de 4 dígitos binarios en los que se ha modificado uno de los valores (una de las asignaciones). Los siguientes estados serían vecinos de 0010:

1010

0011

0110

La última asignación 0110 tiene la particularidad de que hace tomar el valor verdadero a la función y por lo tanto es una solución.

Consideremos ahora el problema TSP (problema del viajante de comercio). En el siguiente mapa vemos las ciudades y sus conexiones.

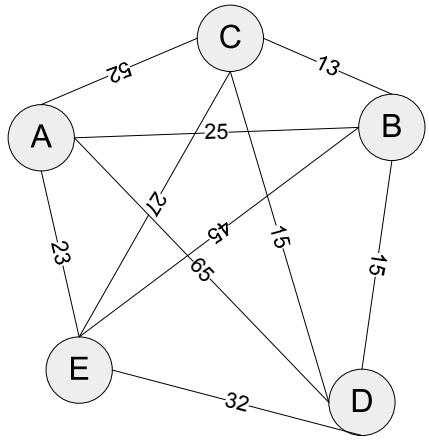


Fig. 4-2 Conexiones entre ciudades para el problema TSP

Un posible estado solución (no óptima) podría representarse como ABCED, que representa las ciudades y el orden en que se recorren. ¿Cuáles serían los posibles vecinos? En el ejemplo anterior del TSP usamos como operador para generar los vecinos el cambio de estado de una de las cifras binarias,es decir, cambiábamos un 1 por un 0, o viceversa. En el caso del TSP una posible operación para obtener un vecino puede ser intercambiar dos ciudades. De esta forma, algunos vecinos de ABCED son:

© RC Libros

BACED

CBAED

AECBD

Al aplicar los operadores para obtener los vecinos de un estado hay que tener en cuenta las restricciones propias del problema. Algunas de estas restricciones vendrán impuestas por el enunciado del problema, como las del problema PLE. Otras restricciones vendrán derivadas de la naturaleza del propio problema. Un ejemplo claro es el problema de las rutas por carreteras. Para obtener un vecino podemos intercambiar dos ciudades igual que hacemos con el TSP, pero puede darse el caso de que el intercambio dé lugar a una solución en la que dos ciudades consecutivas no estén conectadas por carretera. Por ejemplo, si a partir de la ruta:

Málaga – Granada – Madrid – Santiago.

Obtenemos un vecino intercambiando Málaga y Santiago:

Santiago – Granada – Madrid – Málaga.

Pero según el mapa de carreteras propuesto en el capítulo 2, no existe conexión por carretera entre Santiago y Granada.

Al proceso de comprobar si el vecino es una solución factible lo llamamos **satisfacción de restricciones**.

La siguiente figura 4-3 muestra un esquema genérico de cómo se comporta la búsqueda local. En el eje de las abscisas representamos el espacio de estados y en el eje de las ordenadas el valor de la función objetivo. Nuestro objetivo es encontrar el mayor valor de la función de evaluación en el eje de las ordenadas (o el menor si lo que queremos es por ejemplo obtener el camino mínimo). Supongamos que el estado actual que estamos analizando es el que marca la flecha en la figura. Un algoritmo de búsqueda local generará los vecinos de este estado y elegirá el mejor. En nuestro ejemplo, los vecinos son los puntos inmediatamente laterales al estado actual. Tanto por la izquierda como por la derecha. Pueden darse, pues, dos casos, que el algoritmo decida que es mejor escoger el vecino de la derecha. En ese caso un algoritmo de búsqueda local irá generando soluciones cada vez mejores según avance hacia la derecha (suponiendo que lo que se busca es el valor máximo de la función de evaluación) hasta que llegue al pico. El algoritmo interpretará entonces que ha encontrado el máximo de la función, cosa que como podemos observar en la figura, no es cierta. Lo que hemos encontrado es lo que se denomina un **máximo**

© RC Libros

local. Si se hubiera tomado la decisión de avanzar hacia la izquierda sí habríamos encontrado el **máximo global**, que es el valor que realmente nos interesa.

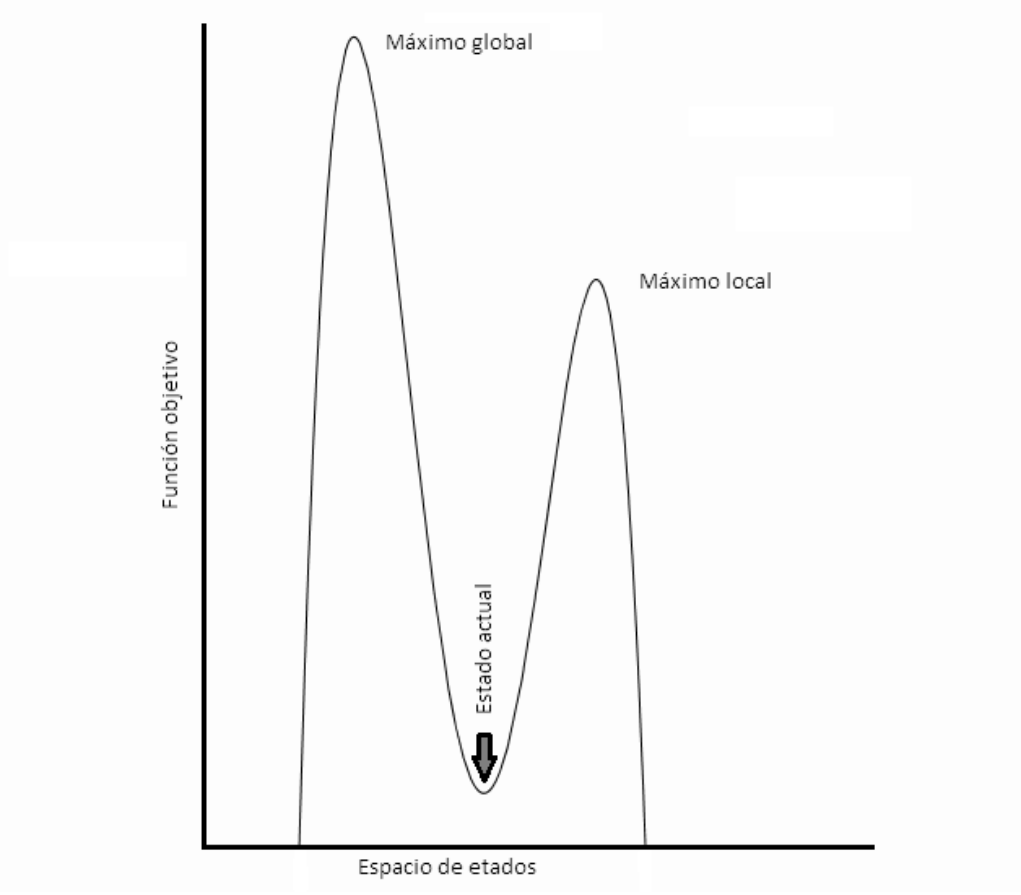


Fig. 4-3 Espacio de búsqueda representado como función

Algoritmos constructivos voraces

En general, para aplicar un algoritmo de búsqueda local necesitamos partir de una solución inicial que trataremos de ir mejorando. Como solución inicial podemos tomar una solución aleatoria, pero a veces interesa que la solución, aunque no sea óptima, sí cumpla una serie de restricciones y condiciones iniciales. Si además la solución es buena aunque no sea perfecta, mejor que mejor.

© RC Libros

Los algoritmos **constructivos voraces** son aquellos que utilizan una función heurística consistente en elegir la mejor opción en cada paso para ir construyendo una solución. Suelen usarse para generar soluciones iniciales y luego intentar mejorarlas mediantes técnicas como hill-climbing, algoritmos genéticos o algún otro de los que analizaremos en el resto del capítulo.

Aquí también es muy importante la elección de una buena función heurística, ya que además de asegurarnos una solución cuasi-óptima, o al menos aceptable, también influirá en si la solución es válida o no. No debemos perder de vista que el algoritmo constructivo ha de tener en cuenta las restricciones del problema. Pongamos como ejemplo el problema del viaje por carretera. De nuevo queremos encontrar un camino para ir de Málaga a Santiago. Como heurística constructiva elegiremos aquella que en cada paso selecciona el camino más corto disponible. Si empezamos en Málaga, la distancia más corta es la que nos lleva a Granada. Desde Granada, la ciudad más cercana es Madrid. En Madrid seleccionaremos Salamanca y finalmente Santiago, que es la única elección disponible desde Salamanca. Nuestro algoritmo constructivo ha generado la siguiente ruta a:

Málaga – Granada – Madrid – Salamanca – Santiago

Como ya vimos antes, esta no es la mejor solución, pero no está mal como punto de partida. ¿Y si queremos ir de Barcelona a Valencia? Aplicando nuestra heurística obtendríamos la siguiente ruta:

Barcelona – Zaragoza – Madrid – Salamanca – Santiago – Madrid –Valencia

Teniendo en cuenta que la ruta óptima es

Barcelona – Valencia

Nuestra heurística no se ha comportado demasiado bien en este caso.

Como se dijo antes, hay que tener muy en cuenta que la heurística respete las restricciones del problema, ya que podríamos caer en bucles o llegar a ciudades, que como Sevilla, no tengan salida, con lo que nunca alcanzaríamos la ciudad de destino.

A pesar de estos problemas, esta técnica puede llegar a ser muy poderosa para construir soluciones iniciales. Vamos a analizar dos algoritmos constructivos voraces bien conocidos. Se trata del algoritmo de Dijkstra para la construcción del camino mínimo en un grafo y el algoritmo de los ahorros de Clarke y Wright para el problema VRP (Vehicle Routing Problem).

© RC Libros

EL ALGORITMO DE DIJKSTRA

El algoritmo de **Dijkstra** es un algoritmo constructivo voraz capaz de encontrar el camino más corto desde un nodo de un grafo ponderado al resto de nodos. Puede ser usado en multitud de escenarios. Se aplica exitosamente en algoritmos de enrutamiento de redes y telefonía, aplicaciones de cálculo de rutas, etc.

En el algoritmo de Dijkstra se va etiquetando cada nodo con la tupla [distancia_acumulada, padre] donde la distancia_acumulada es la distancia mínima desde el nodo inicial y padre es el nodo predecesor en el camino mínimo que une el nodo inicial con el que se está calculando. La secuencia de pasos para ir etiquetando todos los nodos es:

1. Seleccionamos el nodo no visitado con menor distancia acumulada.

2. Sumamos la distancia acumulada con la distancia a los nodos adyacentes y los etiquetamos con [distancia_acumulada, padre]. En caso de que alguno de los nodos adyacentes esté ya etiquetado, nos quedamos con el de menor distancia acumulada.

3. Marcamos el nodo actual como visitado y regresamos al paso 1

Como ejemplo supongamos el siguiente grado ponderado, que puede representar una serie de ciudades con su distancia, los nodos de una red o cualquier cosa que pueda modelarse con un grafo.

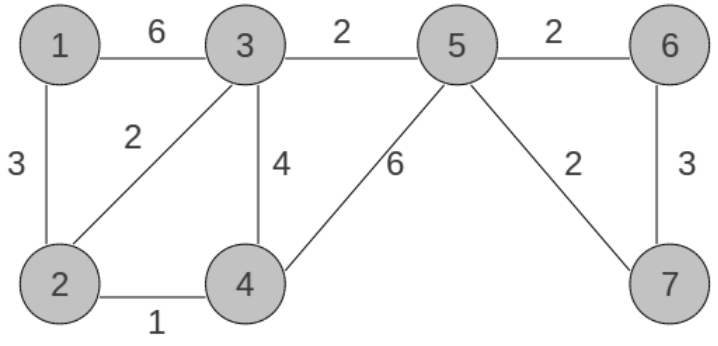


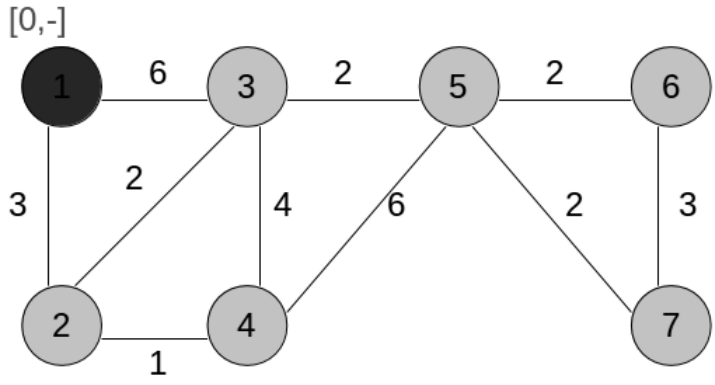
Fig. 4-4 Grafo de ejemplo para el algoritmo de Dijkstra

Queremos conocer la distancia mínima desde el nodo 1 al resto de nodos. Si cada nodo representa una ciudad, cada arista una conexión por carretera y el valor de la

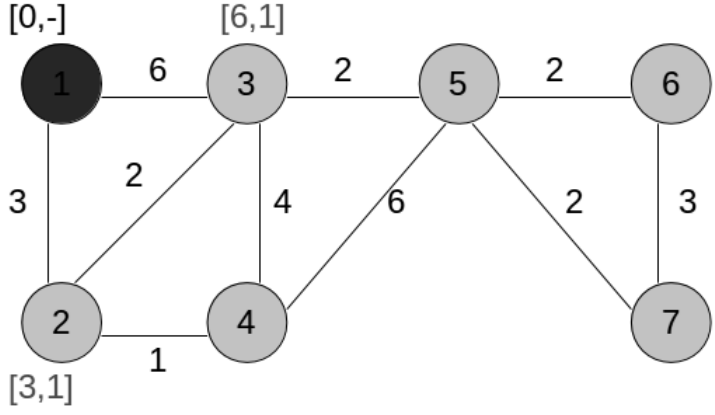
© RC Libros

arista representa los kilómetros de distancia que separa cada ciudad, el algoritmo de Dijkstra nos devolverá la distancia mínima desde la ciudad 1 al resto de ciudades. Veamos paso a paso cómo se aplica el algoritmo.

Paso 1. Elegimos el primer nodo (el nodo 1) y lo marcamos con distancia 0 y sin padre.

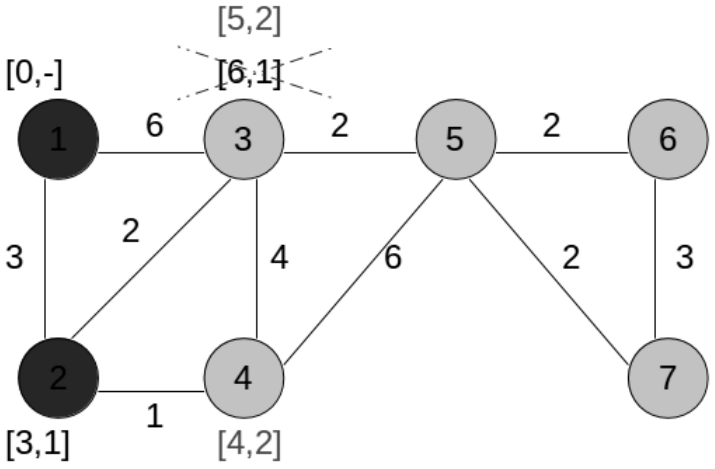


Paso 2. Etiquetamos los dos nodos adyacentes con la distancia y el nodo predecesor. También marcamos el nodo 1 como visitado (en rojo).

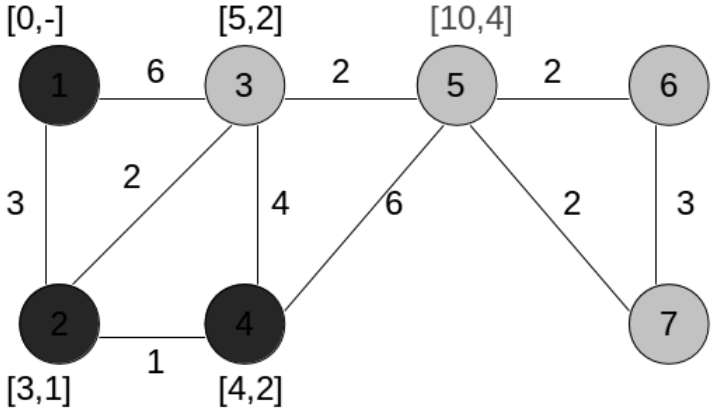


© RC Libros

Paso 3. Seleccionamos el nodo con menor distancia acumulada. En este caso el 2 que tiene peso acumulado 3. Seleccionamos los nodos adyacentes y los etiquetamos con el peso acumulado (peso del nodo actual más el peso de la arista del nodo adyacente). Como padre ahora pondremos el nodo 2. Como vemos el nodo 3 tiene ahora dos etiquetas, así que nos quedamos con la de menor peso. Marcamos el nodo 2 como visitado.

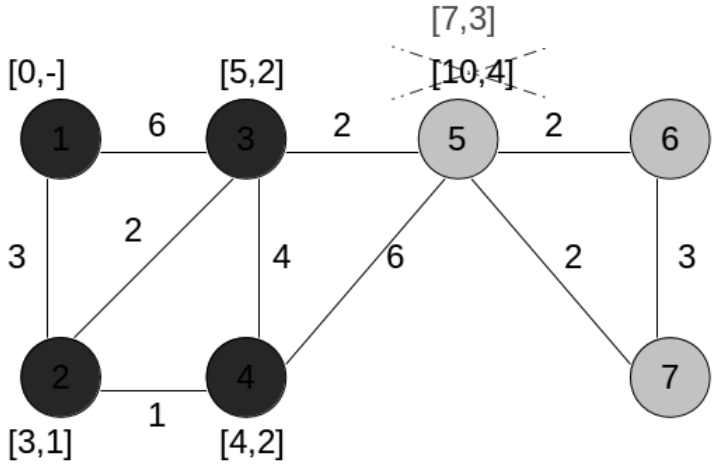


Paso 4. Seleccionamos el nodo 4 que es el de menor peso acumulado. En este caso ya no tenemos en cuenta el nodo 2 porque ya está marcado como visitado ni el nodo 3 porque el peso acumulado que tiene marcado es menor que el nuevo, así que etiquetamos el nodo 5 y marcamos el 4 como visitado.

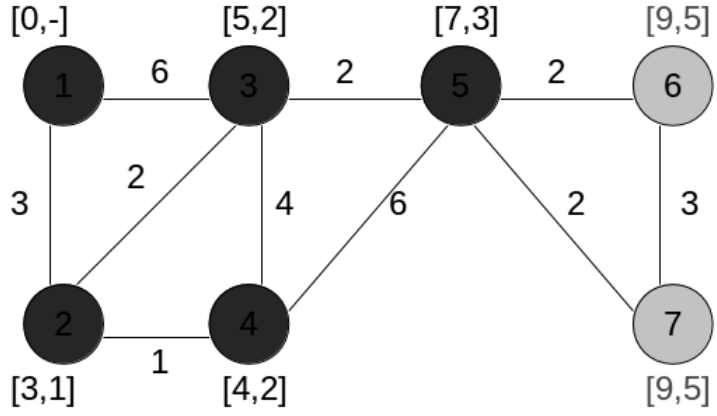


© RC Libros

Paso 5. El siguiente nodo de menor peso acumulado es el 3. Como el 4 ya está visitado solo nos queda por etiquetar el 5. En este caso, como el nuevo peso acumulado es menor reetiquetamos con el nuevo peso y el nuevo padre. Marcamos el nodo 3 como visitado.

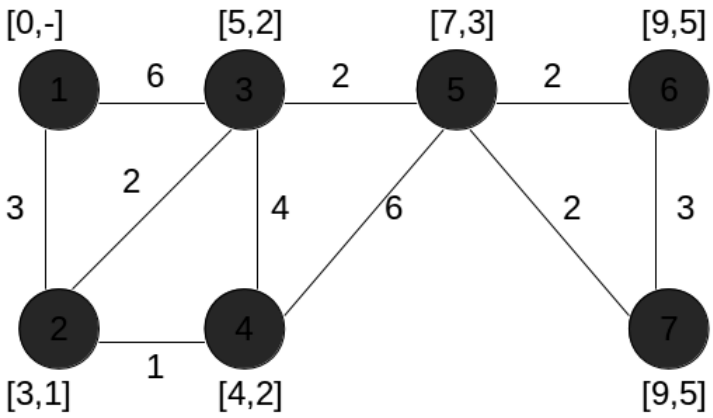


Paso 6. Seleccionamos el nodo 5 y etiquetamos los dos nodos adyacentes no visitados. Seguidamente lo marcamos como visitado.



Paso 7. Finalmente seleccionamos el 6 y como no mejora el peso acumulado hasta el nodo 7 no cambiamos la etiqueta. Lo mismo ocurre con el 7, así que marcamos ambos como visitados y obtenemos finalmente los siguientes pesos y caminos.

© RC Libros



Ahora podemos responder cualquier pregunta que nos hagan sobre el grafo. Por ejemplo:

¿Cuál es la distancia (peso) mínima desde el nodo 1 al nodo 7? Directamente podemos contestar que es 9, ya que es su peso acumulado.

¿Cuál es el camino mínimo del nodo 1 al 7? Solo hay que mirar quién es el padre del nodo 7 e ir recorriendo el camino hacia atrás viendo quién es el padre de cada nodo hasta llegar al 1. En este caso, el camino mínimo es 1-2-3-5-7.

El siguiente código hace una implementación del algoritmo de Dijkstra.

```
# Dijkstra
import sys

def dijkstra(grafo, nodo_inicial):
    etiquetas = {}
    visitados = []
    pendientes = [nodo_inicial]
    nodo_actual = nodo_inicial

    # nodo inicial
    etiquetas[nodo_actual] = [0,'']
```


© RC Libros

```
    # seleccionar el siguiente nodo de menor peso acumulado
    while len(pendientes) > 0:
        nodo_actual = nodo_menor_peso(etiquetas, visitados)
        visitados.append(nodo_actual)

        # obtener nodos adyacentes
        for adyacente, peso in grafo[nodo_actual].iteritems():
            if adyacente not in pendientes and \
            adyacente not in visitados:
                pendientes.append(adyacente)
            nuevo_peso = etiquetas[nodo_actual][0] \
            + grafo[nodo_actual][adyacente]
            # etiquetar
            if adyacente not in visitados:
                if adyacente not in etiquetas:
                    etiquetas[adyacente] = [nuevo_peso, nodo_actual]
                else:
                    if etiquetas[adyacente][0] > nuevo_peso:
                        etiquetas[adyacente] = \
                        [nuevo_peso, nodo_actual]

        del pendientes[pendientes.index(nodo_actual)]

    return etiquetas


def nodo_menor_peso(etiquetas, visitados):
    menor = sys.maxint
    for nodo, etiqueta in etiquetas.iteritems():
        if etiqueta[0] < menor and nodo not in visitados:
            menor = etiqueta[0]
            nodo_menor = nodo
```

© RC Libros

```
        return nodo_menor


    if __name__ == "__main__":
        grafo = {
            '1': {'3':6, '2':3},
            '2': {'4':1, '1':3, '3':2},
            '3': {'1':6, '2':2, '4':4, '5':2},
            '4': {'2':1, '3':4, '5':6},
            '5': {'3':2, '4':6, '6':2, '7':2},
            '6': {'5':2, '7':3},
            '7': {'5':2, '6':3}}

        etiquetas = dijkstra(grafo, '1')
        print etiquetas
```

Listado. 4-6 dijkstra.py

La función *dijkstra()* recibe dos parámetros. El grafo modelado en forma de diccionario de python y la etiqueta del nodo inicial. Como resultado nos devuelve un diccionario con cada nodo y su etiqueta. En nuestro ejemplo se ha obtenido el siguiente resultado.

{'1': [0, ''], '3': [5, '2'], '2': [3, '1'], '5': [7, '3'], '4': [4, '2'], '7': [9, '5'], '6': [9, '5']}

Según esto vemos, por ejemplo, que el peso acumulado (distancia) desde el nodo 1 al nodo 7 es 9 y su nodo padre es el 5.

ALGORITMO DE CLARKE Y WRIGHT

En el capítulo 2 introdujimos el problema VRP y analizamos algunos tipos de restricciones que podían condicionar el problema. Uno de los algoritmos más conocidos para resolver el problema VRP es el algoritmo de **Clarke y Wright**, también conocido como **algoritmo de los ahorros** (**Savings Algorithm**). Este algoritmo usa una heurística, que analizaremos seguidamente, para ir construyendo sistemáticamente

© RC Libros

las rutas de cada vehículo. Evidentemente, hay que adaptar el algoritmo a las restricciones impuestas por el problema que queremos resolver: ¿Tenemos un número limitado de vehículos? ¿Hay un límite de kilómetros para los vehículos? ¿Hay que cumplir horarios de reparto? Las restricciones pueden ser innumerables.

Para atacar el problema nosotros partiremos de los siguientes supuestos:

- Disponemos de un único almacén desde donde parten y a cuál regresan todos los vehículos tras realizar el reparto.
- Por simplicidad, vamos a suponer que no tenemos limitación en el número de vehículos disponibles.
- Tampoco hay límite de kilometraje para los vehículos.
- Todos los vehículos tienen un límite de carga igual, que no puede sobrepasarse.
- Cada cliente hace un pedido con un peso concreto y pudiendo ser diferente del resto.

Para mantener la generalidad, no introduciremos más restricciones espaciales ni temporales. En cualquier caso, introducir otros tipos de restricciones no debe ser demasiado complicado una vez entendido el funcionamiento del algoritmo. Hay que hacer notar también que este algoritmo puede adaptarse perfectamente bien a la optimización de otros problemas similares, como el reparto de prensa, la recogida de basuras o la planificación del transporte público a la hora de crear líneas de autobús.

La heurística utilizada en el algoritmo de los ahorros se basa en el hecho de que dados dos clientes servidos por dos vehículos diferentes, si hacemos que un solo vehículo sirva a los dos obtendremos un ahorro tanto en distancia como en vehículos. Consideremos n clientes que son servidos por n vehículos, es decir, cada cliente es servido por un solo vehículo. En este caso, la distancia total recorrida por todos los vehículos es igual a la suma de las distancias desde el almacén hasta cada cliente y multiplicada por 2, ya que el vehículo debe ir y volver al almacén (suponiendo que el camino de ida tenga la misma distancia que el de vuelta). Matemáticamente lo expresamos como

$$Dist = 2 \times \sum_{i=1}^{n} d(almacen, cliente\ i)$$

Donde la función *d(almacen, cliente i)* representa la distancia entre el almacén y el i-ésimo cliente. Aunque es una solución válida, evidentemente esta solución es poco práctica. No es factible usar un vehículo para cada cliente. Si hacemos que uno de los vehículos sirva a dos clientes (llamémoslos cliente i y cliente j), habremos ahorrado de entrada un camión, pero también tendremos un ahorro en distancia recorrida.

Llamaremos a ese ahorro s, y podemos calcularlo con la siguiente fórmula (siendo A el almacén).

$$s(i,j) = 2d(A,i) + 2d(A,j) - [d(A,i) + d(i,j) + d(A,j)]$$

Simplificando nos queda

$$s(i,j) = d(A,i) + d(A,j) - d(i,j)$$

Podemos verlo de forma más intuitiva en la siguiente figura:

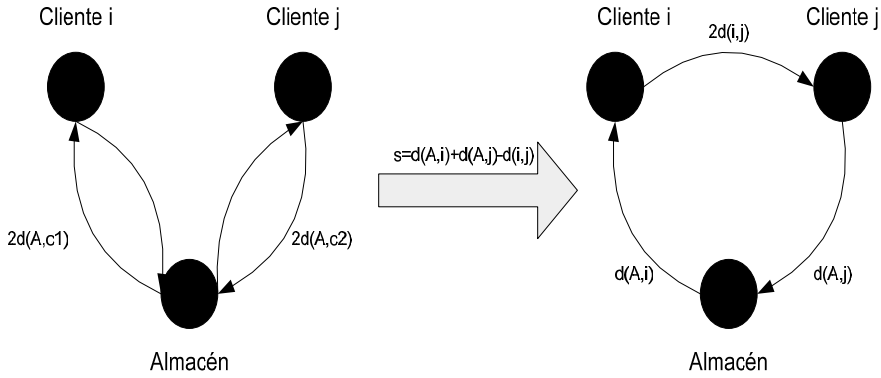


Fig. 4-5 Ahorro producido al eliminar un vehículo

Es decir, que s representa el ahorro de combinar a los clientes i y j en una sola ruta. El algoritmo de los ahorros trata de ir combinando en una misma ruta a los clientes, de forma que se produzca el mayor ahorro posible. Siempre y cuando no se viole ninguna restricción (por ejemplo, que el vehículo sobrepase su máximo de carga).

Pongamos por caso que se dan las siguientes distancias:

d(A,i) = 10

d(A,j) = 15

d(i,j) = 12

© RC Libros

El ahorro obtenido de unir a ambos clientes en una sola ruta es:

$$s(i,j) = d(A,i) + d(A,j) - d(i,j) = 10 + 15 - 12 = 13$$

Estos son los pasos que sigue el algoritmo de los ahorros para generar las rutas:

Paso 1. Calculamos el ahorro $s(i,j) = d(A,i) + d(A,j) - d(i,j)$ para cada par de clientes.

Paso 2. Ordenamos los ahorros de mayor a menor en una lista que llamaremos lista de ahorros.

Paso 3. Recorremos la lista de ahorros. Por cada ahorro en la lista, si no se viola ninguna restricción:

- Si ni i ni j están en ninguna ruta, creamos una ruta con i y j.
- Si i o j están en una ruta (pero no los dos) y además es el primero o el último cliente de la ruta (son exteriores), añadimos al cliente a la ruta de forma que i y j queden juntos.
- Si tanto i como j pertenecen a una ruta y ambos son exteriores en sus respectivas rutas, unimos ambas rutas en una de forma que i y j queden juntos.

Paso 4. Si quedan vehículos sin asignar a una ruta, crearemos una o varias (según las restricciones) que los incluya.

El siguiente programa implementa el algoritmo de los ahorros sobre la red de carreteras de ejemplo que hemos venido usando durante los capítulos anteriores. En un diccionario llamado *coord* introducimos las coordenadas de las ciudades. Las distancias entre ciudades la calculamos con la función *distancia().* Por simplificar, en esta ocasión usaremos las distancias planas en línea recta en vez de calcular la distancia real. Para calcular la distancia en línea recta que hay entre dos puntos del plano, desempolvamos a Pitágoras y usamos la fórmula:

$$d = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

Cada ciudad tiene una demanda n que se almacena en un diccionario llamado *pedidos*. Supondremos que la carga máxima de un vehículo es 40 y situaremos el almacén en las coordenadas (40.23, -3.40), en Madrid.

```
# VRP
import math
from operator import itemgetter

def distancia(coord1, coord2):
  lat1=coord1[0]
  lon1=coord1[1]
  lat2=coord2[0]
  lon2=coord2[1]
  return math.sqrt((lat1-lat2)**2+(lon1-lon2)**2)

def en_ruta(rutas, c):
  ruta = None
  for r in rutas:
    if c in r:
      ruta = r
  return ruta

def peso_ruta(ruta):
  total=0
  for c in ruta:
    total = total + pedidos[c]
  return total



def vrp_voraz():
  # calcular los ahorros
  s={}
  for c1 in coord:
    for c2 in coord:
      if c1 != c2:
        if not (c2,c1) in s:
          d_c1_c2 = distancia(coord[c1],coord[c2])
          d_c1_almacen = distancia(coord[c1], almacen)
```


© RC Libros

```
            d_c2_almacen = distancia(coord[c2], almacen)
            s[c1,c2]= d_c1_almacen + d_c2_almacen - d_c1_c2
    # ordenar ahorros
    s=sorted(s.items(), key=itemgetter(1), reverse=True)

    # construir rutas
    rutas=[]
    for k,v in s:
      rc1 = en_ruta(rutas, k[0])
      rc2 = en_ruta(rutas, k[1])
      if rc1==None and rc2==None:
        # no están en ninguna ruta. La creamos.
        if peso_ruta([k[0],k[1]]) <= max_carga:
          rutas.append([k[0],k[1]])
      elif rc1!=None and rc2==None:
        # ciudad 1 ya en una ruta. Agregamos la ciudad 2
        if rc1[0]==k[0]:
          if peso_ruta(rc1)+peso_ruta([k[1]]) <= max_carga:
            rutas[rutas.index(rc1)].insert(0, k[1])
        elif rc1[len(rc1)-1]==k[0]:
          if peso_ruta(rc1)+peso_ruta([k[1]]) <= max_carga:
            rutas[rutas.index(rc1)].append(k[1])
      elif rc1==None and rc2!=None:
        # ciudad 2 ya en una ruta. Agregamos la ciudad 1
        if rc2[0]==k[1]:
          if peso_ruta(rc2)+peso_ruta([k[0]]) <= max_carga:
            rutas[rutas.index(rc2)].insert(0, k[0])
        elif rc2[len(rc2)-1]==k[1]:
          if peso_ruta(rc2)+peso_ruta([k[0]]) <= max_carga:
            rutas[rutas.index(rc2)].append(k[0])
      elif rc1!=None and rc2!=None and rc1!=rc2:
        # ciudad 1 y 2 ya en una ruta. Tratamos de unirlas
        if rc1[0]==k[0] and rc2[len(rc2)-1]==k[1]:
          if peso_ruta(rc1)+peso_ruta(rc2) <= max_carga:
```

```
            rutas[rutas.index(rc2)].extend(rc1)
            rutas.remove(rc1)
        elif rc1[len(rc1)-1]==k[0] and rc2[0]==k[1]:
          if peso_ruta(rc1)+peso_ruta(rc2) <= max_carga:
            rutas[rutas.index(rc1)].extend(rc2)
            rutas.remove(rc2)

    return rutas


if __name__ == "__main__":
    coord = {
      'Malaga':(36.43, -4.24),
      'Sevilla':(37.23, -5.59),
      'Granada':(37.11, -3.35),
      'Valencia':(39.28, -0.22),
      'Madrid':(40.24, -3.41),
      'Salamanca':(40.57, -5.40),
      'Santiago':(42.52, -8.33),
      'Santander':(43.28, -3.48),
      'Zaragoza':(41.39, -0.52),
      'Barcelona':(41.23, +2.11)
      }

    pedidos = {
      'Malaga':10,
      'Sevilla':13,
      'Granada':7,
      'Valencia':11,
      'Madrid':15,
      'Salamanca':8,
      'Santiago':6,
      'Santander':7,
```


© RC Libros

```
    'Zaragoza':8,
    'Barcelona':14
  }

  almacen = (40.23, -3.40)
  max_carga = 40

  rutas = vrp_voraz()
  for ruta in rutas:
    print ruta
```

Listado. 4-7 vrpvoraz.py

Teniendo en cuenta la carga máxima de los vehículos y la carga a servir en cada ciudad, el programa nos propone las siguientes tres rutas, teniendo en cuenta que las tres rutas empiezan y terminan en el almacén, que está cerca de Madrid. Por lo que habría que añadirlo al principio y al final de cada ruta.

['Zaragoza', 'Barcelona', 'Valencia']

['Granada', 'Malaga', 'Sevilla']

['Santander', 'Santiago', 'Salamanca', 'Madrid']

© RC Libros

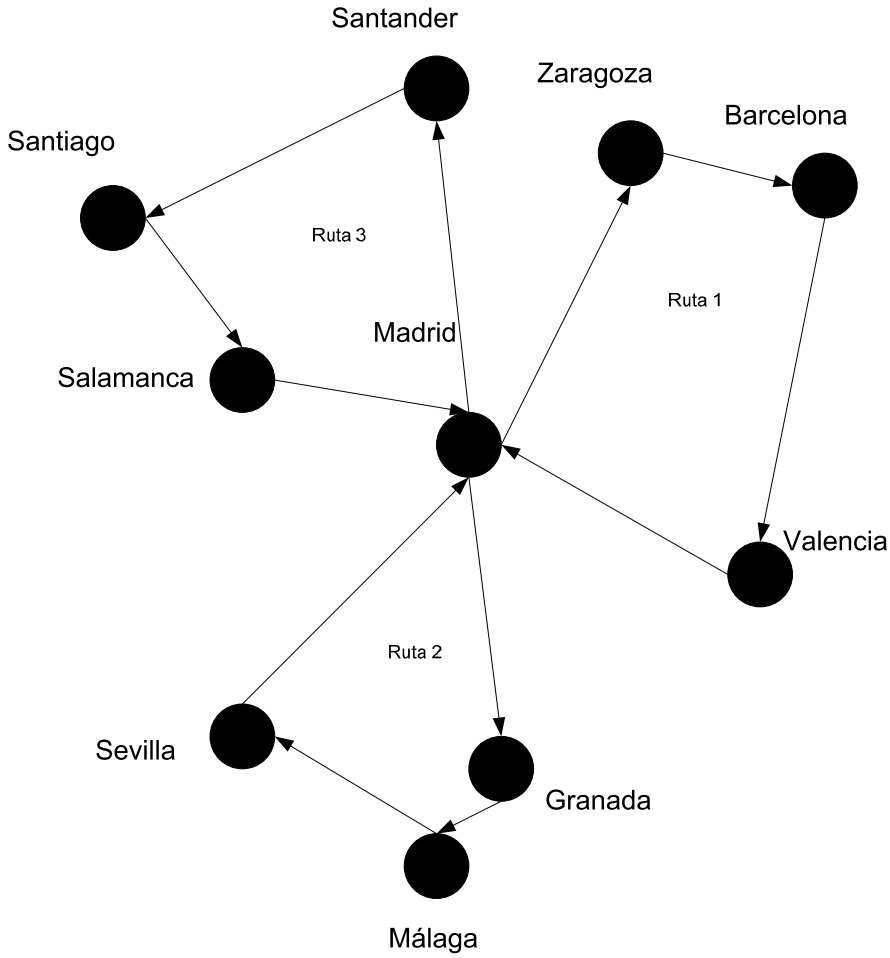


Fig. 4-6 Resultado de la ejecución del listado 4-7

Hill climbing

El algoritmo ***hill climbing***, también conocido como algoritmo de la **escalada de la colina** es un algoritmo de búsqueda local que, a partir de un estado inicial, trata de ir mejorando el resultado analizando los estados vecinos. Siempre que se encuentre un estado vecino mejor que el actual (según la función de evaluación que usemos), este estado sustituye al actual a partir de ese momento y se vuelve a repetir la operación con la vecindad del nuevo estado. El proceso termina cuando no es posible realizar más mejoras. También se suele poner un límite de iteraciones del algoritmo para evitar caer en larguísimos tiempos de proceso.

© RC Libros

El nombre del algoritmo proviene del hecho de que, si imaginamos el espacio de estados como una función similar a la de la figura 4-3, el algoritmo se moverá por el espacio de estados hacia la pendiente creciente, como si estuviera subiendo una colina (o hacia la pendiente descendente si lo que queremos es encontrar el mínimo de la función).

Cuando durante la ejecución del algoritmo no se puedan realizar más mejoras, será porque hemos alcanzado el óptimo global, que es lo deseable, o porque hemos alcanzado un máximo local. Como no es posible conocer el error cometido respecto al óptimo global, ya que no conocemos su valor, no es posible saber en ningún momento si hemos caído en un máximo local.

Tal y como vimos en la gráfica de la figura 4-3, el encontrar un máximo local o global depende únicamente de estado inicial. Otro problema es que no es posible dar un límite superior del tiempo de ejecución del algoritmo, aunque sí podemos limitar el número de iteraciones, en cuyo caso obtendremos una solución que podría estar relativamente cercana a la óptima o no.

A pesar de los problemas que pueda presentar, el algoritmo *hill climbing* tiene la ventaja de ser muy simple de implementar. Solo necesitamos una función de evaluación y un operador que nos permita generar los estados vecinos.

El pseudocódigo del algoritmo *hill climbing* es:

```
Función hill_climbing(estado_actual):
    Mejora=True
    Mientras mejora==True:
    Mejora=False
    Generar vecinos de estado_actual
    Por cada estado vecino:
      Si nuevo_estado mejora a estado_actual:
            Estado_actual=nuevo_estado
              Mejora=True

  Salir con estado_actual
```

Listado. 4-8 pseudocódigo de hill-climbing

© RC Libros

Como ejemplo trataremos de resolver el problema TSP sobre el mapa de ciudades que hemos venido usando hasta ahora. Como representación interna de la ruta usaremos una lista con las ciudades visitadas y con su orden de visita correspondiente. Por ejemplo, una ruta de 3 ciudades podría ser

Sevilla – Málaga – Granada

Como hemos dicho, necesitamos una función de evaluación. Parece obvio que la distancia total de la ruta (la suma de todas las distancias entre ciudades consecutivas) es una función de evaluación más que adecuada. Como operador para obtener los vecinos de una ruta definiremos un operador llamado **intercambio**, que intercambia dos ciudades de la ruta. Por ejemplo, las siguientes serían vecinas de la ruta de tres ciudades que hemos puesto de ejemplo.

Málaga – Sevilla – Granada

Granada – Málaga – Sevilla

Sevilla – Granada – Málaga

En el programa de ejemplo partiremos de una ruta aleatoria, pero muy bien podría ser una ruta construida con un algoritmo constructivo voraz que quisiéramos tratar de mejorar aún más.

```
# TSP con hill climbing
import math
import random

def distancia(coord1, coord2):
  lat1=coord1[0]
  lon1=coord1[1]
  lat2=coord2[0]
  lon2=coord2[1]
  return math.sqrt((lat1-lat2)**2+(lon1-lon2)**2)

# calcula la distancia cubierta por una ruta
def evalua_ruta(ruta):
  total=0
```


© RC Libros

```
    for i in range(0,len(ruta)-1):
      ciudad1=ruta[i]
      ciudad2=ruta[i+1]
      total=total+distancia(coord[ciudad1], coord[ciudad2])
    ciudad1=ruta[i+1]
    ciudad2=ruta[0]
    total=total+distancia(coord[ciudad1], coord[ciudad2])
    return total


  def hill_climbing():
    # crear ruta inicial aleatoria
    ruta=[]
    for ciudad in coord:
      ruta.append(ciudad)
    random.shuffle(ruta)

    mejora=True
    while mejora:
      mejora=False
      dist_actual=evalua_ruta(ruta)
      # evaluar vecinos
      for i in range(0,len(ruta)):
        if mejora:
          break
        for j in range(0,len(ruta)):
          if i!=j:
            ruta_tmp=ruta[:]
            ciudad_tmp=ruta_tmp[i]
            ruta_tmp[i]=ruta_tmp[j]
            ruta_tmp[j]=ciudad_tmp
            dist=evalua_ruta(ruta_tmp)
            if dist<dist_actual:
              # encontrado vecino que mejora el resultado
```

```
                mejora=True
                ruta=ruta_tmp[:]
                break

    return ruta


if __name__ == "__main__":
    coord = {
      'Malaga':(36.43, -4.24),
      'Sevilla':(37.23, -5.59),
      'Granada':(37.11, -3.35),
      'Valencia':(39.28, -0.22),
      'Madrid':(40.24, -3.41),
      'Salamanca':(40.57, -5.40),
      'Santiago':(42.52, -8.33),
      'Santander':(43.28, -3.48),
      'Zaragoza':(41.39, -0.52),
      'Barcelona':(41.23, +2.11)
      }

    ruta = hill_climbing()
    print ruta
    print "Distancia total: " + str(evalua_ruta(ruta))
```

Listado. 4-9 hill_climbing.py

La ejecución del programa nos propone la siguiente solución al problema del TSP:

['Madrid', 'Sevilla', 'Malaga', 'Granada', 'Valencia', 'Barcelona', 'Zaragoza', 'Santander', 'Santiago', 'Salamanca']

Distancia total: 29.8455018341

© RC Libros

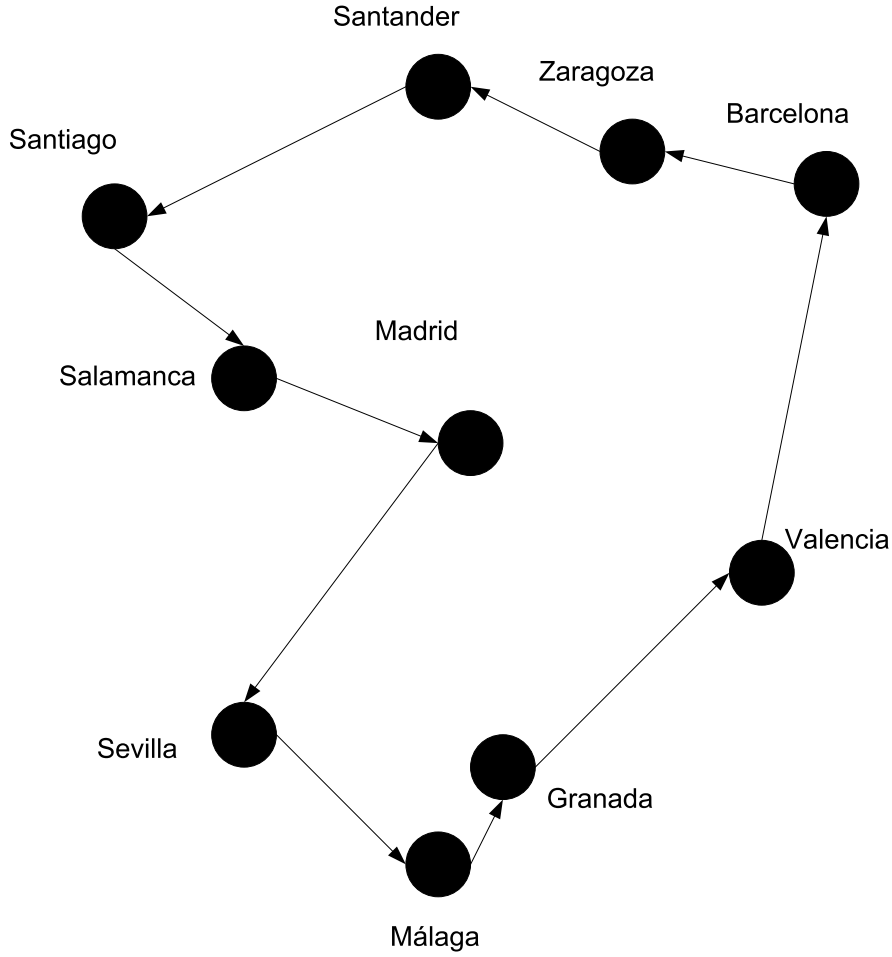


Fig. 4-7 Resultado de la ejecución del listado 4-9

Si realizamos varias ejecuciones, veremos cómo algunos resultados son ligeramente peores que otros. Por ejemplo, una de las ejecuciones nos ha devuelto el siguiente resultado.

['Malaga', 'Sevilla', 'Salamanca', 'Santiago', 'Santander', 'Zaragoza', 'Barcelona', 'Valencia', 'Madrid', 'Granada']

Distancia total: 30.110456262

Esto se debe, como ya comentamos antes, a la posibilidad que existe de caer en máximos locales que evitan que se llegue al máximo global. Recordemos que la

solución final obtenida depende únicamente del estado inicial, que en nuestro ejemplo es una ruta aleatoria.

Existe una variación del algoritmo de *hill climbing* que trata de paliar los problemas relativos a los óptimos locales. Se trata del ***hill climbing* iterativo**. La idea es ejecutar varias veces el algoritmo pero partiendo de un estado inicial diferente cada vez. El siguiente ejemplo realiza 10 ejecuciones partiendo de diferentes rutas aleatorias. Se ha añadido un bucle externo para realizar las 10 ejecuciones (variable *max_iteraciones*) y al final de cada ejecución se compara el resultado con la variable *mejor_ruta*, que almacena la mejor ruta encontrada hasta el momento. Si la ruta encontrada en una de las iteraciones es mejor que la almacenada en *mejor_ruta*, se sustituye por la nueva mejor ruta.

```
# TSP con hill climbing iterativo
import math
import random

def distancia(coord1, coord2):
  lat1=coord1[0]
  lon1=coord1[1]
  lat2=coord2[0]
  lon2=coord2[1]
  return math.sqrt((lat1-lat2)**2+(lon1-lon2)**2)

# calcula la distancia cubierta por una ruta
def evalua_ruta(ruta):
  total=0
  for i in range(0,len(ruta)-1):
    ciudad1=ruta[i]
    ciudad2=ruta[i+1]
    total=total+distancia(coord[ciudad1], coord[ciudad2])
  ciudad1=ruta[i+1]
  ciudad2=ruta[0]
  total=total+distancia(coord[ciudad1], coord[ciudad2])
  return total
```


© RC Libros

```
def i_hill_climbing():
  # crear ruta inicial aleatoria
  ruta=[]
  for ciudad in coord:
    ruta.append(ciudad)
  mejor_ruta = ruta[:]
  max_iteraciones=10

  while max_iteraciones>0:
    mejora=True
    # generamos una nueva ruta aleatoria
    random.shuffle(ruta)
    while mejora:
      mejora=False
      dist_actual=evalua_ruta(ruta)
      # evaluar vecinos
      for i in range(0,len(ruta)):
        if mejora:
          break
        for j in range(0,len(ruta)):
          if i!=j:
            ruta_tmp=ruta[:]
            ciudad_tmp=ruta_tmp[i]
            ruta_tmp[i]=ruta_tmp[j]
            ruta_tmp[j]=ciudad_tmp
            dist=evalua_ruta(ruta_tmp)
            if dist<dist_actual:
              # encontrado vecino que mejora el resultado
              mejora=True
              ruta=ruta_tmp[:]
              break
    max_iteraciones=max_iteraciones-1
```

```
      if evalua_ruta(ruta)<evalua_ruta(mejor_ruta):
        mejor_ruta=ruta[:]

    return mejor_ruta


if __name__ == "__main__":
    coord = {
      'Malaga':(36.43, -4.24),
      'Sevilla':(37.23, -5.59),
      'Granada':(37.11, -3.35),
      'Valencia':(39.28, -0.22),
      'Madrid':(40.24, -3.41),
      'Salamanca':(40.57, -5.40),
      'Santiago':(42.52, -8.33),
      'Santander':(43.28, -3.48),
      'Zaragoza':(41.39, -0.52),
      'Barcelona':(41.23, +2.11)
      }

    ruta = i_hill_climbing()
    print ruta
    print "Distancia total: " + str(evalua_ruta(ruta))
```

Listado. 4-10 it_hill_climbing.py

Al ejecutar varias veces esta nueva versión, podremos observar que el resultado es mucho más coherente entre ejecuciones (con suerte él mismo). Aun así nada nos asegura que hayamos alcanzado el máximo global.

© RC Libros

Simulated annealing

Como hemos visto en la sección anterior, el mayor peligro con el que nos enfrentamos en la búsqueda local es caer en un máximo local. Las tres técnicas que vamos a presentar en lo que queda de capítulo van enfocadas a intentar escapar de los máximos locales. Son **metaheurísticas** (métodos heurísticos genéricos para problemas que no disponen de un algoritmo propio que los resuelva) parametrizables que nos permiten atacar un gran espectro de problemas diferentes. Las tres técnicas que vamos a analizar son el **templado simulado** o ***simulated annealing***, la **búsqueda tabú** y **los algoritmos genéticos o evolutivos**.

La técnica del templado simulado tiene su origen en el proceso físico de templado del acero, el cristal o la cerámica. La técnica del templado consiste en elevar mucho la temperatura del material y luego enfriarlo lentamente para alterar sus propiedades físicas y hacerlos más resistentes.

El templado simulado se parece mucho a la técnica de *hill climbing* pero añade algunas variaciones. La primera diferencia es que en cada iteración no siempre elegirá un vecino que mejore la solución actual, sino que puede seleccionar una solución peor. La aceptación o no de una solución peor depende de una función probabilística. De esta forma no siempre escogeremos el camino de mayor pendiente como hacíamos con *hill climbing*, sino que añadimos un elemento de variación que nos permite escapar de los máximos locales.

Durante el proceso, en lugar de evaluar todos los vecinos del estado actual evaluaremos solo algunos. Si el nuevo estado mejora al actual haremos el cambio tal y como sucedía con *hill climbing*, pero si no, usaremos la función de probabilidad que decidirá si hacemos o no el cambio. ¿Qué función de probabilidad debemos usar?

Básicamente, la probabilidad de hacer el cambio dependerá de dos factores. El primer factor es la diferencia relativa que hay entre el estado actual y el nuevo. Si la nueva solución es considerablemente peor que la actual tendrá menos probabilidad de realizarse el cambio. El segundo factor lo llamaremos temperatura y lo denotaremos como T. La función de probabilidad que vamos a usar es la siguiente:

$$p = e^{\frac{eval(E_{actual}) - eval(E_{nuevo})}{T}}$$

Siendo E_{actual} el estado actual y E_{nuevo} el nuevo estado vecino. La función *eval()* es la función de evaluación que estamos usando para medir la calidad de la solución. Por ejemplo, la distancia de la ruta en el caso del TSP. Si lo que buscamos es el estado mínimo, cambiaremos el minuendo y el sustraendo del numerador del exponente.

El rol que realiza la operación (E_{actual} - E_{nuevo}) es claro: la probabilidad de cambio depende de la medida en que una solución es peor que la otra (diferencia de costes), pero ¿qué misión tiene T (temperatura)? Veamos en la siguiente tabla qué ocurre al ir incrementando el valor de T.

T	$p = e^{\frac{-10}{T}}$
1	0.00004
5	0.13533
10	0.36787

Observamos que a mayor valor de T, mayor es el valor de la función de probabilidad. Esto quiere decir que un valor alto de la temperatura favorecerá que se produzca la selección de un estado peor que el actual.

La técnica se llama templado simulado porque consiste en comenzar con un valor alto de temperatura T e ir disminuyéndolo poco a poco en cada iteración, de forma que en los primeros momentos sea fácil que se seleccionen soluciones peores, y con el paso del tiempo, según se enfría T, ir estabilizándose en la mejor solución encontrada, ya que la probabilidad de seleccionar un vecino peor decrece.

El pseudocódigo del algoritmo es el siguiente:

```
Función simulated_annealing(estado_actual):
 inicializar T
 inicializar T_MIN
 inicializar v_enfriamiento

 Mientras T < T_MIN:
   Para i=0 hasta v_enfriamiento:
     estado_nuevo=elegir un estado vecino
     Si eval(estado_actual)<eval(estado_nuevo):
```


© RC Libros

```
          estado_actual=estado_nuevo
        Sino:
          Si random(0,1)< e^((eval(E_actual)-eval(E_nuevo))/T)
            estado_actual=estado_nuevo

      Enfriar(T)

    Salir con estado_actual
```

Listado. 4-11 pseudocódigo para el templado simulado

Fijémonos en las tres variables que se inicializan al principio de la función. La variable T (temperatura) ya la conocemos, ¿pero qué valor inicial hay que darle? Un valor muy alto haría que se tardara mucho en converger hacia una solución estable, mientras que un valor bajo no permitiría que la ejecución tuviera tiempo de alcanzar un estado estable aceptable.

La variable T_MIN indica el valor inferior que se alcanzará en el proceso de enfriamiento y, por lo tanto, cuando la temperatura alcance el valor T_MIN habremos terminado (es la condición de parada). En la práctica es habitual que el algoritmo termine al llegar a un número máximo de iteraciones en lugar de alcanzar el valor T_MIN.

Finalmente, la variable v_enfriamiento (velocidad de enfriamiento) indica cuántos vecinos se evalúan en cada iteración antes de proceder a llamar a la función *enfriar()*. La velocidad de enfriamiento ha de ser lo suficientemente grande para permitir que se alcance un estado estacionario para la temperatura de ese momento.

En general, la elección de los valores de estos tres parámetros dependerá en gran medida del problema concreto y del tamaño de la vecindad de las soluciones.

Ya hemos comentado que la función *enfriar()* va disminuyendo el valor de la temperatura T en cada iteración, pero ¿con qué velocidad hay que enfriar T? Existen diversos métodos, y la elección de uno u otro dependerá, de nuevo, del problema sobre el que se esté trabajando. La forma más intuitiva de enfriar T es ir disminuyéndola linealmente, es decir, ir restando una cantidad fija en cada vuelta del bucle. Otras funciones que pueden ser utilizadas para el enfriamiento son:

Descenso exponencial	$T_{i+1} = \delta \times T_i \quad (\delta \in [0.8, 0.9])$
Criterio de Boltzmann	$T_i = \frac{T_0}{1 + \log(i)}$
Criterio de Cauchy	$T_i = \frac{T_0}{1 + i}$

El siguiente programa resuelve el problema TSP utilizando la técnica del templado simulado.

```
# TSP con templado simulado
import math
import random

def distancia(coord1, coord2):
  lat1=coord1[0]
  lon1=coord1[1]
  lat2=coord2[0]
  lon2=coord2[1]
  return math.sqrt((lat1-lat2)**2+(lon1-lon2)**2)

# calcula la distancia cubierta por una ruta
def evalua_ruta(ruta):
  total=0
  for i in range(0,len(ruta)-1):
    ciudad1=ruta[i]
    ciudad2=ruta[i+1]
    total=total+distancia(coord[ciudad1], coord[ciudad2])
  ciudad1=ruta[i+1]
  ciudad2=ruta[0]
  total=total+distancia(coord[ciudad1], coord[ciudad2])
  return total
```


© RC Libros

```
def simulated_annealing(ruta):
  T=20
  T_MIN=0
  v_enfriamiento=100

  while T>T_MIN:
    dist_actual=evalua_ruta(ruta)
    for i in range(1, v_enfriamiento):
      # intercambiamos dos ciudades aleatoriamente
      i=random.randint(0, len(ruta)-1)
      j=random.randint(0, len(ruta)-1)
      ruta_tmp=ruta[:]
      ciudad_tmp=ruta_tmp[i]
      ruta_tmp[i]=ruta_tmp[j]
      ruta_tmp[j]=ciudad_tmp
      dist=evalua_ruta(ruta_tmp)
      delta=dist_actual-dist
      if (dist<dist_actual):
        ruta=ruta_tmp[:]
        break
      elif random.random() < math.exp(delta/T):
        ruta=ruta_tmp[:]
        break

    # enfriamos T linealmente
    T=T-0.005

  return ruta


if __name__ == "__main__":
  coord = {
    'Malaga':(36.43, -4.24),
```

```
      'Sevilla':(37.23, -5.59),
      'Granada':(37.11, -3.35),
      'Valencia':(39.28, -0.22),
      'Madrid':(40.24, -3.41),
      'Salamanca':(40.57, -5.40),
      'Santiago':(42.52, -8.33),
      'Santander':(43.28, -3.48),
      'Zaragoza':(41.39, -0.52),
      'Barcelona':(41.23, +2.11)
      }

    # crear ruta inicial aleatoria
    ruta=[]
    for ciudad in coord:
      ruta.append(ciudad)
    random.shuffle(ruta)

    ruta = simulated_annealing(ruta)
    print ruta
    print "Distancia total: " + str(evalua_ruta(ruta))
```

Listado. 4-12 sannealing.py

Como ejemplo de ejecución, el programa nos ofrece la siguiente solución:

['Sevilla', 'Malaga', 'Granada', 'Valencia', 'Barcelona', 'Zaragoza', 'Santander', 'Santiago', 'Salamanca', 'Madrid']

Distancia total: 29.8455018341

© RC Libros

Búsqueda tabú

La **búsqueda tabú**, al igual que la técnica del templado simulado, es un procedimiento metaheurístico que trata de diversificar la exploración en el espacio de estados para evitar caer en óptimos locales. Puede ocurrir que zonas menos prometedoras a priori nos conduzcan al óptimo global. La búsqueda tabú utiliza una memoria adaptativa que va variando durante la ejecución del algoritmo y que le permite "recordar" los caminos visitados recientemente. A estos caminos los llamaremos tabú. En cada iteración del algoritmo evitaremos explorar los caminos marcados como tabú y de esta forma conseguimos explorar otros que, a priori, nunca elegiríamos como prometedores. Es evidente que el algoritmo no sería muy útil si marcáramos un camino como tabú y ya nunca más volvemos a explorar por ahí. Hemos dicho que la memoria adaptativa almacena los caminos tomados más recientemente, por lo que, pasado un tiempo, aquellos caminos marcados como tabú deben dejar de serlo. Al número de iteraciones que permanecerá una variable siendo tabú lo llamamos **tiempo de persistencia**.

Comenzaremos analizando el problema SAT desde esta nueva perspectiva. Supongamos que nuestro problema SAT tiene 5 variables booleanas tales que

$$f = (v_1, v_2, v_3, v_4, v_5)$$

Pudiendo v_n tomar solamente los valores 0 y 1. Una posible asignación de valores para estas variables podría ser

$$f_1 = (1,1,0,1,0)$$

Recordemos que un vecino de V será cualquier otra asignación de valores para v_n tal que solo se diferencien en el valor de una variable. Por ejemplo:

$$f_2 = (1,1,0,1,1)$$

Es decir, que el operador que usaríamos para generar los estados vecinos en la búsqueda será simplemente cambiar el valor de una de las variables. En el ejemplo anterior hemos cambiado el valor de la quinta variable de f_1 con lo que hemos obtenido f_2. Evidentemente, la elección del mejor vecino vendrá dada por la función de evaluación que estemos utilizando.

Dado este escenario, una búsqueda tabú evitaría, durante algunas iteraciones, volver a cambiar el valor de la quinta variable. Necesitamos, pues, una estructura de datos para almacenar qué variables que se han cambiado recientemente y, además, durante cuánto tiempo ha de seguir siendo un cambio tabú.

© RC Libros

Vamos a suponer que cuando marcamos una variable como tabú, estará prohibido volver a cambiarla durante las siguientes tres iteraciones. Además, para almacenar esta información vamos a utilizar un vector de enteros de longitud 5 (ya que hay 5 variables), que en un principio tendrá todos sus valores inicializados a cero. Si, por ejemplo, durante la búsqueda seleccionamos un vecino en el cual se modifica el valor de la quinta variable (como en el ejemplo anterior), anotaremos en el vector que durante las siguientes tres iteraciones, dicho cambio no estará disponible, y lo hacemos poniendo el valor 3 en la quinta posición del vector.

Si en las siguientes tres iteraciones, el mejor vecino vuelve a ser el resultante de cambiar la quinta variable, al comprobar que dicha posición del vector es distinta a cero, evitaremos hacer el cambio. Por supuesto, en cada iteración deberemos decrementar aquellas posiciones del vector con valores mayores a 0, de forma, que en nuestro ejemplo, al ocurrir 3 iteraciones, el valor de la quinta casilla volvería a ser 0 y por lo tanto estaría de nuevo disponible. Veamos un ejemplo concreto. Supongamos que partimos de

$$f_1 = (1,1,0,1,0)$$

Y nuestra memoria está inicializada a cero

0	0	0	0	0

Supongamos que en la primera iteración, la función de evaluación nos indica que el mejor vecino es el resultante de cambiar la quinta variable. En ese caso la seleccionamos como nuevo estado actual y marcamos como tabú la quinta variable durante las tres próximas iteraciones.

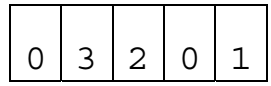
0	0	0	0	3

En la siguiente vuelta, el algoritmo decide cambiar la tercera variable, así que la marcamos, y además decrementamos aquellas variables mayores que cero.

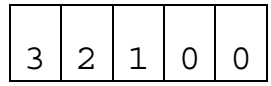
0	0	3	0	2

© RC Libros

En la siguiente iteración se decide al cambio de la segunda variable, quedando el vector de la siguiente manera.



En la siguiente vuelta, la función de evaluación nos indica que el mejor cambio es el de la quinta variable; sin embargo, al ser mayor que cero y estar marcado como tabú, no podremos hacerlo. Como este cambio es tabú se escoge el siguiente mejor, que en este caso podría ser el 1.



Finalmente, el algoritmo selecciona de nuevo la quinta variable para el cambio. Ahora su valor es 0, por lo que sí que podemos seleccionarla.

Puede ocurrir que algún estado vecino, incluso siendo tabú, nos ofrezca una solución excepcionalmente buena. En una situación como esta no conviene desperdiciar un estado tan prometedor, así que bajo circunstancias especiales nos permitiremos romper las reglas. Al criterio que nos permite decidir si rompemos o no las reglas lo lamamos **criterio de aspiración**. En la práctica, la búsqueda tabú recuerda siempre cuál es la mejor solución que ha encontrado desde el inicio de la búsqueda, y si el nuevo estado es mejor que el que está almacenado, lo selecciona aunque sea tabú.

El pseudocódigo para el algoritmo de búsqueda tabú es

```
Función busqueda_tabú(estado_actual):
  mejor_estado=estado_actual
  inicializar memoria_tabu
  inicializar persistencia
  inicializar numero_iteraciones



  Mientras numero_iteraciones>0:
    numero_iteraciones=numero_iteraciones-1
```

```
      val_actual=evaluar(estado_actual)
      para cada vecino:
        estado_nuevo=elegir un estado vecino
        si estado_nuevo no es tabú:
          si estado_nuevo mejora a estado_actual:
            estado_actual=estado_nuevo
            almacenar cambio en memoria tabú
            si estado_actual es mejor que mejor_estado:
              mejor_estado=estado_actual
            salir del bucle para
        si no:
          si estado_nuevo mejor que mejor_estado:
            estado_actual=estado_nuevo
            almacenar cambio en memoria tabú
            mejor_estado=estado_actual
            salir del bucle para

      decrementar persistencia de estados tabú

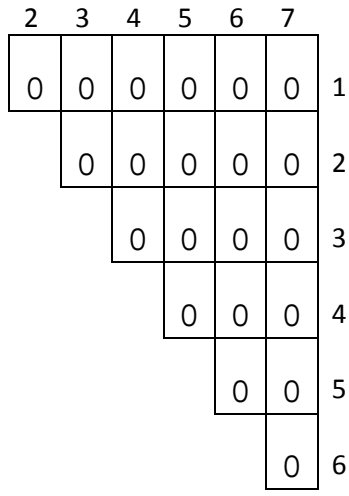
    salir con mejor_ruta
```

Listado. 4-13 pseudocódigo para la búsqueda tabú

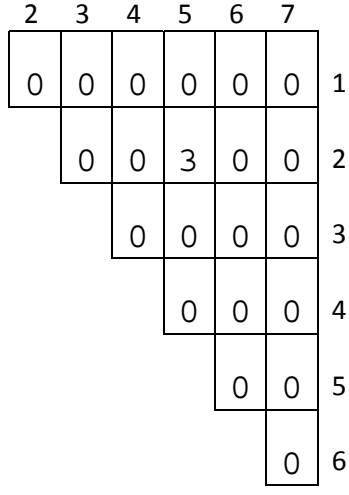
La idea básica de la búsqueda tabú no es compleja en sí misma, pero hay que adaptarla a cada problema concreto. Si queremos aplicarla al problema TSP, la estructura de datos que hemos usado con el problema SAT ya no nos vale.

Hemos venido utilizando como operador para generar las rutas vecinas en el problema TSP el intercambio de ciudades. ¿Qué estructura de datos podemos usar para recordar qué dos ciudades hemos intercambiado recientemente? En este caso podemos usar una estructura de datos como la siguiente.

© RC Libros



Los valores en la parte superior y en el lateral derecho representan ciudades. Los valores de las casillas tienen el mismo significado que el ejemplo del problema SAT que hemos visto antes. Es decir, que si hacemos un intercambio de las ciudades 2 y 5 y queremos que sea tabú para las próximas 3 iteraciones, la estructura de datos quedará así:



Consideraremos que, por ejemplo, el intercambio entre las ciudades 2 y 5 es el mismo que el intercambio entre las ciudades 5 y 2.

En el siguiente listado vamos a implementar el problema TSP usando búsqueda tabú. La estructura de datos presentada es conceptual, así que en vez de utilizarla tal

cual se ha descrito (podríamos haberlo hecho sin mayor problema), vamos a usar un diccionario de python, en el que la clave será la pareja de ciudades intercambiadas separadas por un guion bajo; y el valor, el número de iteraciones, que permanecerá siendo un movimiento tabú. Por ejemplo, si intercambiamos Granada y Madrid, y queremos que permanezca siendo un movimiento tabú durante las siguientes 5 iteraciones, almacenaremos la siguiente entrada en el diccionario:

memoria_tabu['Granada_Madrid']=5

Evidentemente, para comprobar si un movimiento es tabú o no hay que verificar si se encuentran en el diccionario las claves ['Granada_Madrid'] o ['Madrid_Granada'].

```
# TSP con búsqueda tabú
import math
import random

def distancia(coord1, coord2):
  lat1=coord1[0]
  lon1=coord1[1]
  lat2=coord2[0]
  lon2=coord2[1]
  return math.sqrt((lat1-lat2)**2+(lon1-lon2)**2)

# calcula la distancia cubierta por una ruta
def evalua_ruta(ruta):
  total=0
  for i in range(0,len(ruta)-1):
    ciudad1=ruta[i]
    ciudad2=ruta[i+1]
    total=total+distancia(coord[ciudad1], coord[ciudad2])
  ciudad1=ruta[i+1]
  ciudad2=ruta[0]
  total=total+distancia(coord[ciudad1], coord[ciudad2])
  return total
```


© RC Libros

```
def busqueda_tabu(ruta):
  mejor_ruta=ruta
  memoria_tabu={}
  persistencia=5
  mejora=False
  iteraciones=100

  while iteraciones>0:
    iteraciones = iteraciones-1
    dist_actual=evalua_ruta(ruta)
    # evaluar vecinos
    mejora=False
    for i in range(0,len(ruta)):
      if mejora:
        break
      for j in range(0,len(ruta)):
        if i!=j:
          ruta_tmp=ruta[:]
          ciudad_tmp=ruta_tmp[i]
          ruta_tmp[i]=ruta_tmp[j]
          ruta_tmp[j]=ciudad_tmp
          dist=evalua_ruta(ruta_tmp)

          # comprobar si el movimiento es tabú
          tabu=False
          if ruta_tmp[i]+"_"+ruta_tmp[j] in memoria_tabu:
            if memoria_tabu[ruta_tmp[i]+"_"+ruta_tmp[j]]>0:
              tabu=True
          if ruta_tmp[j]+"_"+ruta_tmp[i] in memoria_tabu:
            if memoria_tabu[ruta_tmp[j]+"_"+ruta_tmp[i]]>0:
              tabu=True

          if dist<dist_actual and not tabu:
```

```
            # encontrado vecino que mejora el resultado
            ruta=ruta_tmp[:]
            if evalua_ruta(ruta)<evalua_ruta(mejor_ruta):
              mejor_ruta=ruta[:]
            # almacenamos en memoria tabú
            memoria_tabu[ruta_tmp[i]+"_"+ruta_tmp[j]]=persistencia
            mejora=True
            break
          elif dist<dist_actual and tabu:
            # comprobamos criterio de aspiración
            # aunque sea movimiento tabú
            if evalua_ruta(ruta_tmp)<evalua_ruta(mejor_ruta):
              mejor_ruta=ruta_tmp[:]
              ruta=ruta_tmp[:]
              # almacenamos en memoria tabú
              memoria_tabu[ruta_tmp[i]+"_"+ruta_tmp[j]]=persistencia
              mejora=True
              break


    # rebajar persistencia de los movimientos tabú
    if len(memoria_tabu)>0:
      for k in memoria_tabu:
        if memoria_tabu[k]>0:
          memoria_tabu[k]=memoria_tabu[k]-1

  return mejor_ruta


if __name__ == "__main__":
  coord = {
    'Malaga':(36.43, -4.24),
    'Sevilla':(37.23, -5.59),
    'Granada':(37.11, -3.35),
```


© RC Libros

```
    'Valencia':(39.28, -0.22),
    'Madrid':(40.24, -3.41),
    'Salamanca':(40.57, -5.40),
    'Santiago':(42.52, -8.33),
    'Santander':(43.28, -3.48),
    'Zaragoza':(41.39, -0.52),
    'Barcelona':(41.23, +2.11)
    }

# crear ruta inicial aleatoria
ruta=[]
for ciudad in coord:
  ruta.append(ciudad)
random.shuffle(ruta)

ruta = busqueda_tabu(ruta)
print ruta
print "Distancia total: " + str(evalua_ruta(ruta))
```

Listado. 4-14 tabu.py

La siguiente ruta es el resultado de la ejecución del programa de ejemplo:

['Santander', 'Santiago', 'Salamanca', 'Madrid', 'Sevilla', 'Malaga', 'Granada', 'Valencia', 'Barcelona', 'Zaragoza']

Distancia total: 29.8455018341

El algoritmo puede mejorarse usando una segunda memoria que, además de guardar las últimas operaciones, guarde la frecuencia de estas, de forma que se favorezcan aquellos cambios que se han dado con menos frecuencia.

© RC Libros

Algoritmos genéticos

Los **algoritmos genéticos** o **algoritmos evolutivos** se inspiran en las teorías evolutivas de Darwin sobre selección natural. En grandes poblaciones de seres vivos, los individuos tienden a entrar en competencia por los recursos disponibles como alimento, agua o refugio. Los mejor adaptados y los más fuertes tenderán a vivir más tiempo y a dejar una mayor descendencia mientras que aquellos menos adaptados tendrán una descendencia menor. Con el tiempo, los genes de aquellos individuos mejor adaptados serán más abundantes y, por lo tanto, es de esperar una mejora global de la especie. Cuando dos individuos tienen descendencia, los genes de ambos se combinan pudiendo dar lugar a un individuo con unas características genéticas superiores a las de sus procreadores. Evidentemente, esto no siempre será así, ya que podríamos encontrar descendientes menos adaptados; sin embargo, es de esperar que a mejor calidad genética de los procreadores, mayor será la probabilidad de que las características genéticas del descendiente sean buenas.

Deducimos entonces que el mecanismo que permite la mejora de la especie es el intercambio de genes entre individuos; sin embargo, no es el único. De cuando en cuando surgen mutaciones accidentales en los genes. Estas mutaciones suelen ser negativas en la mayoría de los casos, pero de vez en cuando se da una mutación que mejora las características genéticas del individuo. Por lo tanto, tenemos dos mecanismos básicos de mejora de la especie: el entrecruzamiento de genes y la mutación de genes.

Los algoritmos genéticos, inspirándose en los mecanismos de la evolución natural, parten de una gran población de individuos (soluciones) y va generando descendencia basándose en las características de los mejores individuos, siempre según la función de evaluación. Las nuevas soluciones van sustituyendo a las antiguas de forma que, con el tiempo, la población tienda a converger hacia la solución global del problema.

Al igual que ocurre con los algoritmos de búsqueda local examinados en este capítulo, nada nos garantiza que acabemos encontrando la solución óptima, aunque si el algoritmo está bien diseñado y parametrizado, podemos tener cierto grado de confianza en que la solución alcanzada será, al menos, suficientemente buena.

Definiremos algunos conceptos antes de continuar.

- **Individuo**: Una solución válida al problema, independientemente de lo buena que sea. Debe cumplir con las restricciones impuestas por el problema.
- **Población**: Conjunto de individuos (soluciones).

© RC Libros

- **Función de adaptación**: Nos indica con qué grado se adapta el individuo a la solución. Es sinónimo de la función de evaluación que hemos venido usando hasta ahora.
- **Genes**: Conjunto de parámetros o características que describen al individuo (solución). Por ejemplo, en el problema TSP, una ruta válida se correspondería con un gen. En el caso del problema SAT, el conjunto de valores asignados a las variables sería un gen. Por ejemplo (1,0,0,1,1).
- **Cromosoma**: Cada uno de los parámetros o características que conforman un gen. Por ejemplo, si tenemos un gen que representa una ruta del problema TSP, cualquiera de las ciudades que conforman esa ruta sería un cromosoma.
- **Convergencia de la población**: No se trata de conseguir que una de las soluciones sea muy buena, sino que la población, en general, sea buena. Dicho de otro modo, si la población en general converge hacia unas características similares cuya función de adaptación es buena, esto nos servirá de piedra de toque para poder afirmar que convergemos hacia la solución óptima. Decimos que una población ha convergido si hay un gran porcentaje de individuos cuya función de adaptación es igual o muy similar. A ese porcentaje lo llamaremos **factor de convergencia** y lo denotaremos con la letra griega γ. Un valor habitual que nos permite afirmar que la población ha convergido suele ser γ = 95%. Es decir, que el 95% de la población tiene genes similares.

En general, un algoritmo genético está compuesto por cuatro fases bien diferenciadas.

- **Selección**: En esta fase elegimos las mejores soluciones (individuos) de toda la población. Dichas soluciones tendrán más probabilidad de dejar descendencia.
- **Cruce**: En esta fase se combinan los genes de los mejores individuos seleccionados en la fase anterior.
- **Mutación**: Con cierta probabilidad, se mutará un cromosoma de algunos de los individuos. Hay que hacer notar que la probabilidad de que se dé una mutación ha de mantenerse lo suficientemente baja como para que no se degrade la potencial mejora del cruce genético.
- **Eliminación**: Si no eliminamos individuos, la población crecerá indefinidamente hasta hacerse inmanejable. Es por ello que hay que eliminar a aquellos individuos (soluciones) peor adaptados.

En general, el pseudocódigo para un algoritmo genético es el siguiente:

```
Función algoritmo_genético():
  generar población inicial de forma aleatoria
  fin=falso

  Mientras no fin:
    Para tamaño(población)/2:
      seleccionar dos individuos
      cruzar los dos individuos
      mutar con cierta probabilidad los nuevos individuos
      añadir los dos nuevos individuos a la población
      eliminar los dos peores individuos de la población

    si la población converge:
      fin=verdadero

  salir con mejor solución
```

Listado. 4-15 pseudocódigo para el algoritmo genético básico

Alternativamente, puede usarse un límite en el número de iteraciones en vez de una condición de convergencia. O pueden usarse ambos criterios a la vez: el que se cumpla antes.

Hemos descrito someramente las cuatro fases del algoritmo genético. Ahora profundizaremos un poco más en cada una de ellas.

La fase de selección es la que nos permite escoger aquellas soluciones más cercanas a la solución óptima. El criterio que se sigue para puntuar las soluciones es la función de adaptación. Básicamente lo que hacemos es dar una puntuación a cada individuo basándonos en el valor devuelto por la función de adaptación y según sea dicho valor, el individuo tendrá más o menos probabilidades de ser seleccionado.

Una estrategia alternativa podría ser directamente la de escoger a los dos mejores individuos para cruzarlos, pero esta estrategia ha demostrado ser poco eficiente, ya que la falta de diversidad genética puede hacer que la evolución sea lenta e incluso que se detenga.

© RC Libros

Hay varias técnicas que nos permiten seleccionar un solo individuo de un conjunto con una probabilidad dada. Nosotros usaremos una técnica sencilla y muy intuitiva. Por ejemplo, supongamos que tenemos los siguientes individuos con sus respectivas puntuaciones:

Individuo	Puntuación
Individuo 1	10
Individuo 2	33
Individuo 3	5

Evidentemente, el individuo 2 tendrá mayor probabilidad de ser escogido que el individuo 1, y a su vez, este tendrá mayor probabilidad que el individuo 3 de ser seleccionado. La diferencia de probabilidad al elegir un individuo respecto a otro puede verse de forma más clara en la siguiente gráfica.

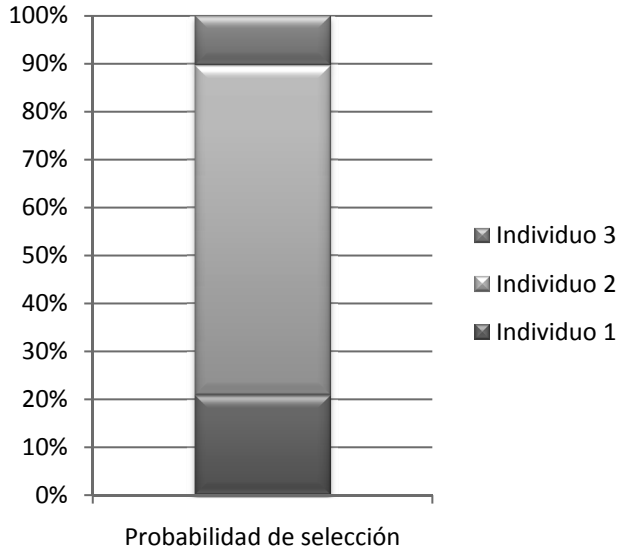


Fig. 4-8 Probabilidad de seleccionar a uno u otro individuo

Para hacer la selección según la probabilidad de cada individuo, calculamos la suma total de las puntuaciones según su función de adaptación. En este caso, la

suma total de las puntuaciones es 48. Seguidamente, obtendremos un número aleatorio comprendido entre 0 y 48. Supongamos que obtenemos el 25. Empezando por el primer individuo vamos a ir sumando todas las puntuaciones hasta que igualemos o sobrepasemos el valor 25. En este caso, primero sumamos 10, que corresponde a la puntuación del primer individuo. Como 10<25 seguimos con el siguiente. Sumamos 33, correspondiente a la puntuación del segundo individuo. Como 10+33 > 25, entonces ya podemos seleccionar al individuo 2.

En cada vuelta del algoritmo seleccionamos dos individuos para el cruce. Hay que hacer notar que si seleccionamos al primer individuo, este sigue pudiendo ser seleccionado de nuevo como segundo individuo; es decir, se puede dar el caso de que se seleccione un gen para cruzarse consigo mismo.

Una vez seleccionados dos individuos procedemos al cruce genético. Sin embargo, aún no hemos hablado de cómo podemos representar o codificar la información dentro de los genes para poder hacer el cruce. Es evidente que la representación de cada individuo está muy relacionada con el tipo de problema a solucionar. Tomemos como ejemplo el problema SAT. Si tenemos n variables booleanas que deben satisfacer una función, parece lógico que una buena representación puede ser un vector de ceros y unos. Por ejemplo, si quisiéramos satisfacer una función de 5 variables booleanas, un gen que podría representar una solución es el siguiente.

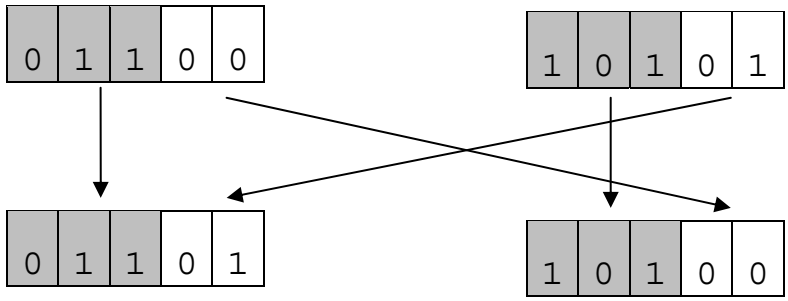
0	1	1	0	0

Supongamos que en la fase de selección hemos escogido el gen anterior y el siguiente.

1	0	1	0	1

El proceso de cruce consiste en elegir un punto de corte al azar. Por ejemplo supongamos que aleatoriamente elegimos el tercer cromosoma como punto de corte. Los dos nuevos genes resultantes del cruce serían los siguientes.

© RC Libros



Es decir, el primer gen hijo tomaría sus primeros cromosomas del primer gen padre y el resto del segundo gen padre. En el caso del segundo gen hijo, sus primeros cromosomas los tomaría heredados del segundo gen padre y los restantes del primer gen padre.

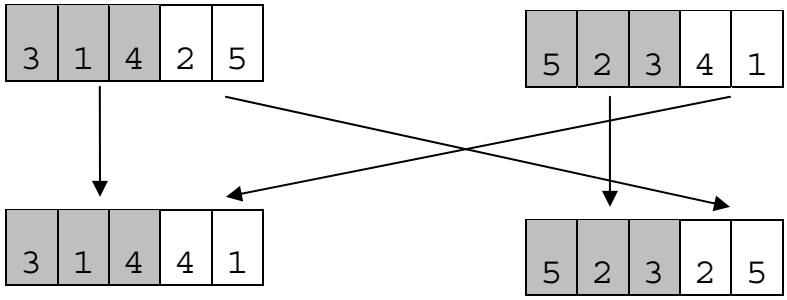
Esta representación en forma de vector de ceros y unos es muy usada en los algoritmos genéticos. Cualquier problema basado en si están presentes o no ciertas características de los individuos puede representarse de esta forma. También podemos afrontar problemas de optimización de funciones simplemente representando las variables con su valor binario en vez de en forma decimal. Esta es, por lo tanto, una codificación muy interesante siempre que podamos representar la información del problema de esta manera.

En cualquier caso, existen muchas más formas de codificar la información. Tantas como se puedan imaginar. Nada nos impide usar vectores con valores enteros o de valores alfanuméricos. Pongamos por caso el problema TSP. Una posible representación de una ruta de cinco ciudades puede ser la siguiente.

3	1	4	2	5

Donde a cada valor entero le hacemos corresponder una ciudad. Es decir, que este gen representa la ruta 3 – 1 – 4 – 2 – 5. Sin embargo, hay que tener muy en cuenta las restricciones propias del problema. Por ejemplo, supongamos el siguiente cruce de genes correspondientes al problema TSP.

© RC Libros



Los genes resultantes no son rutas válidas, ya que se repiten algunas ciudades y otras ni siquiera son visitadas. Es decir, que no se cumplen las restricciones del problema.

Diferentes operadores de cruce han sido descritos por diversos autores para este caso concreto. Uno de los más sencillos es el cruce basado en la alternancia de las posiciones. Este operador va tomando alternativamente cromosomas de ambos genes y colocándolos según su orden en el padre para crear el gen hijo. Se omiten los cromosomas que ya se encuentren en el gen hijo. Con los genes de arriba se obtendría el hijo

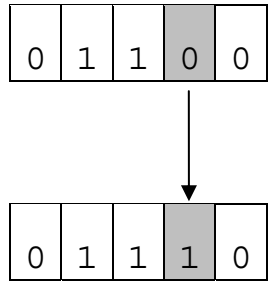
3	5	1	2	4

Y cambiando el orden de los padres obtendríamos el segundo hijo

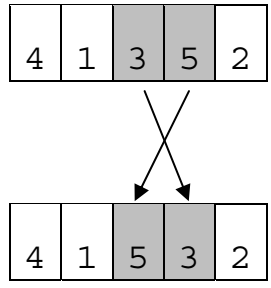
5	3	2	1	4

El cruce genético es un poderoso operador; sin embargo, puede ocurrir que la configuración de la población inicial impida o haga muy difícil que se llegue a explorar ciertos espacios de estados. Es por ello que se introduce el operador de mutación. Este operador introduce una mutación en un cromosoma con cierta probabilidad.

© RC Libros



De nuevo es importante tener en cuenta las restricciones del problema. En el caso del TSP, por ejemplo, un posible operador de mutación podría ser aquel que intercambia dos ciudades consecutivas. Aunque hay otros posibles.



En cualquier caso, no conviene darle mucho peso a este operador. Debe actuar con una baja probabilidad para no quitar protagonismo al operador de cruce, que es el que debe conducir la mejora y la evolución de la población.

Finalmente, la última fase del algoritmo es la de eliminación de individuos, que tiene dos finalidades: por un lado, evita que la población crezca y hace que se mantenga estable; por otro, elimina a los dos individuos peor adaptados evitando que sus cromosomas se perpetúen. Hay que tener en cuenta que los genes recién creados también entran en competencia a la hora de elegir qué individuos son eliminados; es decir, si se genera un hijo de muy baja calidad puede ser seleccionado para ser eliminado en la misma iteración.

© RC Libros

El siguiente programa resuelve el problema SAT (en concreto 3-SAT) usando un algoritmo genético. Por simplicidad no se usa un criterio de convergencia para detener el algoritmo, sino que indicamos un límite en el número de iteraciones en la variable *max_iter*. Como función de adaptación, en vez de evaluar una función concreta según los valores de las variables, se genera una solución aleatoria al principio del programa y la función *adaptación_3sat()* se encarga de comprobar el grado de adaptación de un gen a la solución generada aleatoriamente. Concretamente, la función de adaptación hace recuento del número de cláusulas (grupos de 3 variables) que toman el valor verdadero.

```
import math
import random

def poblacion_inicial(max_poblacion, num_vars):
  # crear población inicial aleatoria
  poblacion=[]
  for i in range(max_poblacion):
    gen=[]
    for j in range(num_vars):
      if random.random()>0.5:
        gen.append(1)
      else:
        gen.append(0)
    poblacion.append(gen[:])
  return poblacion

def adaptacion_3sat(gen, solucion):
  # contar cláusulas correctas
  n=3
  cont=0
  clausula_ok=True
  for i in range(len(gen)):
    n=n-1
    if (gen[i]!=solucion[i]):
      clausula_ok=False
```


© RC Libros

```
      if n==0:
        if clausula_ok:
          cont=cont+1
        n=3
        clausula_ok=True
    if n>0:
      if clausula_ok:
          cont=cont+1
    return cont


def evalua_poblacion(poblacion, solucion):
    # evalúa todos los genes de la población
    adaptacion=[]
    for i in range(len(poblacion)):
      adaptacion.append(adaptacion_3sat(poblacion[i], solucion))
    return adaptacion


def seleccion(poblacion, solucion):
    adaptacion=evalua_poblacion(poblacion, solucion)
    # suma de todas las puntuaciones
    total=0
    for i in range(len(adaptacion)):
      total=total+adaptacion[i]
    # escogemos dos elementos
    val1=random.randint(0,total)
    val2=random.randint(0,total)
    sum_sel=0
    for i in range(len(adaptacion)):
      sum_sel=sum_sel+adaptacion[i]
      if sum_sel>=val1:
        gen1=poblacion[i]
        break
```

```
    sum_sel=0
    for i in range(len(adaptacion)):
      sum_sel=sum_sel+adaptacion[i]
      if sum_sel>=val2:
        gen2=poblacion[i]
        break
    return gen1, gen2

  def cruce(gen1, gen2):
    # cruza 2 genes y obtiene 2 descendientes
    nuevo_gen1=[]
    nuevo_gen2=[]
    corte=random.randint(0, len(gen1))
    nuevo_gen1[0:corte]=gen1[0:corte]
    nuevo_gen1[corte:]=gen2[corte:]
    nuevo_gen2[0:corte]=gen2[0:corte]
    nuevo_gen2[corte:]=gen1[corte:]
    return nuevo_gen1, nuevo_gen2

  def mutacion(prob, gen):
    # muta un gen con una probabilidad prob.
    if random.random<prob:
      cromosoma=random.randint(0, len(gen))
      if gen[cromosoma]==0:
        gen[cromosoma]=1
      else:
        gen[cromosoma]=0
    return gen

  def elimina_peores_genes(poblacion, solucion):
    # elimina los dos peores genes
    adaptacion=evalua_poblacion(poblacion, solucion)
    i=adaptacion.index(min(adaptacion))
    del poblacion[i]
```


© RC Libros

```
    del adaptacion[i]
    i=adaptacion.index(min(adaptacion))
    del poblacion[i]
    del adaptacion[i]

  def mejor_gen(poblacion, solucion):
    # devuelve el mejor gen de la población
    adaptacion=evalua_poblacion(poblacion, solucion)
    return poblacion[adaptacion.index(max(adaptacion))]

  def algoritmo_genetico():
    max_iter=10
    max_poblacion=50
    num_vars=10
    fin=False
    solucion = poblacion_inicial(1, num_vars)[0]
    poblacion = poblacion_inicial(max_poblacion, num_vars)

    iteraciones=0
    while not fin:
      iteraciones=iteraciones+1
      for i in range((len(poblacion))/2):
        gen1, gen2 = seleccion(poblacion, solucion)
        nuevo_gen1, nuevo_gen2 = cruce(gen1, gen2)
        nuevo_gen1 = mutacion(0.1, nuevo_gen1)
        nuevo_gen2 = mutacion(0.1, nuevo_gen2)
        poblacion.append(nuevo_gen1)
        poblacion.append(nuevo_gen2)
        elimina_peores_genes(poblacion, solucion)

      if (max_iter<iteraciones):
        fin=True

    print "Solución: "+str(solucion)
```

```
    mejor = mejor_gen(poblacion, solucion)
    return mejor, adaptacion_3sat(mejor, solucion)



if __name__ == "__main__":
    random.seed()
    mejor_gen = algoritmo_genetico()
    print "Mejor gen encontrado: "+ str(mejor_gen[0])
    print "Función de adaptación: "+str(mejor_gen[1])
```

Listado. 4-16 genetico.py

La ejecución del programa nos ofrece el siguiente resultado:

Solución: [1, 0, 1, 1, 1, 0, 0, 0, 1, 0]

Mejor gen encontrado: [1, 0, 1, 1, 1, 0, 0, 0, 1, 0]

Función de adaptación: 4

Cuando buscamos una solución a un problema usando un algoritmo genético, conviene realizar varias ejecuciones, ya que al igual que pasa con el algoritmo de templado simulado, estamos frente a un algoritmo **no determinista** (no siempre se comporta igual) y alguna de las ejecuciones podría no obtener un buen resultado. Por otro lado, es importante ajustar bien los parámetros del algoritmo a cada problema, en concreto, el número de iteraciones y la población máxima. En nuestro ejemplo, dependiendo del número de variables de la función con la que estemos trabajando, nos vendrá mejor un ajuste u otro. Veamos un par de ejemplos: la siguiente gráfica muestra la velocidad de convergencia hacia la solución ajustando los siguientes parámetros.

Iteraciones=10; Max_poblacion=50; Num_vars=10

© RC Libros

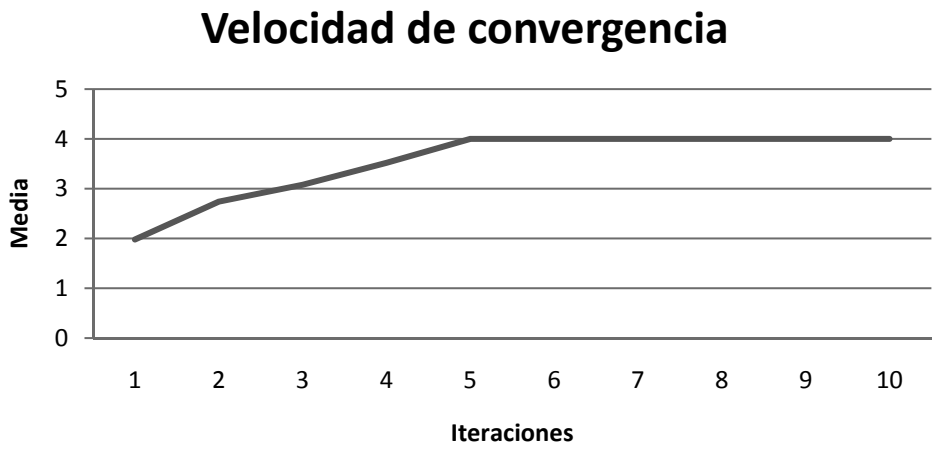


Fig. 4-9 Velocidad de convergencia para pocas variables

Si aumentamos el número de variables, necesitaremos más iteraciones y posiblemente una población más amplia. Los siguientes valores nos dan como resultado la velocidad de convergencia de la figura 4-10.

Iteraciones=30; Max_poblacion=150; Num_vars=100

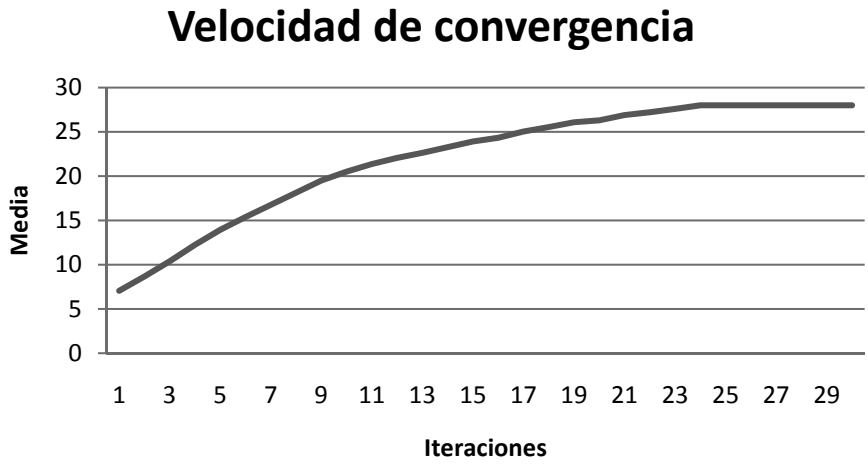


Fig. 4-10 Velocidad de convergencia para más variables

© RC Libros

5 Juegos

INTELIGENCIA ARTIFICIAL Y JUEGOS

Los juegos nos permiten poner en práctica algoritmos como los vistos hasta ahora en entornos más o menos reales. Son un banco de prueba interesante, ya que gracias a ellos podemos poner a prueba los algoritmos contra una inteligencia humana. Además de esto, los juegos suelen ser problemas muy complejos tanto espacial como temporalmente, ya que la mayoría pertenece a la clase NP.

Hay muchos tipos de juegos, pero en este capítulo vamos a centrarnos en una clase especial de juegos que tienen las siguientes características.

- Son juegos con adversario, es decir, compiten dos jugadores con intereses opuestos.
- Son juegos de los denominados de suma cero. Básicamente, esto significa que si un jugador gana, el otro tiene que perder, a no ser que queden en tablas o empatados si el juego lo permite.
- Son juegos perfectamente informados, lo que quiere decir que conocemos toda la información referente al juego incluyendo las jugadas que realiza el oponente.
- En cada jugada, las reglas o movimientos posibles están perfectamente articulados, son precisos y no interviene el azar (como en juegos de cartas o dados).

Dentro de esta categoría entran juegos como *Tres en Raya*, *Ajedrez*, *Damas*, o *Conecta 4*, así como la mayoría de los juegos de tablero. Al final del capítulo se presentarán algunas técnicas para tratar con algunos juegos que no cumplen algunos de estos requisitos.

En general, podemos considerar a los juegos como problemas de búsqueda en los que habitualmente se utiliza alguna heurística. La mayor diferencia estriba en que ahora hay un oponente que no sabemos qué movimiento realizará en la siguiente jugada, y por lo tanto, tendremos que analizar todos sus posibles movimientos. De forma visual, el despliegue de una partida puede representarse mediante un árbol.

La siguiente figura demuestra un despliegue parcial para el juego *Tres en Raya*. Recordemos que este es un juego donde un jugador juega con círculos (O) y el oponente con equis (X), y alternativamente van colocando una ficha en una casilla libre del tablero. Gana el que consigue una línea de tres fichas iguales ya sea en vertical, horizontal o diagonal.

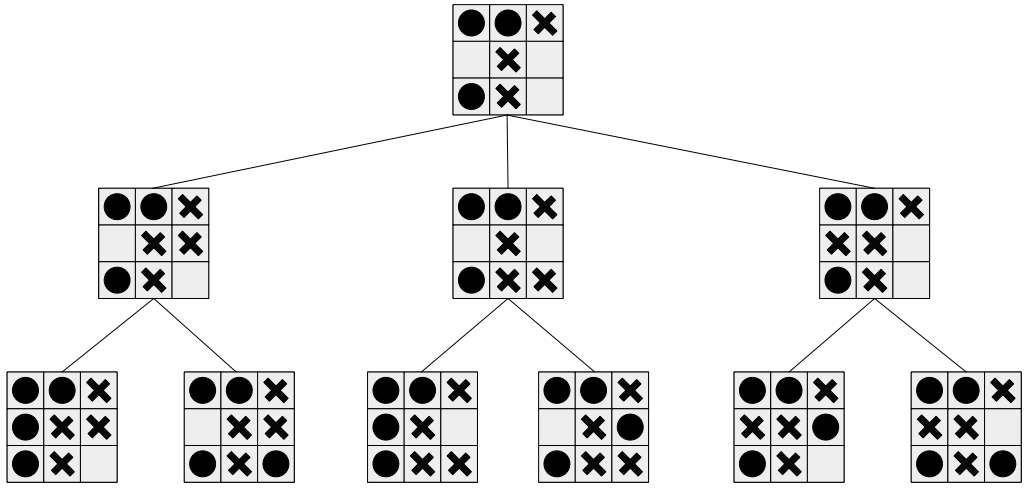


Fig. 5-1 Despliegue parcial del árbol de juego para el Tres en Raya

En el árbol de juego anterior cada uno de los nodos corresponde a un estado del juego que, a su vez, se corresponde con una configuración concreta del tablero. En el nodo raíz tenemos la situación de partida o el estado inicial, que en este caso se corresponde con una partida donde ambos jugadores ya han colocado tres fichas cada uno. En el nodo raíz supondremos que ha movido el jugador O, así que en el

© RC Libros

segundo nivel del árbol le toca jugar al jugador X. Como hay tres casillas libres, el jugador X tiene tres posibles movimientos y, por lo tanto, del nodo raíz parten tres estados nuevos.

En el tercer nivel del árbol el turno es del jugador O que tiene dos posibles movimientos por cada uno de los tres posibles que tenía el jugador X en la jugada anterior.

En este caso solo hemos desplegado dos niveles partiendo desde un estado avanzado de juego. Teniendo en cuenta que en la situación de partida, con el tablero vacío, tenemos 9 posibles movimientos y, por cada uno de los nueve, el oponente tiene 8 posibles jugadas; si representamos todas las posibilidades de movimiento de un juego completo en un árbol, este puede ser inmenso.

Tres en Raya es un buen juego para tomar como ejemplo, ya que aunque hemos visto que el árbol de juego es muy grande, podemos considerar que es bastante contenido en cuanto tamaño con respecto a otros juegos como el *Ajedrez*, que tiene un factor de ramificación medio de 35, lo que nos da un árbol que puede tener 35^{100} nodos o más y que es imposible explorarlo completamente ya que estamos ante un problema NP.

El algoritmo minimax

Acabamos de ver que es computacionalmente imposible explorar el árbol de estados de un juego de *Ajedrez*. El juego de *Tres en Raya* está en el límite de lo que es computacionalmente tratable por un ordenador, así que vamos a usarlo como ejemplo para presentar uno de los algoritmos más usados en juegos. Se trata del **algoritmo minimax**. Una vez presentado el algoritmo básico veremos algunas técnicas para mejorar su rendimiento. También analizaremos cómo podemos adaptarlo a juegos más complejos en los que no es posible desplegar todo el árbol de estados.

El nombre del algoritmo minimax proviene del hecho de que vamos a denominar como MAX al turno del ordenador y MIN al turno del oponente humano al que hay que vencer. El turno del ordenador se llama MAX porque el algoritmo trata de maximizar las posibilidades de vencer y el turno del oponente humano se llama MIN porque es necesario minimizar sus posibilidades de vencer.

© RC Libros

Un **nodo terminal** es aquel en el que gana alguno de los jugadores o hay empate (tablas). El algoritmo minimax, al igual que ocurre con los algoritmos estudiados anteriormente, utiliza una función de evaluación que, por ahora, utilizaremos para evaluar los nodos terminales. En el caso de *Tres en Raya* nos interesa saber si, cuando evaluamos un nodo terminal, ha vencido el ordenador, el oponente o ha habido empate. Una posible función de evaluación es aquella que asigna el valor +1 a los nodos terminales que dan la victoria al ordenador, -1 a los nodos terminales en los que vence el oponente humano y 0 para los casos en los que hay empate.

El algoritmo minimax es un algoritmo que recorre el árbol de juego en profundidad hasta alcanzar un nodo terminal. Una vez alcanzado, se evalúa y se le asigna el valor +1, -1 o 0 según la función de evaluación que acabamos de definir. Una vez marcado el nodo terminal con su valor correspondiente, el algoritmo minimax irá propagando estos valores hacia los nodos padre, de forma que si un nodo padre tiene dos o más hijos con distintos valores, este heredará el valor menor si se trata de un nodo MIN (es el turno del jugador humano) o el valor mayor si es un nodo MAX (es el turno del ordenador). Este comportamiento es totalmente coherente con el objetivo que perseguimos. Si es el turno del jugador MAX, queremos maximizar la función de evaluación para que el ordenador gane la partida; es decir, seleccionaremos el valor más alto de entre todos los nodos hijo (idealmente el valor +1). Si es el turno de MIN queremos minimizar la función de evaluación por lo que escogeremos el nodo hijo con el valor más bajo (idealmente el valor -1).

En la figura 5-2 se muestra el despliegue del árbol de juego completo que ya vimos en la figura 5-1. Se han añadido los nombres de los niveles para saber a qué turno corresponde cada uno (MIN o MAX) y se han indicado los valores de los nodos según la función de evaluación que hemos definido. En negrita y subrayado están los valores de los nodos terminales. Los demás valores son los propagados desde los nodos hijo.

En esta figura también podemos apreciar cómo no todos los nodos terminales tienen por qué estar en el mismo nivel. Por ejemplo, en el tercer nivel del árbol (nivel MAX) hay dos nodos terminales en los que la victoria es para el jugador humano y, por lo tanto, son marcados con el valor -1 por la función de evaluación. En el cuarto nivel (nivel MIN) tenemos dos victorias para el ordenador, que son marcadas con el valor +1 por la función de evaluación y otros dos nodos terminales marcados con 0, ya que se da una situación de tablas.

© RC Libros

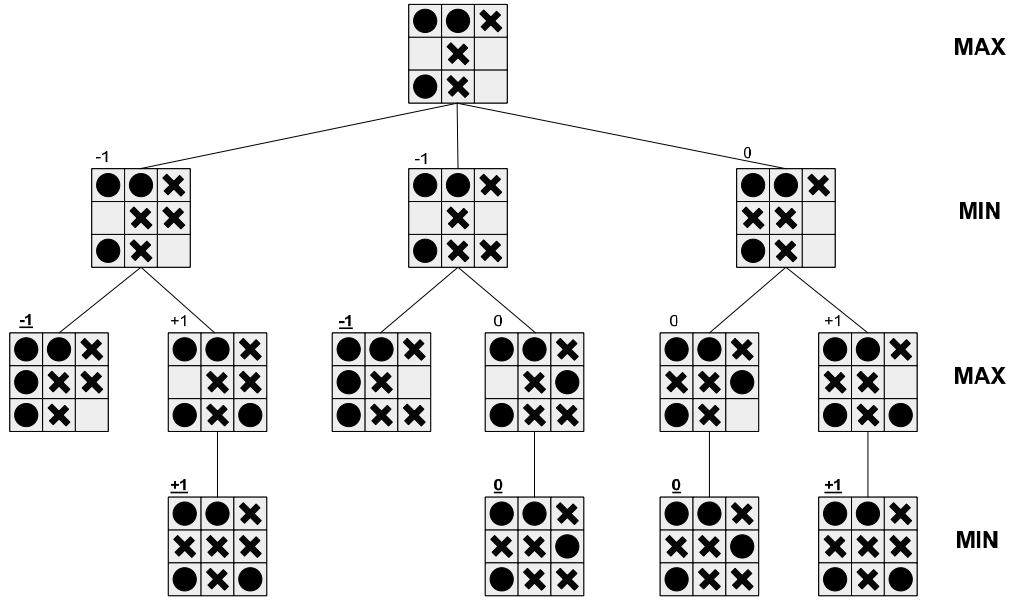


Fig. 5-2 Despliegue del árbol minimax para Tres en Raya

Vamos a analizar por separado cada una de las tres ramas principales que parten del nodo raíz para ver cómo se propagarían los valores hacia arriba.

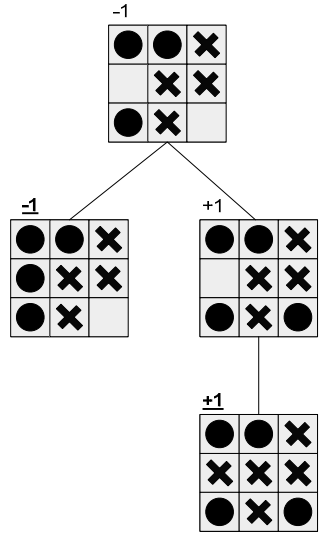


Fig. 5-3 Despliegue primera rama del árbol minimax

© RC Libros

En el tercer nivel de la primera rama (fig. 5-3) hay una situación de vencedor para el ordenador, por lo que se marca con el valor +1. En el segundo nivel, hay otro nodo terminal cuya victoria corresponde al jugador humano, por lo que marcamos el nodo con -1. El otro nodo de este nivel toma el valor de su hijo: +1.

Finalmente, en el primer nodo se toma el valor -1, ya que al tratarse de un nodo MIN seleccionaremos el menor valor de los nodos hijo.

Aunque hay una situación ganadora y favorable para el ordenador en el tercer nivel de esta rama, el primer nodo se ha marcado con el valor -1 porque en la siguiente jugada, que es turno MIN, el algoritmo supone siempre que el oponente nunca cometerá un error y acabará llegando al nodo terminal del nivel 2 en el que gana.

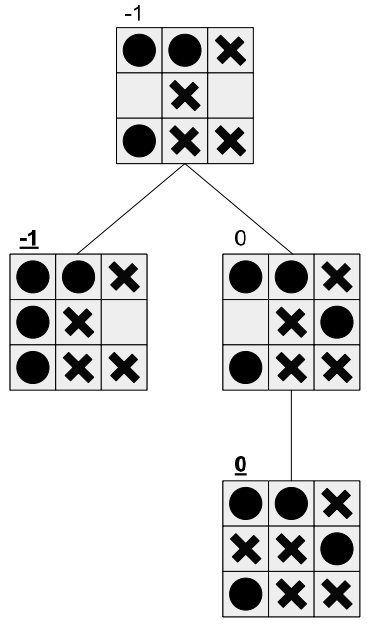


Fig. 5-4 Despliegue segunda rama del árbol minimax

En el tercer nivel de esta segunda rama (fig. 5-4) hay una situación de empate que se marca con el valor 0.

© RC Libros

En el segundo nivel hay otro nodo terminal cuyo desenlace corresponde a una victoria del oponente, por lo que se marca con valor -1. El otro nodo de este nivel tiene el valor 0 que es el del hijo.

En el primer nodo se seleccionará el valor -1 ya que es el turno del oponente, es decir, un nivel MIN.

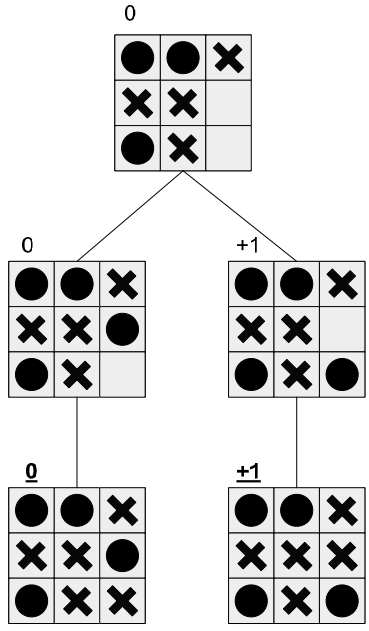


Fig. 5-4 Despliegue tercera rama del árbol minimax

En el tercer nivel de esta tercera rama (fig. 5-4) hay dos nodos terminales. El primero lleva a empate y se marca con el valor 0. El segundo corresponde a una victoria del ordenador por lo que se marca con +1.

En el segundo nivel se propagan los valores de los nodos hijo hacia arriba.

En el primer nodo, como es turno de MIN se selecciona el valor más bajo que es 0. Esto es debido a que el algoritmo minimax supone que el jugador MIN siempre elegirá el nodo marcado con 0 frente al nodo marcado con +1 en el segundo nivel.

© RC Libros

Si observamos el segundo nivel del árbol de la figura 5-2, vemos que hay tres nodos hijo del nodo raíz, cada uno con su correspondiente valor. Como el nodo raíz es un nodo MAX elegirá la rama con mayor valor, que en este caso es la tercera, marcada con el valor 0. Cualquiera de las otras dos le llevaría a una victoria del oponente, mientras que esta, en el peor de los casos, le llevará a un empate. El pseudocódigo para el algoritmo minimax es el siguiente.

```
función minimax(jugador, tablero)
   Si nodo es terminal:
       devolver ganador

   nodos_hijos=todos los movimientos legales desde el estado actual
   Si es el turno de MAX:
       devolver valor máximo de la llamada a minimax() para nodos hijo.
   Sino:
       devolver valor mínimo de la llamada a minimax() para nodos hijo.
```

Listado. 5-1 Pseudocódigo del algoritmo minimax

El siguiente código implementa el juego de *Tres en Raya* usando el algoritmo minimax.

```
# tres en raya con algoritmo minimax
import sys

MAX = 1
MIN = -1
global jugada_maquina

def minimax(tablero, jugador):
  global jugada_maquina

  # Hay ganador o tablas? (nodo terminal)
  if game_over(tablero):
    return [ganador(tablero), 0]
```


© RC Libros

```
  # generar las posibles jugadas
  movimientos=[]
  for jugada in range(0,len(tablero)):
    if tablero[jugada] == 0:
      tableroaux=tablero[:]
      tableroaux[jugada] = jugador

      puntuacion = minimax(tableroaux, jugador*(-1))
      movimientos.append([puntuacion, jugada])

  if jugador == MAX:
    movimiento = max(movimientos)
    jugada_maquina = movimiento[1]
    return movimiento
  else:
    movimiento = min(movimientos)
    return movimiento[0]


def game_over(tablero):
  # Hay tablas?
  no_tablas = False
  for i in range(0,len(tablero)):
    if tablero[i] == 0:
      no_tablas = True

  # Hay ganador?
  if ganador(tablero) == 0 and no_tablas:
    return False
  else:
    return True
```

```
def ganador(tablero):
  lineas = [[0,1,2], [3,4,5], [6,7,8], [0,3,6], [1,4,7], [2,5,8], \
  [0,4,8], [2,4,6]]
  ganador = 0
  for linea in lineas:
    if tablero[linea[0]] == tablero[linea[1]] and \
    tablero[linea[0]] == tablero[linea[2]] and tablero[linea[0]] != 0:
      ganador = tablero[linea[0]]

  return ganador

def ver_tablero(tablero):
    for i in range(0,3):
      for j in range(0,3):
        if tablero[i*3+j] == MAX:
          print 'X',
        elif tablero[i*3+j] == MIN:
          print 'O',
        else:
          print '.',

      print ''

def juega_humano(tablero):
  ok=False
  while not ok:
    casilla = input ("Casilla?")
    if str(casilla) in '0123456789' and len(str(casilla)) == 1 \
    and tablero[casilla-1] == 0:
      if casilla == 0:
        sys.exit(0)
      tablero[casilla-1]=MIN
      ok=True
  return tablero
```


© RC Libros

```
def juega_ordenador(tablero):
  global jugada_maquina
  punt = minimax(tablero[:], MAX)
  tablero[jugada_maquina] = MAX
  return tablero


if __name__ == "__main__":
  print 'Introduce casilla o 0 para terminar'
  tablero = [0,0,0,0,0,0,0,0,0]

  while (True):
    ver_tablero(tablero)
    tablero = juega_humano(tablero)
    if game_over(tablero):
      break

    tablero = juega_ordenador(tablero)
    if game_over(tablero):
      break

  ver_tablero(tablero)
  g = ganador(tablero)
  if g == 0:
    gana = 'Tablas'
  elif g == MIN:
    gana = 'Jugador'
  else:
    gana = 'Ordenador'

  print 'Ganador:' + gana
```

Listado. 5-2 tic-tac-toe.py

© RC Libros

Para representar gráficamente un árbol minimax genérico, se utilizan triángulos con la punta hacia arriba para indicar que se trata de un nodo MAX y un triángulo hacia abajo para los nodos MIN. Junto a los triángulos ponemos el valor de la función de evaluación. La siguiente figura muestra el despliegue de un árbol minimax para un juego cualquiera. En la figura podemos observar cómo el valor de la función de evaluación puede ser cualquier valor arbitrario siempre que este sea directamente proporcional a la conveniencia o no de la jugada para el ordenador; es decir, a mayor valor, mejor jugada para el ordenador.

En la figura 5-5 podemos ver cómo el nodo raíz queda finalmente etiquetado con el valor 12, lo que quiere decir que en la siguiente jugada optará por realizar el movimiento que conduce al nodo hijo etiquetado con el 12.

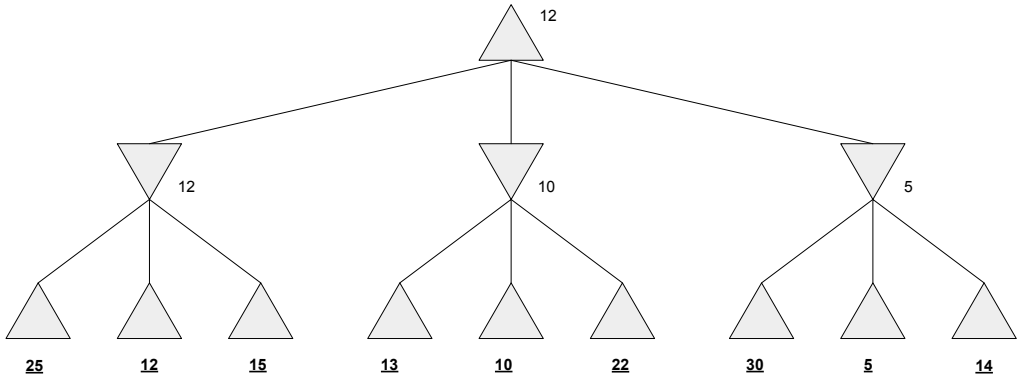


Fig. 5-5 Despliegue de árbol minimax

Hemos comenzado diciendo que el juego de *Tres en Raya* está en los límites de lo computable, lo que quiere decir que podemos desplegar todo el árbol de juego. Esto tiene la ventaja de que el ordenador conoce de antemano todas las posibles jugadas y no es posible sorprenderlo. Podemos decir que nunca puede perder y en el peor de los casos empatará.

Desgraciadamente, en juegos más complejos no podremos desplegar todo el árbol del juego, solo unos pocos niveles, por lo que la mayoría del tiempo no estaremos alcanzando nodos terminales. Un juego como el *Ajedrez* o las *Damas* tienen un factor de ramificación muy alto que nos hace imposible avanzar más que unas pocas jugadas (niveles del árbol) en un tiempo razonable. ¿Qué hacer entonces?

© RC Libros

Vamos a suponer que estamos programando un juego de *Ajedrez* y hemos visto que podemos descender cinco niveles del árbol pero más allá de esto el tiempo de proceso no es razonable, por lo que tomamos la decisión de que nuestro algoritmo minimax solo explorará cinco niveles a partir del estado actual. A aquellos nodos que se encuentran en el quinto nivel y a partir de los cuales desconocemos sus nodos hijo los llamaremos **nodos frontera**. Podemos imaginar los nodos frontera como aquellos a partir de los cuales el algoritmo minimax no es capaz de ver qué ocurre.

Si dentro de estos cinco niveles encontramos un nodo terminal, lo seguiremos tratando como tal. Sin embargo, a los nodos frontera tendremos que asignarles un valor mediante la función de evaluación para que podamos seguir usando el algoritmo minimax tal y como lo hemos venido haciendo hasta ahora. Evidentemente, la función de evaluación ahora será ligeramente diferente a la que usábamos en el *Tres en Raya*. No podemos asignar los valores +1, -1 o 0 porque en los nodos frontera aún no hay un ganador ni sabemos si hay empate, así que lo que podemos hacer es asignarle un valor que no será más que una estimación de lo favorable que es la jugada en ese nodo frontera. Al ser una estimación nunca estamos seguros de que es la mejor jugada posible, pero basándonos en la experiencia y en las reglas de cada juego, podemos construir una función heurística que nos permita valorar la jugada.

Vamos a utilizar como ejemplo un juego algo más complicado que el *Tres en Raya* pero lo suficientemente simple como para poder aplicar los conceptos básicos sin perdernos en un maremágnum de reglas complicadas. Nos centraremos en un juego llamado *Conecta 4*. El *Conecta 4* se juega en un tablero vertical de 7x7 donde los jugadores van introduciendo fichas redondas alternativamente. Gana aquel jugador que consigue primero crear una línea vertical, horizontal o diagonal de cuatro fichas del mismo color.

En el juego del *Conecta 4*, cuando analizamos un nodo frontera, no tiene por qué haber aún un ganador o indicios de empate, por lo que tendremos que analizar la situación de las fichas para ver cuál es más favorable al ordenador. Intuitivamente podemos pensar que una jugada en la que ya hay formada una línea de tres fichas es más favorable que otra en la que solo hay dos. De la misma forma, una línea de dos siempre será mejor que ninguna, aunque evidentemente no tiene tanta importancia como la de tres. Hay, sin embargo, situaciones que no son tan fáciles de dirimir. Supongamos que tenemos la posibilidad de poner la ficha en una casilla formaría tres líneas de dos fichas o en otra que formaría solo una línea de tres fichas. ¿Cuál es la mejor jugada de las dos? ¿Por cuál nos decidimos? Evidentemente, la experiencia y el conocimiento del juego son factores clave a la hora de elegir una buena función de

© RC Libros

evaluación. Huelga decir que a mejor función de evaluación, mejor jugará el ordenador al juego que estemos implementando.

Para el caso del *Conecta 4* vamos a quedarnos con la siguiente función de evaluación:

$$f = (n2 \times 4) + (n3 \times 9) + (n4 \times 1000)$$

Donde n2 es el número de líneas de dos fichas, n3 es el número de líneas de tres fichas y n4 el número de líneas de 4 fichas (líneas ganadoras). Según esta función, una línea de tres fichas vale más del doble que una de dos. Evidentemente, el valor de una de cuatro tiene que ser lo suficientemente alta como para que, sin importar el número de líneas de dos o de tres, siempre elijamos la de cuatro.

Como ejemplo, supongamos que estamos evaluando un nodo en el que tenemos dos líneas de dos fichas y una de tres. La función de evaluación arrojaría el siguiente valor:

$$f = (2 \times 4) + (1 \times 9) + (0 \times 1000) = 17$$

La función de evaluación que hemos elegido para el juego del *Conecta 4* suele funcionar bien; sin embargo, para otros juegos la elección de una buena función puede ser mucho menos obvia. Tomemos como ejemplo el juego del *Ajedrez*. Aquí hay muchos más factores que entran en juego:

- Número de piezas sobre el tablero: cada pieza tiene un valor relativo con respecto a las demás. Un peón es mucho menos importante que la reina o un alfil. Se suele asignar un valor a las piezas y la suma de los valores de las fichas sobre el tablero forma parte del valor final de la función de evaluación.
- Cuántas de las fichas del tablero están amenazadas: si una pieza está amenazada es probable que en el futuro sea capturada, por lo que también influye en el valor final de la función de evaluación. En este caso también se asigna un peso relativo a cada pieza según de qué clase sea.
- Posición de las piezas: la experiencia dice que el jugador que domina el centro del tablero tiene mejores condiciones para ganar, por lo que también tenemos que tener en cuenta este factor.
- Jaque: si tenemos al oponente en jaque la función de evaluación tendrá que ofrecer un valor muy alto ya que la jugada es muy favorable.

Hay cientos de consideraciones más que podríamos tener en cuenta, pero estas valen como ejemplo.

© RC Libros

Hay que considerar también que la función de evaluación se ejecutará una vez por cada nodo analizado, por lo que no conviene que sea demasiado pesada. Hay una mejora que podemos realizar sobre el algoritmo minimax para aliviar un poco el peso del algoritmo. El problema del algoritmo minimax es que en cada evaluación de un estado hay que comprobar si estamos en un nivel MIN o en un nivel MAX. Podemos ahorrarnos esta comprobación si observamos la siguiente igualdad matemática.

$$\max(x, y) = -\min(-x, -y)$$

Lo veremos más claro con el siguiente ejemplo:

$$\max(3, 5) = 5$$

$$-\min(-3, -5) = -(-5) = 5$$

Basándonos en esta igualdad, podemos asegurar que si al descender un nivel en el árbol de juego cambiamos el signo de los valores de la función de evaluación, el resultado obtenido al quedarnos siempre con el mayor de los valores será equivalente al obtenido por el algoritmo minimax. Esta mejora es conocida como **algoritmo negamax**, debido al hecho de que siempre que llamamos a la función negamax de forma recursiva lo hacemos poniendo un signo negativo delante. El siguiente listado muestra el pseudocódigo del algoritmo negamax.

```
función negamax(jugador, tablero)
  Si es nodo terminal o frontera:
     devolver varlor de la función de evaluación
  nodos_hijos=todos los movimientos legales desde el estado actual
  devolver valor máximo de la llamada a -negamax() para cada nodo hijo
```

Listado. 5-3 Algoritmo negamax

Poda alfa-beta

La poda alfa-beta es una técnica que nos permite mejorar el rendimiento del algoritmo minimax. La idea básica consiste en no examinar aquellas partes del árbol que sabemos que no nos van a aportar más información de la que ya disponemos. El nombre de poda proviene del hecho de que al no examinar ciertas ramas es como si las podáramos. Se estima que al usar la poda alfa-beta obtenemos una mejora media

del 30% de rendimiento respecto al algoritmo minimax aunque, como veremos, todo depende de la distribución de los nodos y sus valores según la función de evaluación.

Vamos a comenzar con un ejemplo simple que nos permita hacernos una idea intuitiva del funcionamiento de esta técnica y después examinaremos más en detalle el funcionamiento y la implementación de la poda alfa-beta. En todo caso, podremos usarla en los dos algoritmos que hemos visto en el apartado anterior ya sea en el algoritmo minimax o en el algoritmo negamax.

En la siguiente figura vemos un despliegue de un supuesto árbol de juego minimax con los valores de la función de evaluación en sus nodos terminales o frontera. Nos centraremos en analizar qué ocurre en los nodos marcados con una estrella.

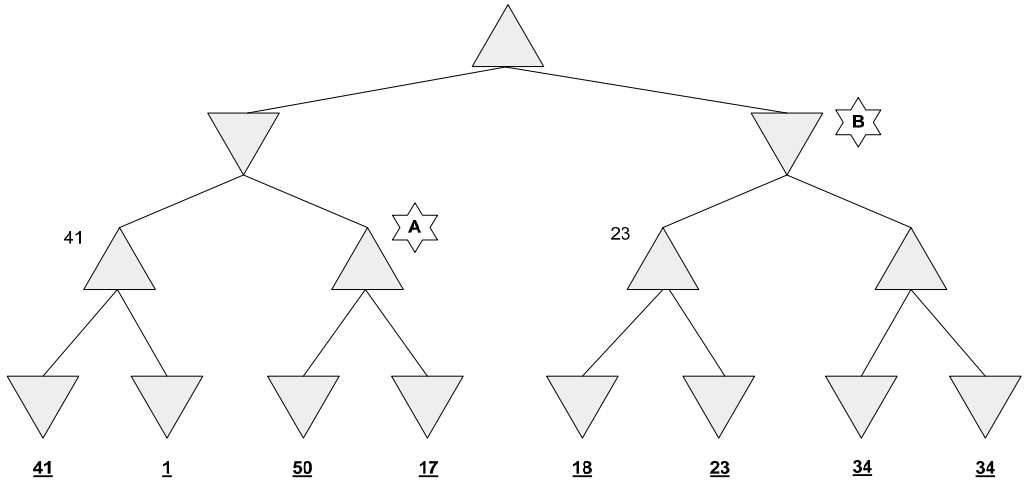


Fig. 5-6 Minimax sin poda alfa-beta

El padre del nodo marcado con la estrella A deberá elegir entre sus dos nodos hijo. Podemos observar que el valor de su primer nodo hijo es 41. El nodo A, usando el valor de su primer hijo, obtendrá el valor provisional de 50, a la espera de examinar el otro nodo hermano; sin embargo, el nodo padre de A es un nodo MIN, lo que quiere decir que escogerá el valor mínimo de entre 41 y el valor que resulte de evaluar el nodo A. Como A es un nodo MAX, podemos asegurar que su valor, como mínimo, será de 50, ya que este valor solo será sustituido si el otro nodo hijo de A tiene un valor superior. Bajo este escenario, no tiene sentido seguir explorando la rama del segundo nodo hijo de A ya que al tener como mínimo un valor de 50, no

© RC Libros

será seleccionado por el nodo padre de A que recordemos que es MIN. Por lo tanto, siempre seleccionará el nodo hijo con valor 41.

Ocurre algo parecido en el nodo marcado con la estrella B. Tras evaluar la primera sub-rama de B habremos asignado el valor 23 de forma provisional a B. Como B es un nodo MIN, sea cual sea el valor obtenido tras evaluar la segunda sub-rama, el valor de B será 23 o inferior (nunca mayor por ser un nodo MIN). Bajo este escenario no tiene sentido examinar la segunda sub-rama de B ya que el nodo padre de B (nodo raíz) es MAX y elegirá el valor mayor entre 41 y el obtenido de la rama del nodo B, que ya sabemos que es como mucho 23.

En la siguiente figura hemos marcado con un aspa aquellas ramas que podemos podar así como el despliegue completo con sus valores asociados a cada nodo.

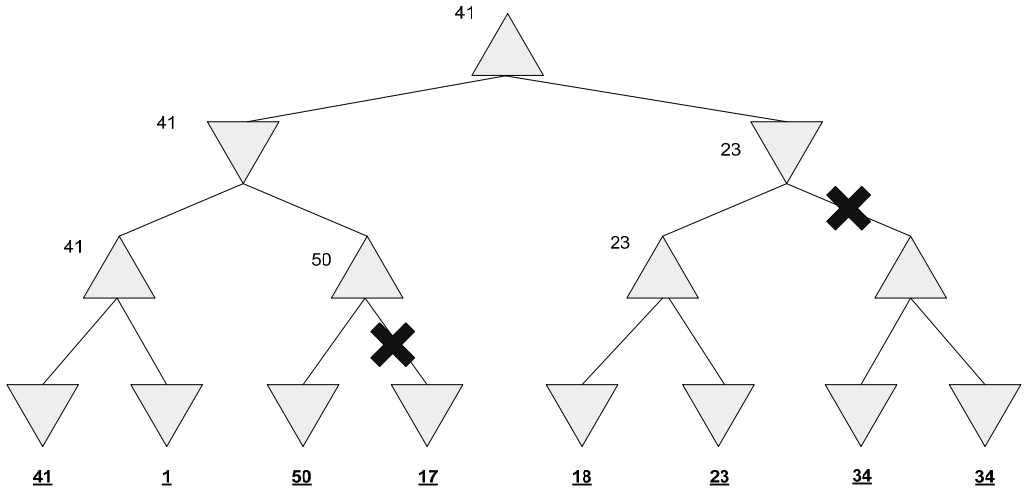


Fig. 5-7 Minimax con poda alfa-beta

Hemos advertido antes que la mejora media es de un 30%; sin embargo, el nivel de mejora de la poda alfa-beta será mucho mejor si los nodos están ordenados convenientemente. Por ejemplo, si intercambiamos los nodos terminales con valor 50 y 17, ahora no habríamos podido hacer la poda del segundo hijo de B, ya que 17 es menor que 41 y por lo tanto no sería posible descartar la rama.

© RC Libros

En la implementación de la poda alfa-beta se van marcando los nodos con dos etiquetas dependiendo si se trata de un nodo MAX o un nodo MIN. A estas dos etiquetas las lamamos alfa y beta. De ahí el nombre del algoritmo. Las etiquetas alfa contienen el mejor valor obtenido por la función de evaluación en cualquiera de los descendientes de ese nodo para el jugador MAX. Análogamente, las etiquetas beta contienen el peor valor obtenido por la función de evaluación para todos los descendientes de ese nodo para el jugador MIN. Siempre que encontremos un nodo en el que alfa es mayor o igual que beta podremos podar esa rama.

El siguiente listado es el pseudocódigo de la poda alfa-beta usando el algoritmo minimax.

```
función minimax_alfa_beta(jugador, tablero, alfa, beta):
  Si es nodo terminal:
    devolver valor de función de evaluación
  nodos_hijos=todos los movimientos legales desde el estado actual
  Si es el turno de MAX:
    por cada nodo hijo:
      puntuacion=minimax_alfa_beta(MIN, tablero_hijo, alfa, beta)
      si puntuacion>alfa:
        alfa=puntuacion
      si alfa>=beta:
        devolver alfa #(poda)

    devolver alfa
  Sino
    por cada nodo hijo:
      puntuacion=minimax_alfa_beta(MAX, tablero_hijo, alfa, beta)
      si puntuacion<beta:
        beta=puntuacion
      si alfa>=beta:
        devolver beta #(poda)

    devolver beta
```

Listado. 5-4 Poda alfa-beta para minimax

© RC Libros

La poda alfa-beta también puede aplicarse con el algoritmo negamax. De hecho, se simplifica bastante al no tener que estar comprobando si estamos en un nivel MIN o MAX.

```
función negamax_alfa_beta(jugador, tablero_hijo, alfa, beta):
  Si es nodo terminal o frontera:
    devolver valor de la función de evaluación
  nodos_hijos=todos los movimientos legales desde el estado actual
  por cada nodo hijo:
    puntucion = -negamax_alfa_beta(-jugador, tablero, -beta, -alfa)
    si puntucion>alfa:
      alfa=puntuacion
    si alfa>=beta:
      devolver alfa #(poda)

  devolver alfa
```

Listado. 5-5 Poda alfa-beta para negamax

Para asentar los conceptos de poda alfa-beta, mostramos a continuación el listado para una implementación del juego *Conecta 4* usando poda alfa-beta sobre el algoritmo negamax. Se ha utilizado como función de evaluación la descrita más arriba en el ejemplo. Hemos añadido la posibilidad de jugar en tres niveles diferentes: en el primer nivel el algoritmo negamax desciende hasta 3 niveles en el árbol de juego, en el nivel medio 4 y en el difícil 6. Evidentemente, en el nivel difícil el ordenador tarda más en procesar cada jugada.

```
# conecta4 con poda alfa-beta
import sys
import copy

MAX = 1
MIN = -1
MAX_PROFUNDIDAD=4
```

© RC Libros

```
def negamax(tablero, jugador, profundidad, alfa, beta):
  max_puntuacion = -sys.maxint-1
  alfa_local = alfa
  for jugada in range(7):
    # columna totalmente llena?
    if tablero[0][jugada] == 0:
      tableroaux = copy.deepcopy(tablero)
      inserta_ficha(tableroaux, jugada, jugador)
      if game_over(tableroaux) or profundidad==0:
        return [evalua_jugada(tableroaux, jugador), jugada]
      else:
        puntuacion = -negamax(tableroaux, jugador*(-1), \
         profundidad-1, -beta, -alfa_local)[0]
      if puntuacion>max_puntuacion:
        max_puntuacion = puntuacion
        jugada_max=jugada

      # poda alfa beta
      if max_puntuacion >= beta:
        break
      if max_puntuacion > alfa_local:
        alfa_local = max_puntuacion

  return [max_puntuacion, jugada_max]


def evalua_jugada(tablero, jugador):
  n2=comprueba_linea(tablero, 2, jugador)[1]
  n3=comprueba_linea(tablero, 3, jugador)[1]
  n4=comprueba_linea(tablero, 4, jugador)[1]
  valor_jugada = 4*n2+9*n3+1000*n4
  return valor_jugada
```


© RC Libros

```
def game_over(tablero):
  # Hay tablas?
  no_tablas = False
  for i in range(7):
    for j in range(7):
      if tablero[i][j] == 0:
        no_tablas = True

  # Hay ganador?
  if ganador(tablero)[0] == 0 and no_tablas:
    return False
  else:
    return True



def comprueba_linea(tablero, n, jugador):
  # Comprueba si hay una linea de n fichas
  ganador = 0
  num_lineas = 0
  lineas_posibles=8-n

  # Buscar línea horizontal
  for i in range(7):
    for j in range(lineas_posibles):
      cuaterna = tablero[i][j:j+n]
      if cuaterna == [tablero[i][j]]*n and tablero[i][j]!=0:
        ganador = tablero[i][j]
        if ganador==jugador:
          num_lineas=num_lineas+1;

  # Buscar línea vertical
  for i in range(7):
    for j in range(lineas_posibles):
      cuaterna=[]
```

```
    for k in range(n):
      cuaterna.append(tablero[j+k][i])
    if cuaterna == [tablero[j][i]]*n and tablero[j][i]!=0:
      ganador = tablero[j][i]
      if ganador==jugador:
        num_lineas=num_lineas+1;

# Buscar línea diagonal
for i in range(4):
  for j in range(lineas_posibles-i):
    cuaterna1=[]
    cuaterna2=[]
    cuaterna3=[]
    cuaterna4=[]

    for k in range(n):
      cuaterna1.append(tablero[i+j+k][j+k])
      cuaterna2.append(tablero[i+j][i+j+k])
      cuaterna3.append(tablero[i+j+k][6-(j+k)])
      cuaterna4.append(tablero[j+k][6-(i+j+k)])

    if cuaterna1==[cuaterna1[0]]*n and tablero[i+j][j]!=0:
      ganador = tablero[i+j][j]
      if ganador==jugador:
        num_lineas=num_lineas+1;

    elif cuaterna2==[cuaterna2[0]]*n and tablero[i+j][i+j]!=0:
      ganador = tablero[i+j][i+j]
      if ganador==jugador:
        num_lineas=num_lineas+1;

    elif cuaterna3==[cuaterna3[0]]*n and tablero[i+j][6-j]!=0:
      ganador = tablero[i+j][6-j]
      if ganador==jugador:
```


© RC Libros

```
          num_lineas=num_lineas+1;

      elif cuaterna4==[cuaterna4[0]]*n and tablero[j][6-(i+j)]!=0:
        ganador = tablero[j][6-(i+j)]
        if ganador==jugador:
          num_lineas=num_lineas+1;

  return (ganador, num_lineas)

def ganador(tablero):
  return comprueba_linea(tablero, 4, 0)

def ver_tablero(tablero):
  for i in range(7):
    for j in range(7):
      if tablero[i][j] == MAX:
        print 'X',
      elif tablero[i][j] == MIN:
        print 'O',
      else:
        print '.',
    print ''
  print '-------------'
  print '1 2 3 4 5 6 7'

def inserta_ficha(tablero, columna, jugador):
  # encontrar la primera casilla libre en la columna
  # y colocar la ficha
  ok = False
  for i in range(6,-1, -1):
    if (tablero[i][columna] == 0):
      tablero[i][columna]=jugador
      ok=True
      break
```

```
  return ok

def juega_humano(tablero):
  ok=False
  while not ok:
    col = input ("Columna (0=salir)?")
    if str(col) in '01234567' and len(str(col)) == 1:
      if col==0:
        sys.exit(0)
      ok=inserta_ficha(tablero, col-1, MIN)
    if ok == False:
      print "Movimiento ilegal"

  return tablero

def juega_ordenador(tablero):
  tablerotmp=copy.deepcopy(tablero)
  punt, jugada = \
  negamax(tablerotmp, MAX, MAX_PROFUNDIDAD, -sys.maxint-1, sys.maxint)
  inserta_ficha(tablero, jugada, MAX)

  return tablero

if __name__ == "__main__":
  # tablero de 7x7
  tablero = [[0 for j in range(7)] for i in range(7)]

  ok=False
  profundidades=[3, 4, 6]
  while not ok:
    dificultad = input ("Dificulatad (1=facil; 2=medio; 3=dificil): ")
    if str(dificultad) in '123' and len(str(dificultad)) == 1:
      MAX_PROFUNDIDAD=profundidades[dificultad-1]
```


© RC Libros

```
    ok=True


while (True):
  ver_tablero(tablero)
  tablero = juega_humano(tablero)
  if game_over(tablero):
    break

  tablero = juega_ordenador(tablero)
  if game_over(tablero):
    break

ver_tablero(tablero)

g = ganador(tablero)[0]
if g == 0:
  gana = 'Tablas'
elif g == MIN:
  gana = 'Jugador'
else:
  gana = 'Ordenador'

print 'Ganador:' + gana
```

Listado. 5-6 conecta4.py

Otros tipos de juegos

Nos hemos centrado durante el capítulo en lo que se denominan juegos perfectamente informados. Siempre para dos oponentes. Además, ninguno de estos juegos tiene ninguna componente aleatoria o de azar como puedan ser las cartas o juegos en los que intervienen el uso de dados. Vamos a revisar someramente las

técnicas que podemos usar para juegos de más de dos jugadores y los juegos en los que interviene el azar.

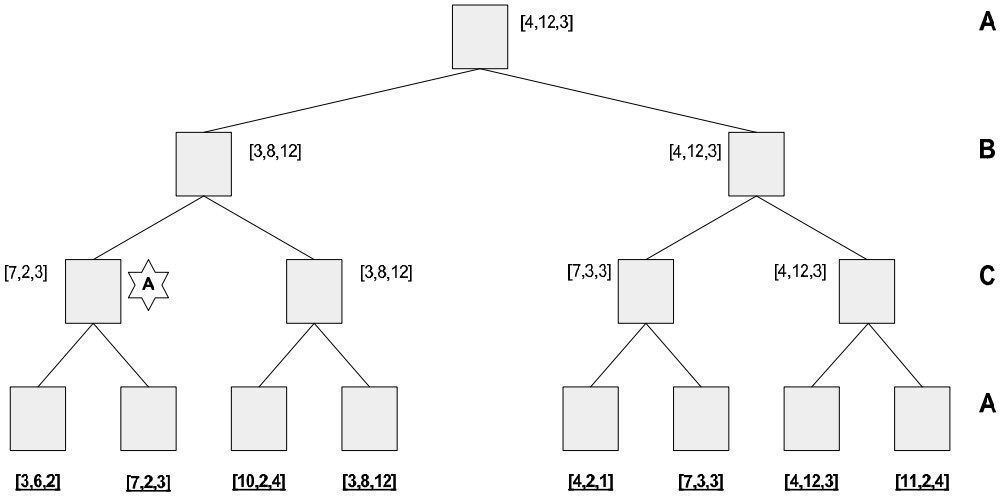


Fig. 5-8 Minimax con varios contrincantes

En juegos donde compiten más de dos jugadores, como puedan ser el *Parchís*, el *Dominó* o cualquier otro juego similar, será necesario hacer algunas adaptaciones al algoritmo minimax para que nos siga siendo útil. Supongamos un juego de tres jugadores A, B y C, cuyo árbol minimax es el de la figura 5-8. En cada nodo tendremos un vector con los valores para la función de evaluación correspondiente a cada jugador, en este caso, un vector con tres valores, donde el primer valor corresponde al jugador A, el segundo al jugador B y el tercero al C. En el árbol minimax correspondiente a dos jugadores que hemos analizado al principio del capítulo no era necesario poner la puntuación de ambos jugadores ya que al tratarse de juego de suma cero, el valor obviado del otro jugador podíamos suponerlo igual al ofrecido por la función de evaluación pero con signo contrario.

Imaginemos que en un momento dado nos encontramos evaluando el nodo marcado con la estrella A. En este nodo se da la circunstancia de que es el turno del jugador C. ¿Cuál de los dos nodos hijo escogería este jugador? El primer hijo tiene un valor en su función de evaluación (tercer valor de vector) de 2, mientras que el

© RC Libros

segundo hijo tiene el valor 3, por lo que será preferible elegir el segundo (siempre desde la perspectiva del jugador C).

Por claridad se ha optado por poner en cada nodo el vector correspondiente a la mejor jugada desde el punto de vista del jugador que tiene el turno. Evidentemente, una implementación del algoritmo minimax para más de dos jugadores tendría que tratar de maximizar la función de evaluación del ordenador y minimizar la de los adversarios tal y como hacíamos en la versión para dos jugadores.

De esta forma modelamos el comportamiento de juegos con varios jugadores; sin embargo, muchos de estos juegos tienen una componente aleatoria. Juegos como el *Backgammon* o el *Parchís* introducen esta aleatoriedad mediante el uso de dados. A este grupo de jugos los llamamos **juegos estocásticos**.

En los juegos donde no interviene el azar es fácil construir el árbol ya que conocemos, sea cual sea el nivel de profundidad en el árbol, todos los movimientos legales que puede realizar el jugador. En los juegos en los que intervienen dados o cualquier elemento de azar no es posible avanzar cuáles serán los movimientos legales de las siguientes jugadas, ya que estos dependen de los valores obtenidos al lanzar los dados. Por ejemplo, en el juego del *Parchís* no sabemos cuántas casillas podremos avanzar hasta que no tiremos los dados, por lo que, a priori, no podemos construir el árbol de juego.

Para tratar con esta situación, vamos a añadir un nuevo tipo de nodo al árbol minimax. A estos nodos lo llamamos **nodos de probabilidad** y se representan como círculos en el árbol minimax. De cada nodo de probabilidad parten tantos hijos como posibles resultados haya al lanzar los dados. A cada uno de esos hijos lo etiquetamos con la probabilidad que tiene cada valor de salir. Si solo jugamos con un dado, tendremos una probabilidad de 1/6 de que salga un 1, 1/6 de que salga un 2 y así sucesivamente. Si el juego se juega con dos dados, en vez de 6 posibles resultados tendremos 6x6 = 36 resultados. Si tenemos en cuenta que sacar un 3 en el primer dado y un 4 en el segundo es lo mismo que sacar 4 en el primero y 3 en el segundo, nos quedan 21 posibilidades. Por otra parte tenemos una probabilidad de 1/36 de sacar el mismo valor en ambos dados (por ejemplo, un seis doble), lo que nos deja una probabilidad de sacar el mismo valor en ambos dados de 1/36 y una probabilidad de 1/18 de sacar cualquiera de las otras 21-6 = 15 combinaciones. En la siguiente figura podemos observar el despliegue del árbol para una jugada de un juego jugado con dos dados.

© RC Libros

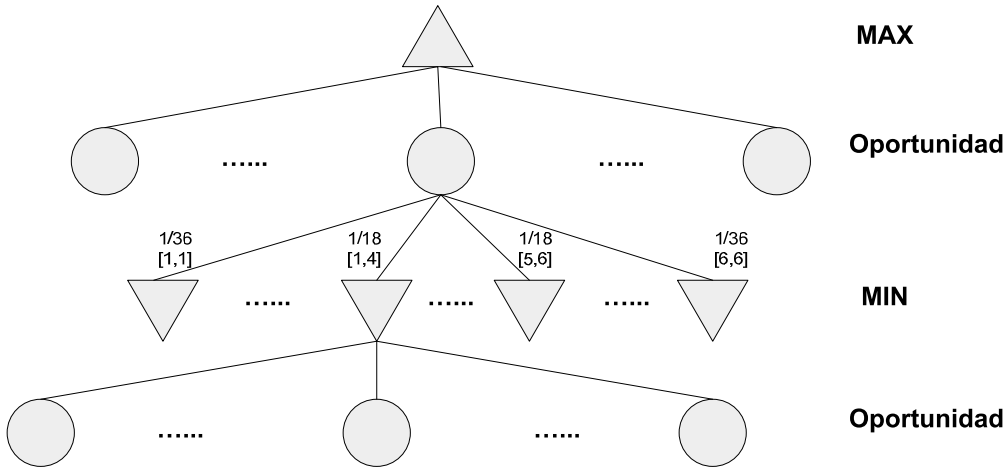


Fig. 5-9 Minimax con nodos de probabilidad para dos dados

En las etiquetas vemos representados entre corchetes los valores obtenidos en los dos dados y sobre este la probabilidad de la tirada. Los nodos de probabilidad también tienen un valor asociado que llamaremos **valor esperado** y que sustituye al valor de la función de evaluación de los nodos MAX y MIN. El valor esperado de un nodo de oportunidad se obtiene sumando los valores de la función de evaluación para cada posible tirada multiplicada por su probabilidad. Veamos un ejemplo simplificado. En la figura 5-10 observamos el despliegue de un nodo de probabilidad con dos hijos cuyas probabilidades de salir son 0,6 y 0,4 respectivamente. La función de evaluación de los nodos MIN hijo tienen un valor de 5 y 3, respectivamente. El valor esperado del nodo de probabilidad será por tanto:

$$(0.6 \times 5) + (0.4 \times 3) = 3{,}7$$

© RC Libros

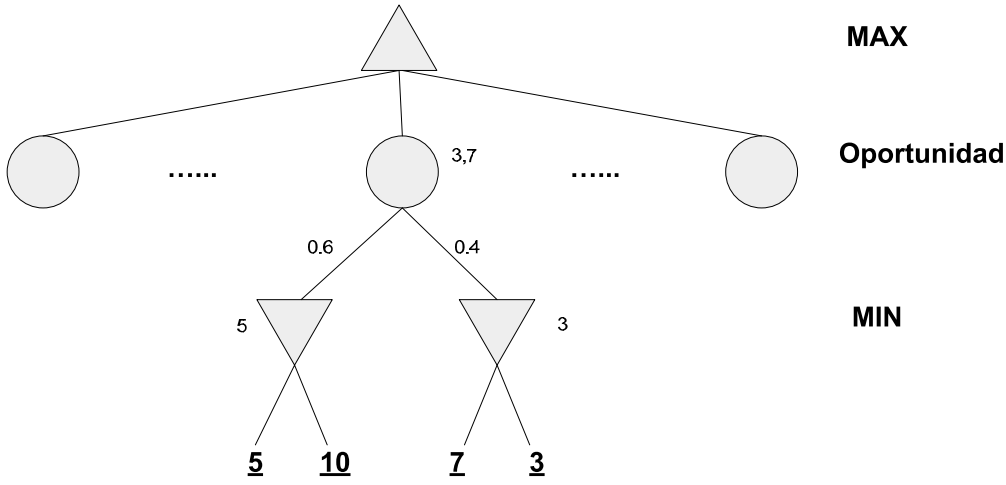


Fig. 5-10 Cálculo del valor esperado del nodo de probabilidad

Es importante en los juegos donde interviene el azar que la estimación de los valores de la función de evaluación sea lo más lineal posible. Esto quiere decir que si un nodo tiene una función de evaluación de 5 y otro de 10, es interesante que la jugada puntuada con el valor 10 sea, dentro de lo razonable, el doble de mejor que la de 5. En juegos donde no interviene el azar esto no es necesario; sin embargo, en los juegos donde intervienen probabilidades, sí es conveniente.

© RC Libros

6 Razonamiento

INTRODUCCIÓN

El razonamiento es un atributo atribuido a la especie humana y del que, en principio, no están dotados el resto de seres vivos. La capacidad de raciocinio implica que el ser humano ha de ser capaz de recordar información relacionada con los hechos sobre los que se quiere razonar, y por supuesto, la capacidad de relacionar esos hechos para inferir conocimiento nuevo en situaciones sobre las que no hemos tenido experiencias previas.

La capacidad de razonar nos permite, entre otras cosas, clasificar objetos y situaciones de forma que seamos capaces de reconocerlas y reaccionar ante ellas. Por ejemplo, si un amigo se compra un coche nuevo y nos lo enseña, no es necesario que nos explique que se trata de un coche. Somos capaces de reconocerlo aunque sea la primera vez que vemos ese coche concreto. Partiendo de ciertos hechos que ya conocemos acerca de los coches, como por ejemplo:

- Tienen ruedas.
- Tienen un parabrisas delantero y otro trasero con escobillas.
- Tienen ventanas y puertas laterales.
- Etc.

Somos capaces de clasificar lo que estamos viendo en una clase de objeto llamada “Coche”, aunque sea la primera vez que observamos ese modelo concreto. El proceso que nos ha llevado a reconocer el coche, es el mismo que permite al

mecánico descubrir una avería o a un médico diagnosticar una enfermedad a partir de unos síntomas.

SISTEMAS EXPERTOS

Desde los inicios de la Inteligencia Artificial se ha tratado de conseguir que los ordenadores razonen de forma similar a como lo hacen los humanos. Los primeros sistemas que trataban de conseguirlo se denominaron **Sistemas Basados en Conocimiento (SBC)**. Los SBC no persiguen ser capaces de razonar en cualquier circunstancia, sino que se circunscriben a un problema concreto. Por ejemplo, podríamos pensar en un SBC capaz de diagnosticar enfermedades respiratorias. Es lo que se denomina **dominio de aplicación** del SBC.

Un SBC se compone de dos partes principales: la **base de conocimiento** y el **motor de inferencia**. La base de conocimiento es la encargada de almacenar la información necesaria sobre el dominio de aplicación para poder ser capaz de llegar a conclusiones. La información almacenada en la base de conocimiento la proporciona un experto en el dominio de aplicación con el que estemos tratando. En nuestro ejemplo concreto para el diagnóstico de enfermedades respiratorias sería un médico especialista o un neumólogo el encargado de proveer dicha información. Desgraciadamente, la información que el especialista aporta no puede ser introducida en el sistema tal cual. Es necesario adaptarla y darle forma, o lo que es lo mismo, crear un modelo que el SBC pueda almacenar y comprender. La traducción de la información del especialista al modelo para el SBC corresponde a la figura del ingeniero del conocimiento, que suele ser un especialista en Inteligencia Artificial. No hay que perder de vista que el modelo de conocimiento que se almacena en el SBC es una abstracción de la información orientada a resolver un problema concreto.

El motor de inferencia es la parte que razona sobre la solución a un problema propuesto. A partir de una cuestión, el motor de inferencia busca información en la base de conocimiento y la relaciona hasta conseguir obtener una conclusión coherente con el problema que le hemos planteado.

Uno de los tipos más conocidos de SBC son los **sistemas expertos**, también llamados **sistemas basados en reglas**. La base de conocimiento de un sistema experto está compuesta de una **base de reglas** y de una **base de hechos** o **memoria de trabajo**. La base de reglas almacena **reglas** y la memoria de trabajo almacena **hechos**.

© RC Libros

Un hecho es toda aquella información que se da por cierta o demostrada dentro del dominio de aplicación. Pueden ser hechos previamente introducidos o hechos inferidos a partir de las reglas. Por otra parte, las reglas nos permiten relacionar hechos para inferir otros hechos nuevos.

Las reglas almacenadas en la base de reglas tienen la forma

Antecedente → Consecuente

Es muy similar a la construcción IF/THEN de los leguajes de programación. Veamos un ejemplo de regla para nuestro supuesto sistema experto en diagnóstico de problemas respiratorios:

R001: **SI** dificultad al respirar **Y** pitidos en los bronquios **ENTONCES** paciente con ataque de asma.

R002: **SI** dolor muscular **Y** tos **Y** fiebre **ENTONCES** paciente con gripe.

Cuando se da el antecedente obtenemos el consecuente, que es almacenado en la base de hechos a partir de ese momento. Por ejemplo, si se cumple la regla R001 es porque tenemos dos hechos previos que son: dificultad al respirar y pitidos en los bronquios. Como se cumple la regla, obtenemos un nuevo hecho (paciente con ataque de asma) que se almacenará en la base de hechos junto con todos los demás que ya estuvieran ahí almacenados.

Respecto al motor de inferencia de un sistema experto, hay dos estrategias para la búsqueda dentro de la base de conocimiento. La más utilizada es la llamada estrategia de **encadenamiento hacia adelante**. La otra se denomina estrategia de **encadenamiento hacia atrás**.

Un motor de inferencia con encadenamiento hacia adelante está gobernado por los antecedentes de las reglas. El motor identifica aquellas reglas en las que se cumple dicho antecedente para ejecutarlos. Un motor con encadenamiento hacia atrás parte del consecuente que se quiere demostrar o al que se quiere llegar y a partir de ahí va encadenando los antecedentes de las reglas. Es decir, funciona de forma contraria a un motor con encadenamiento hacia adelante.

Un motor con encadenamiento hacia adelante tiene dos fases bien diferenciadas: una fase de selección de reglas y otra de ejecución de las reglas seleccionadas. A su vez, la fase de selección de reglas está dividida en tres sub-fases: fase de restricción, fase filtrado y fase de resolución de conflictos.

© RC Libros

- Fase de selección: se seleccionan aquellas reglas que son de aplicación según las siguientes tres sub-fases.
    - Restricción: se trata de restringir el dominio de aplicación según la cuestión sobre la que se quiere razonar. Si tenemos un sistema experto capaz de diagnosticar enfermedades, pero sabemos que la enfermedad que tiene el paciente es respiratoria, no tiene sentido aplicar reglas que permitan el diagnóstico de enfermedades cardiacas; por lo tanto, el sistema experto restringirá la base de conocimiento a aquellas reglas de aplicación en el campo de las enfermedades respiratorias.
    - Filtrado: en esta fase se seleccionan las reglas que cumplen sus antecedentes según los datos que hay en la base de hechos. De esta fase de filtrado pueden salir seleccionadas una o varias reglas que son de aplicación (también podría ocurrir que no se seleccionara ninguna). Cuando hay varias reglas que son aplicables, decimos que es un **conjunto de conflicto**.
    - Resolución de conflictos: de la fase de filtrado pueden obtenerse más de una regla que podría ser aplicada. ¿Qué criterio seguir para seleccionar una u otra? Aquí hay varias estrategias posibles como puedan ser: ejecutar la primera regla del conjunto de conflicto, ejecutar la que incorpore más conocimiento a la base de hechos, establecer una prioridad en las reglas, ejecutar una regla arbitraria, usar una función heurística o ejecutar todas de forma exhaustiva para ver si alguna llega a la solución buscada.
- Fase de ejecución: básicamente consiste en añadir o modificar la base de conocimiento según los nuevos hechos (consecuente de la regla). Cuando encontramos el hecho que se estaba buscando, se para la ejecución del sistema. Puede ocurrir que lleguemos a un punto muerto en el que no haya más reglas aplicables, en cuyo caso podemos usar técnicas de backtracking para regresar a un punto anterior y seguir la búsqueda por otro camino.

Cuando en la fase de ejecución se ha aplicado una regla podemos optar por dejarla activa en la base de conocimiento o extraerla para que no se ejecute más.

Para ilustrar todo esto vamos a ver en detalle un ejemplo paso a paso. Vamos a definir una serie de reglas y hechos. Para no perder generalidad, los hechos serán nombrados con una letra mayúscula. Solo hay que asignar un significado concreto a cada letra. Por ejemplo, A = Dificultad al respirar. Las reglas de nuestro sistema experto de ejemplo serán las siguientes:

© RC Libros

```
R001: SI A Y  NO B ENTONCES D
R002: SI F Y D ENTONCES C
R003: SI M ENTONCES L
R004: SI H ENTONCES J
R005: SI F Y K ENTONCES B
R006: SI J Y C ENTONCES K
R007: SI C Y NO N ENTONCES NO A
```

Cuando al hecho le precede el operador NO en el antecedente, significa que la regla se activa cuando no exista dicho hecho en la base de hechos. Si es en el consecuente, lo que se hará es extraer el hecho que está negado de la base de hechos. Supondremos que la base de hechos inicial contiene los hechos H, F y A. El criterio para resolver qué regla se elegirá en la fase de resolución de conflicto será el orden de la regla. Una vez activada una regla en la fase de ejecución será retirada de la base de conocimiento. Nuestro objetivo es B, que es aquello que queremos demostrar a partir de los hechos y las reglas que hemos definido.

Paso 1. Base de hechos {H, F, A}

Conjunto de conflicto {R001, R004}

Regla aplicada: R001

Acción: se añade D a la base de hechos.

Paso 2. Base de hechos {H, F, A, D}

Conjunto de conflicto {R002, R004}

Regla aplicada: R002

Acción: se añade C a la base de hechos.

Paso 3. Base de hechos {H, F, A, D, C}

Conjunto de conflicto {R004, R007}

Regla aplicada: R004

Acción: se añade J a la base de hechos.

Paso 4. Base de hechos {H, F, A, D, C, J}

Conjunto de conflicto {R006, R007}

Regla aplicada: R006

Acción: se añade K a la base de hechos.

Paso 5. Base de hechos {H, F, A, D, C, J, K}

Conjunto de conflicto {R005, R007}

Regla aplicada: R005

Acción: se añade B a la base de hechos.

En este punto hemos añadido B a la base de hechos, por lo que hemos demostrado aquello que queríamos. B podría tener cualquier significado que le hubiéramos asignado de antemano; por ejemplo, B = Paciente tiene gripe.

SISTEMAS DIFUSOS

Los sistemas expertos son buenos trabajando con estados binarios, lo que quiere decir que pueden realizar un buen trabajo en situaciones en las que un hecho solo tiene la posibilidad de estar o no presente y las reglas o se cumplen o no se cumplen. Por ejemplo, en un sistema experto de diagnóstico médico puede tener datos en su base de conocimiento que permita discernir qué consecuencias tiene que un paciente presente pitidos en los bronquios al respirar pero, desgraciadamente, en el mundo en que vivimos no todo es o blanco o negro. Por ejemplo, ¿a partir de qué temperatura podemos considerar que un paciente tiene fiebre? ¿Debemos tratar igual a un paciente con una temperatura de 37,5º que a otro con 39,5º? Si

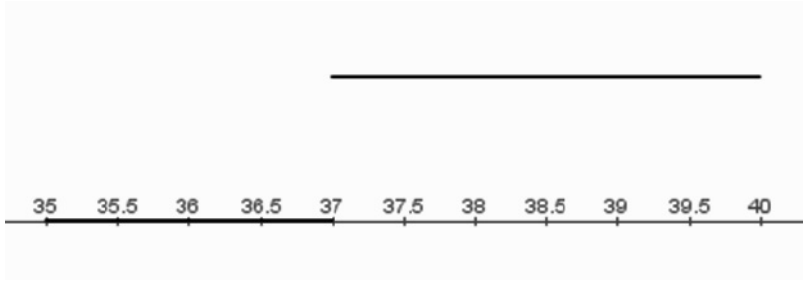
© RC Libros

consideramos que un paciente tiene fiebre a partir de 37º, uno con 37,1º activaría la regla de que el paciente tiene fiebre, pudiendo llevar al sistema experto a un diagnóstico equivocado.

En la lógica booleana que se utiliza en los sistemas expertos podríamos definir la función lógica que nos indica si un paciente tiene o no fiebre de la siguiente manera:

$$f(x) = \begin{cases} 0, & x \leq 37 \\ 1, & x > 37 \end{cases}$$

O lo que es lo mismo, f(x) será igual a 1 (verdadero) si x, que en este caso representa la temperatura corporal, tiene un valor superior a 37º y 0 en caso contrario. Gráficamente lo podríamos representar como:



También podríamos verlo desde un punto de vista conjuntista y definir el conjunto F como los valores de x para los que consideramos que el paciente tiene fiebre. En este caso, cuando x tome el valor 1 diremos que pertenece al conjunto F y si vale 0 diremos que no pertenece al conjunto F. Desde este punto de vista podemos decir que un valor de la variable x pertenece o no pertenece al conjunto, pero ¿y si hubiera una herramienta matemática que nos permita decir que el valor de la variable x tiene un grado de pertenencia p al conjunto F? Lo que buscamos es una herramienta que nos permita decir que si x = 37.1, pertenece al conjunto pero en un grado muy bajo, mientras que si x = 39, tiene un grado de pertenencia muy alto.

Conjuntos difusos

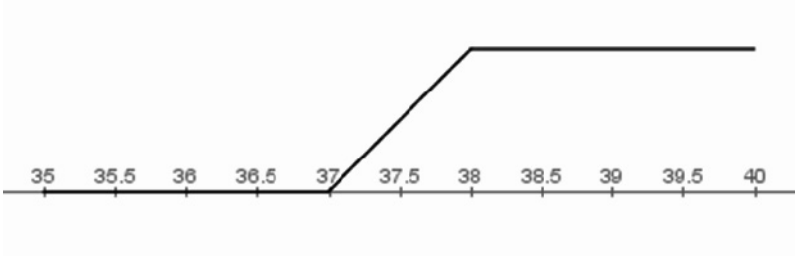
Lo que nos va a permitir trabajar con grado de pertenencia a un conjunto son los conjuntos difusos. Acabamos de ver que una función lógica puede ser vista como un conjunto. Para trabajar con sistemas difusos, podemos apoyarnos en la **lógica difusa (fuzzy logic)** que puede ser tratada también desde un punto de vista conjuntista, en cuyo caso hablamos de **conjuntos difusos (fuzzy sets)**. Aquí nos interesa trabajar bajo un punto de vista conjuntista.

Al igual que con los conjuntos normales, que a partir de ahora llamaremos **conjuntos nítidos** para distinguirlos, un elemento puede o no pertenecer al conjunto. Sin embargo, ahora ese elemento puede pertenecer en un grado concreto que está en el rango [0,1]. El grado con que un elemento x pertenece al conjunto A lo denotamos como $\mu_A(x)$. Por lo tanto, si x no pertenece en absoluto al conjunto A diremos que $\mu_A(x) = 0$, mientas que si $\mu_A(x) = 1$ querrá decir que x pertenece completamente al conjunto A. Un valor $\mu_A(x) = 0.2$ nos indica que x tiene un bajo grado de pertenencia al conjunto A mientras que $\mu_A(x) = 0.9$ nos habla de un alto grado de pertenencia.

Retomando el ejemplo de la temperatura corporal, podríamos definir la siguiente función de pertenencia al conjunto F de los valores de x que son considerados fiebre.

$$\mu(x)=\begin{cases} 0, & x\leq 37 \\ \dfrac{x-37}{38-37}, & x\epsilon(37,38) \\ 1, & x>38 \end{cases}$$

Gráficamente tiene el siguiente aspecto:



Para los valores de x entre 37 y 38, el grado de pertenencia de x está definido por la ecuación $\frac{x-37}{38-37}$, que es la ecuación de una recta. Si le damos el valor 37.5 a x tenemos:

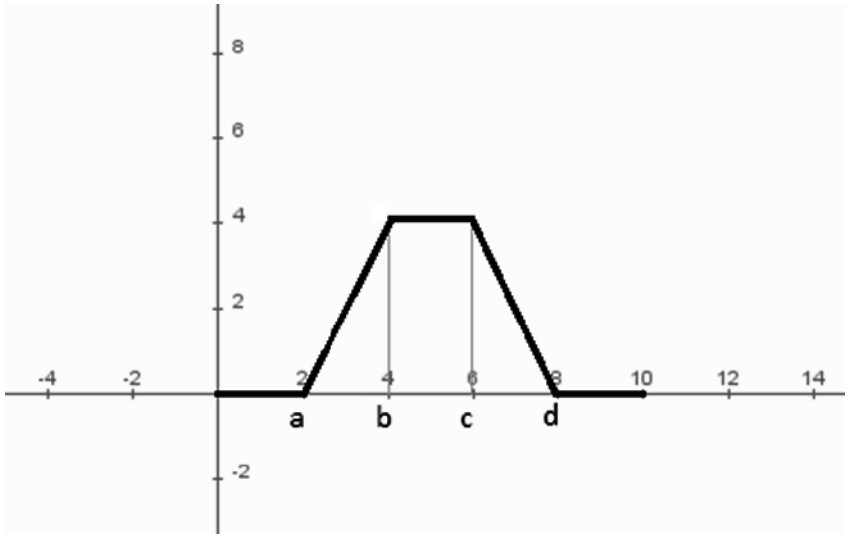
$$\mu_F(x) = \frac{37.5 - 37}{38 - 37} = 0.5$$

Lo que indica que tiene un grado de pertenencia medio al conjunto F. Para valores superiores a 38 el grado de pertenencia es siempre 1, lo que indica una pertenencia total a F.

Hay varios tipos de funciones de pertenencia pero son muy habituales las que tienen forma trapezoidal o de medio trapecio como el ejemplo que estamos viendo.

© RC Libros

En general, para obtener la función de pertenencia podemos usar el siguiente esquema:



Siendo la función de pertenencia:

$$\mu(x) = \begin{cases} 0, & (x \leq a) o\ (x \geq d) \\ \dfrac{x-a}{b-a}, & x \epsilon (a, b] \\ 1, & x \epsilon (b, c) \\ \dfrac{d-x}{d-c}, & x \epsilon [c, d) \end{cases}$$

En los ejemplos que usaremos en este capítulo nos basaremos en este tipo de función de pertenencia.

Una vez definido el concepto de pertenencia a un conjunto difuso, nos resta definir las funciones que podemos realizar sobre estos, que son las mismas de las que disponemos en los conjuntos nítidos. En concreto, vamos a definir las operaciones de complementación, unión e intersección.

En los conjuntos nítidos definimos la **función de complementación** N(x) como aquella que toma el valor 1 cuando x = 0 y toma el valor 0 cuando x = 1, es decir, invierte el valor de x. La denotaremos como $\mu_A{}^C(x)$. En los conjuntos difusos ocurre igual cuando x = 0 y x = 1, pero ¿qué ocurre para cualquier otro valor de x?

© RC Libros

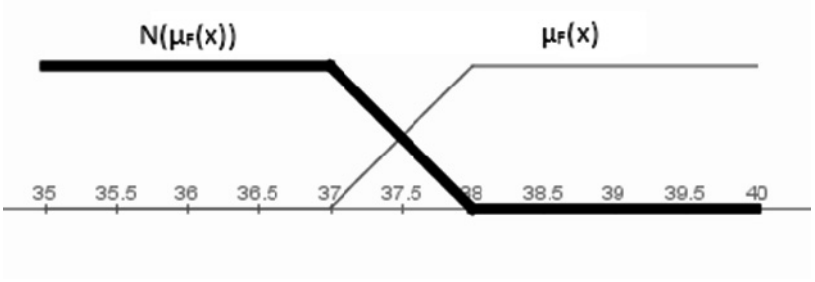
Podríamos definir diferentes funciones de complementación siempre que cumplieran una serie de condiciones matemáticas. Por simplicidad aquí elegiremos la más sencilla. En general, si tenemos un valor a de pertenencia a un conjunto difuso diremos que su complementario es 1-a.

$$N(a) = 1 - a$$

En el ejemplo de la temperatura corporal habíamos definido $\mu_F(x)$ como la función de pertenencia de los valores de x que se consideraban fiebre. Su función complementaria será por lo tanto:

$$\mu_{F^C}(x) = \begin{cases} 1, & x \leq 37 \\ 1 - \dfrac{x-37}{38-37}, & x\epsilon(37,38) \\ 0, & x > 38 \end{cases}$$

En la siguiente gráfica podemos ver la representación de $N(\mu_F(x))$ comparada con $\mu_F(x)$.



Consideremos ahora la función de unión de dos conjuntos nítidos. Dados dos conjuntos A y B diremos que un elemento x pertenece a la unión de ambos conjuntos (lo escribimos como AUB) si x pertenece al conjunto A o x pertenece al conjunto B. En el caso difuso ocurre la misma circunstancia, pero denotaremos a la función de pertenencia resultante como $\mu_{AUB}(x)$. Al igual que definimos la función N(x) para la complementación, podemos definir una función S(A, B) tal que $S(\mu_A, \mu_B)=\mu_{AUB}(x)$.

Al igual que la función N(x), a la función S(x), a la que llamaremos **t-conorma**, vamos a exigirle una serie de propiedades matemáticas (conmutatividad, asociatividad, monotonía y elemento neutro). Podríamos definir distintas funciones de t-conorma que cumplan estas propiedades, pero de nuevo nos quedaremos con la más sencilla que se define como:

© RC Libros

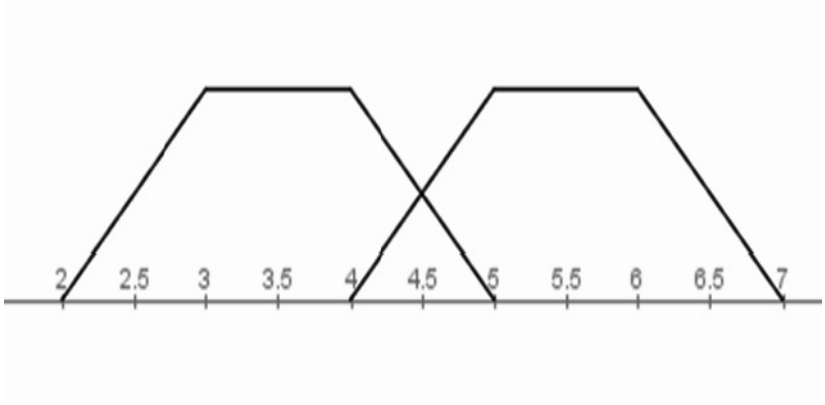
$$S(a,b) = \max(a,b)$$

Vamos a ilustrar la operación de t-conorma que acabamos de definir con un ejemplo. Sean los dos conjuntos difusos A y B siguientes:

$$\mu_A(x) = \begin{cases} \dfrac{x-2}{3-2}, & x\epsilon(2,3] \\ 1, & x\epsilon(3,4) \\ \dfrac{5-x}{5-4}, & x\epsilon[4,5) \end{cases}$$

$$\mu_B(x) = \begin{cases} \dfrac{x-4}{5-4}, & x\epsilon(4,5] \\ 1, & x\epsilon(5,6) \\ \dfrac{7-x}{7-6}, & x\epsilon[6,7) \end{cases}$$

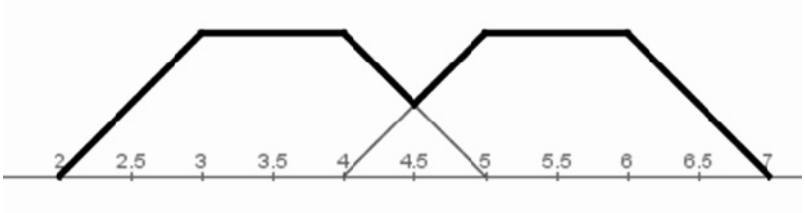
Cuya representación gráfica es:



Aplicando la t-conorma definida más arriba obtendríamos la siguiente función de pertenencia.

$$\mu_{AUB}(x) = \begin{cases} \dfrac{x-2}{3-2}, & x\epsilon(2,3] \\ 1, & x\epsilon(3,4) \\ \dfrac{5-x}{5-4}, & x\epsilon[4,4.5) \\ \dfrac{x-4}{5-4}, & x\epsilon(4.5,5] \\ 1, & x\epsilon(5,6) \\ \dfrac{7-x}{7-6}, & x\epsilon[6,7) \end{cases}$$

Que se corresponde con los trazos más gruesos de la gráfica siguiente.



La última operación que vamos a definir es la intersección. Dados dos conjuntos nítidos A y B, la intersección de ambos se define como aquellos elementos que se encuentran a la vez en el conjunto A y en el conjunto B. En los conjuntos difusos la intersección de dos conjuntos, que se escribe $\mu_{A\cap B}(x)$, queda definida por una función T(x) que de nuevo ha de cumplir las mismas propiedades matemáticas que exigíamos a la unión. De entre aquellas posibles definiciones de T(x) nos quedamos con la más simple.

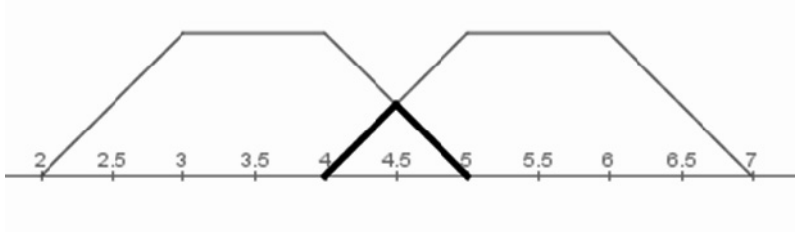
$$T(a,b) = \min(a,b)$$

A esta función la denominamos **t-norma**. Retomando el ejemplo de los dos conjuntos definidos en la figura anterior, la intersección de A y B sería.

© RC Libros

$$f(x) = \begin{cases} \dfrac{x-4}{5-4}, & x\epsilon(4, 4.5] \\ \dfrac{5-x}{5-4}, & x\epsilon(4.5, 5) \end{cases}$$

Que se corresponde con los trazos más gruesos de la gráfica siguiente.



Conviene no perder de vista que si x = 4.5 quiere decir que tiene un grado de pertenencia de 0.5 al conjunto A y también de 0.5 al conjunto B, es decir, pertenece a ambos conjuntos en igual medida.

Inferencia difusa

Definidos los conjuntos difusos y sus operaciones, vamos a ver cómo utilizarlos para construir sistemas difusos. Estos sistemas son muy utilizados en tareas de modelización y control. Los sistemas difusos no son más que sistemas expertos definidos en términos de conjuntos difusos en vez de como conjuntos nítidos. Como tal, dispondremos de reglas cuyas variables se valorarán según su grado de pertenencia a los conjuntos usados en los antecedentes. El consecuente también es una variable difusa.

Un ejemplo clásico de sistema de control difuso es el del termostato de un horno. En el sistema de control del termostato, siendo x la temperatura leída por el sensor del horno, podríamos tener reglas como las siguientes.

```
R001: SI μfrio ENTONCES …
R002: SI μcaliente ENTONCES …
R003: SI μmuy_caliente ENTONCES …
```

Donde los antecedentes de las reglas son funciones de pertenencia a un conjunto difuso similares a las que hemos visto en el apartado anterior. Ahora, en vez de dar valores concretos para las variables del antecedente, podemos utilizar variables con un valor más vago como frío, caliente o muy caliente. Dependiendo del grado de pertenencia de la variable (en este caso la temperatura) a cada uno de los conjuntos difusos que hayamos definido, el consecuente de la regla se activará en un mayor o menor grado. Por ejemplo, podría darse el caso de que la variable temperatura tuviera los grados de pertenencia $\mu_{frio} = 0.2$, $\mu_{caliente}=0.8$ y $\mu_{muy_caliente} = 0$ (la suma de todos los grados de pertenencia tiene que valer 1). Bajo estas condiciones el horno podría activar la resistencia que usa para generar calor con un grado inferior que si se dieran las siguientes condiciones: $\mu_{frio} = 0.8$, $\mu_{caliente} = 0.2$ y $\mu_{muy_caliente} = 0$, en las que, evidentemente, tendría que usar más energía para acelerar el aumento de temperatura.

A las variables que toman como posibles valores palabras del lenguaje natural (como frío, caliente o muy caliente) las llamamos variables lingüísticas. Para definir completamente una variable lingüística, debemos definir los siguientes atributos:

- **Nombre de la variable lingüística**: Nombre que describe la variable. Por ejemplo, Temperatura.
- **Valores lingüísticos**: Posibles valores que puede tomar la variable. Por ejemplo, frío, caliente o muy caliente.
- **Universo de discurso**: Describe los posibles valores que puede tomar la variable. Por ejemplo, si el sensor que mide la temperatura del horno tiene un rango de trabajo de entre 0 y 300 grados, el universo de discurso de la variable Temperatura será [0..300].
- **Función semántica**: Es la función que asigna significado a los posibles valores. Es la función de pertenencia que hemos visto en el apartado anterior.

No hay un solo método de inferencia difusa. Nosotros utilizaremos uno que se llama método de Mandani. Para ver los pasos de los que se compone, vamos a partir de un ejemplo que vamos a ir desarrollando.

Vamos a implementar un sistema de control difuso para el termostato de un horno. El horno dispone de un sensor que hace una lectura de la temperatura cada minuto y una resistencia para calentar el horno cuya temperatura controlaremos mediante un voltaje de control. Nuestro sistema va a tener dos variables de entrada y una de salida. Definimos a continuación la primera variable.

- Nombre de la variable lingüística: error de temperatura. Nos indica la diferencia entre la temperatura actual leída por el sensor y la temperatura deseada.

© RC Libros

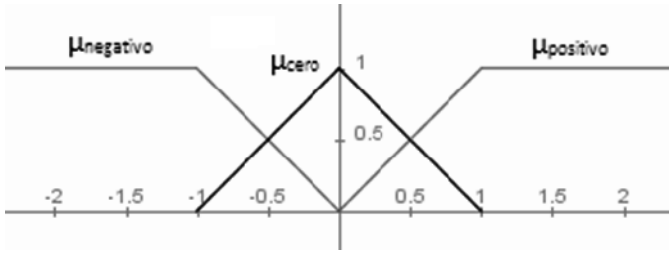
- Valores lingüísticos: Negativo, Cero, Positivo. Si es negativo, la temperatura actual se halla por debajo de la deseada.
- Universo de discurso: Consideraremos como universo de discurso el conjunto de los números reales. No impondremos, en principio, ningún rango de valores válidos.
- Funciones semánticas:

$$\mu_{negativo}(x) = \begin{cases} 1, & x < -1 \\ -x, & x\epsilon(-1,0) \end{cases}$$

$$\mu_{cero}(x) = \begin{cases} x+1, & x\epsilon(-1,0) \\ -x+1, & x\epsilon(0,1) \end{cases}$$

$$\mu_{positivo}(x) = \begin{cases} x, & x\epsilon(0,1) \\ 1, & x > 1 \end{cases}$$

Podemos representar las variables gráficamente.



La segunda variable de entrada queda definida de la siguiente manera:

- Nombre de la variable lingüística: Incremento de temperatura. Nos indica la diferencia entre la temperatura actual leída por el sensor y la anterior lectura. Nos indica la velocidad a la que varía la temperatura en un minuto.

© RC Libros

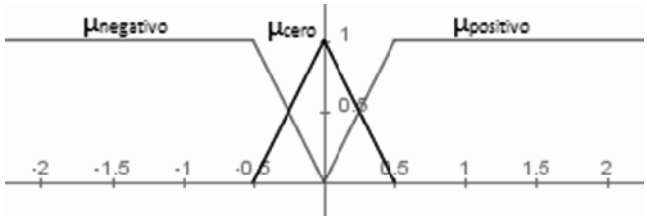
- Valores lingüísticos: Negativo, Cero, Positivo. Si es negativo, la temperatura actual ha descendido.
- Universo de discurso: Consideraremos como universo de discurso el conjunto de los números reales. No impondremos, en principio, ningún rango de valores válidos.
- Funciones semánticas:

$$\mu_{negativo}(x) = \begin{cases} 1, & x < -0.5 \\ -x, & x\epsilon(-0.5, 0) \end{cases}$$

$$\mu_{cero}(x) = \begin{cases} x + 1, & x\epsilon(-0.5, 0) \\ -x + 1, & x\epsilon(0, 0.5) \end{cases}$$

$$\mu_{positivo}(x) = \begin{cases} x, & x\epsilon(0, 0.5) \\ 1, & x > 0.5 \end{cases}$$

Gráficamente, representamos las variables en la siguiente figura.



Por último, nos queda definir la variable de salida.

- Nombre de la variable lingüística: Voltaje de control. En esta variable indicamos la cantidad de voltaje necesario a aplicar en la resistencia para controlar su temperatura.
- Valores lingüísticos: Disminuir, Mantener, Aumentar. Si toma el valor Aumentar, quiere decir que la resistencia del horno aumentará su temperatura.

© RC Libros

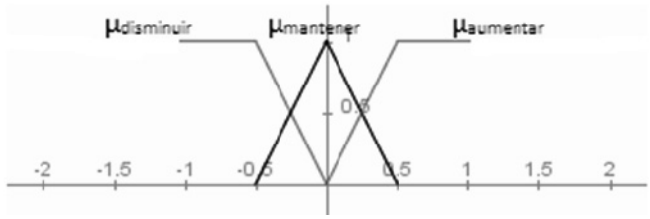
- Universo de discurso: Consideraremos como universo de discurso el conjunto de los números reales que van desde -1 a +1.
- Funciones semánticas:

$$\mu_{disminuir}(x) = \begin{cases} 1, & x\epsilon(-1,-0.5) \\ -x, & x\epsilon(-0.5,0) \end{cases}$$

$$\mu_{mantener}(x) = \begin{cases} x+1, & x\epsilon(-0.5,0) \\ -x+1, & x\epsilon(0,0.5) \end{cases}$$

$$\mu_{aumentar}(x) = \begin{cases} x, & x\epsilon(0,0.5) \\ 1, & x\epsilon(0.5,1) \end{cases}$$

Gráficamente, representamos las variables en la siguiente figura.



El siguiente paso es definir las reglas. Llamaremos E a la variable Error, ΔT al incremento de la temperatura y VC al voltaje de control. Para nuestro termostato vamos a definir las siguientes:

R001: **SI** $E=\mu_{negativo}$ **Y** $\Delta T=\mu_{negativo}$ **ENTONCES** $VC=\mu_{aumentar}$

R002: **SI** $E=\mu_{negativo}$ **Y** $\Delta T=\mu_{cero}$ **ENTONCES** $VC=\mu_{aumentar}$

R003: **SI** $E=\mu_{negativo}$ **Y** $\Delta T=\mu_{positivo}$ **ENTONCES** $VC=\mu_{mantener}$

R004: **SI** $E=\mu_{cero}$ **Y** $\Delta T=\mu_{negativo}$ **ENTONCES** $VC=\mu_{aumentar}$

R005: **SI** $E=\mu_{cero}$ **Y** $\Delta T=\mu_{cero}$ **ENTONCES** $VC=\mu_{mantener}$

R006: **SI** $E=\mu_{cero}$ **Y** $\Delta T=\mu_{positivo}$ **ENTONCES** $VC=\mu_{disminuir}$

© RC Libros

R007: **SI** $E=\mu_{positivo}$ **Y** $\Delta T=\mu_{negativo}$ **ENTONCES** $VC=\mu_{mantener}$

R008: **SI** $E=\mu_{positivo}$ **Y** $\Delta T=\mu_{cero}$ **ENTONCES** $VC=\mu_{disminuir}$

R009: **SI** $E=\mu_{positivo}$ **Y** $\Delta T=\mu_{positivo}$ **ENTONCES** $VC=\mu_{disminuir}$

Definidas las variables y las reglas, ya podemos hacer la inferencia dados unos valores concretos de entrada. Siguiendo con el ejemplo del termostato, supondremos que se dan los siguientes valores en las entradas: E = -0.5 y ΔT = 0.7. Los pasos para realizar la inferencia y obtener el valor al que hay que poner la variable VC son los siguientes:

Paso 1. Evaluación del antecedente de cada regla

Dados los valores de las variables de entrada debemos evaluar cada una de las reglas y ver cuáles se activan y cuáles no. Si alguno de los antecedentes de las reglas implicadas está negado (un NOT lógico), realizaremos una operación de complementación sobre él. Tras evaluar cada uno de los antecedentes, si hay una operación lógica O (OR) entre ellos, realizaremos una t-conorma (por ejemplo, el máximo de ambos valores). Si se trata de una operación Y (AND) aplicaremos una t-norma (por ejemplo, el mínimo de ambos valores).

Según los valores de entrada que tenemos en el termostato obtenemos los siguientes valores:

Para la variable E, el valor 0.5 puede activar aquellas reglas en las que $E = \mu_{negativo}$ y $E = \mu_{cero}$, siempre que se cumpla el resto del antecedente, ya que

$\mu_{negativo}(-0.5) = 0.5$

$\mu_{cero}(-0.5) = 0.5$

$\mu_{positivo}(-0.5) = 0$

Para la variable ΔT tenemos que

$\mu_{negativo}(0.7) = 0$

$\mu_{cero}(0.7) = 0$

© RC Libros

$\mu_{positivo}(0.7) = 1$

Por lo tanto, al tratarse de dos variables conectadas mediante el operador lógico Y (t-norma) las reglas que se activarán son:

R003: **SI** $E=\mu_{negativo}$ **Y** $\Delta T=\mu_{positivo}$ **ENTONCES** $VC=\mu_{mantener}$

R006: **SI** $E=\mu_{cero}$ **Y** $\Delta T=\mu_{positivo}$ **ENTONCES** $VC=\mu_{disminuir}$

Observemos cómo ambas reglas por separado pueden tener distintos consecuentes.

En la regla R003 tenemos pues el consecuente formado por los valores E = 0.5 y ΔT = 1. Como se trata de una t-norma nos quedamos con el valor mínimo:

$$resultado = \min(0.5, 1) = 0.5$$

Con la regla R006 ocurre lo mismo ya que E = 0.5 y ΔT = 1, por lo que también obtenemos el resultado 0.5.

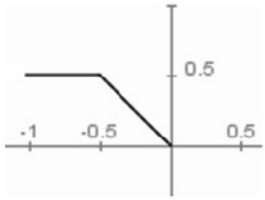
Paso 2. Obtener la conclusión de cada regla

Para cada regla implicada en el paso anterior, debemos calcular su consecuente. En lógica difusa, el operador de implicación (ENTONCES) es el mínimo entre el valor del consecuente y el valor del antecedente (puede haber otros operadores, pero por simplicidad nos quedaremos con este). En este caso como el antecedente tiene el valor 0.5, la función de pertenencia de la variable VC queda truncada a este valor, es decir, las variables $\mu_{disminuir}$ y $\mu_{mantener}$ pasan a tener la siguiente función de pertenencia.

Para el caso de $\mu_{disminuir}$ tenemos:

$$\mu_{disminuir}(x) = \begin{cases} 0.5, & x\epsilon(-1, -0.5) \\ -x, & x\epsilon(-0.5, 0) \end{cases}$$

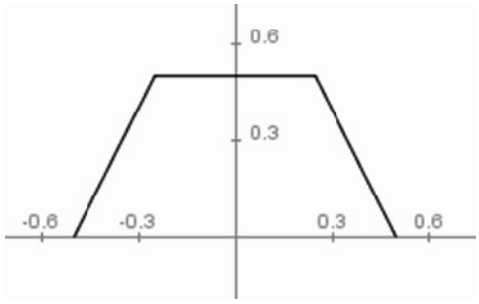
Gráficamente es:



Para el caso de $\mu_{mantener}$ tenemos:

$$\mu_{mantener}(x) = \begin{cases} x+1, & x\epsilon(-0.5,-0.25) \\ 0.5, & x\epsilon(-0.25,0.25) \\ -x+1, & x\epsilon(0.25,0.5) \end{cases}$$

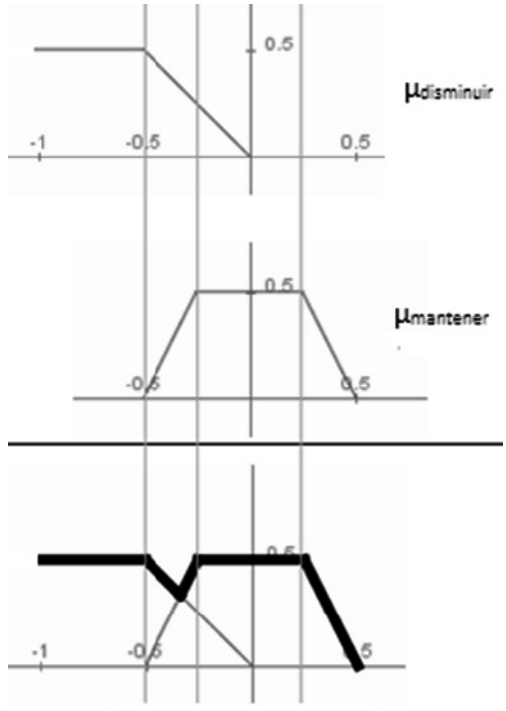
Gráficamente se representaría así:



Paso 4. Agregar conclusiones

Hemos obtenido los dos consecuentes tras aplicar las reglas R003 y R006 en forma de función de pertenencia y también los hemos representado gráficamente. Para obtener un consecuente final, hemos de agregar ambas conclusiones en una sola. Para ello, usamos los combinamos como una suma difusa, o lo que es lo mismo, aplicamos una t-conorma o un operador O (OR) difuso. Como la t-conorma que estamos usando es la función máxima obtendremos el siguiente consecuente tras aplicar la t-conorma.

© RC Libros



Cuya función de pertenencia es ahora

$$\mu_{resultado}(x) = \begin{cases} 0.5, & x\epsilon(-1,-0.5) \\ -x, & x\epsilon(-0.5,-0.33) \\ x+1, & x\epsilon(-0.33,-0.25) \\ 0.5, & x\epsilon(-0.25,0.25) \\ -x+1, & x\epsilon(0.25,0.5) \end{cases}$$

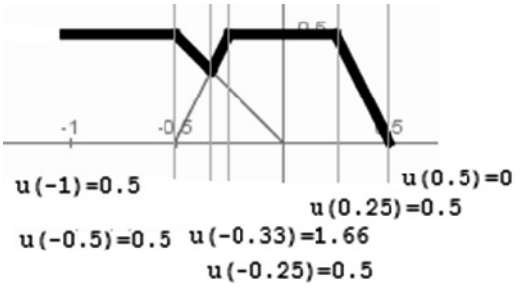
Paso 5. Nitidificación

Hemos obtenido el consecuente, pero necesitamos traducirlo a un valor concreto que será al voltaje de control a aplicar a la resistencia del horno. El proceso de transformar un consecuente difuso en un valor final se llama nitidificación. Para calcular el valor, hay diversos métodos, pero el más usado consiste en hallar el centroide que calcula el centro del área definida por el conjunto difuso formado por el consecuente obtenido en el paso anterior. Para calcularlo usamos la siguiente fórmula.

© RC Libros

$$C = \frac{\sum_{i=1} x_i \times u(x_i)}{\sum_{i=1} u(x_i)}$$

Donde x_i representa los puntos con lo que hay un cambio de tendencia en la función y $u(x_i)$ es el valor de la función en dicho punto. En nuestro ejemplo se calcularía así:



$$C = \frac{(-1 \times 0.5) + (-0.5 \times 0.5) + (-0.33 \times 1.66) + (-0.25 \times 0.5) + (0.25 \times 0.5) + (0.5 \times 0)}{0.5 + 0.5 + 1.66 + 0.5 + 0.5 + 0}$$

$$c = 0.96$$

El valor al que tendremos que poner nuestra variable VC o voltaje de control es 0.96.

© RC Libros

7 Aprendizaje

INTRODUCCIÓN

Desde que nacemos, nuestros sentidos nos ponen en contacto con una realidad cambiante y a la que debemos adaptarnos para poder sobrevivir. El aprendizaje es una facultad que nos permite, a partir de la información que obtenemos del exterior, desarrollar o modificar hábitos, conductas o habilidades para adaptarnos mejor al entorno.

En Inteligencia Artificial, la capacidad de aprendizaje está mucho más limitada que la de los seres humanos. Tareas que a priori pueden parecernos sencillas, como reconocer la cara de un conocido, es una tarea tremendamente compleja para un ordenador. De hecho, seremos capaces de reconocer esa cara aunque la persona se ponga unas gafas de sol, se deje barba o se rodee la boca con una bufanda. Nuestro cerebro seguirá reconociendo la cara. Para un ordenador, sin embargo, cualquier cambio, incluso gestual, complica el proceso de reconocimiento.

La neurociencia actual demuestra que el cerebro está especialmente dotado para el reconocimiento de patrones, y básicamente es así como asociamos las imágenes que recibimos visualmente con las personas u objetos que conocemos. En Inteligencia Artificial se llevan años aplicando con éxito técnicas basadas en el funcionamiento del cerebro, concretamente en el funcionamiento de las neuronas. Son las llamadas redes neuronales, a las que dedicaremos la segunda parte de este capítulo.

El proceso de clasificar cosas, ya sean caras de personas o cualquier otro elemento o concepto, es una tarea muy interesante con aplicaciones en múltiples campos. A modo de ejemplo podemos nombrar el reconocimiento óptico de caracteres (OCR) que permite a una máquina leer textos, matrículas de vehículos, etc., o los programas clasificadores de texto, como puedan ser los programas anti-spam que usan nuestros servidores de correo electrónico. Estos programas son capaces de, a través de un entrenamiento y un aprendizaje continuo, clasificar cada vez con mayor precisión los correos recibidos para decidir si son correos basura o no.

Hay diferentes técnicas que nos permiten realizar clasificaciones. Además de las redes neuronales, los clasificadores basados en modelos probabilísticos, como los clasificadores bayesianos, dan muy buenos resultados. Con este último tipo de clasificadores vamos a comenzar este capítulo.

CLASIFICACIÓN PROBABILÍSTICA

Los modelos basados en probabilidad son muy utilizados en diferentes problemas de clasificación y aprendizaje. Quizás, las redes bayesianas sean las más conocidas y utilizadas. En este capítulo vamos a presentar un modelo probabilístico para el problema de la clasificación llamado **clasificador bayesiano ingenuo** o **naive bayes classifier**.

Este método de clasificación está construido sobre modelos probabilísticos y, en concreto, hace uso del teorema de Bayes sobre las probabilidades condicionadas. Los conocimientos sobre probabilidad necesarios para comprender y aplicar esta técnica no son especialmente complejos, aunque convendrá dar un repaso a lo más básico antes de continuar.

Un poco de probabilidad

Cuando lanzamos una moneda al aire varias veces, de forma más o menos intuitiva, podemos prever que aproximadamente la mitad de la veces debería salir cara. Decimos que hay una probabilidad del 50% de obtener cara. Si ahora lanzamos un dado, la probabilidad de sacar, por ejemplo, un 1, ya no es tan evidente como en el caso de la moneda. Aún podríamos complicarlo más:

© RC Libros

¿Cuál es la probabilidad de sacar un número par?

¿Y la probabilidad de obtener un 1 o un 6?

Cada vez que lanzamos un dado de seis caras podemos obtener seis resultados distintos. Podemos sacar un 1, un 2, un 3, un 4, un 5 o un 6. El conjunto de posibles resultados es lo que llamamos espacio muestral, que denotaremos con la letra griega Ω. También decimos que el tamaño, o el cardinal, de este espacio muestral es 6, porque hay seis resultados posibles. Definiremos finalmente un suceso como un subconjunto del espacio muestral. Por ejemplo, el suceso de sacar número par al lanzar un dado es:

$A=\{2,4,6\}$

El cardinal de P es 3, ya que el suceso A es un subconjunto del espacio muestral formado por 3 resultados posibles. Lo podemos escribir como card(A) = 3.

El suceso de sacar un valor igual o superior a 3 lo definimos como:

$B = \{3,4,5,6\}$

En este caso card(B) = 4.

¿Y cuál sería la probabilidad de sacar un número impar o de sacar un valor estrictamente menor que 3? Son los sucesos justamente contrarios a A y B. Los llamamos sucesos complementarios, y serían:

$A^c = \{1,3,5\}$

$B^c = \{1,2\}$

Ahora card(A^c) = 3 que coincide con card(Ω)-card(A). Además, card(B^c) = 2 que también coincide con card(Ω)-card(B).

A partir de dos sucesos diferentes, podemos construir otros utilizando el operador de unión (U) y el operador de intersección (∩). La intersección de dos sucesos es otro suceso favorable a ambos a la vez. Por ejemplo:

$A \cap B =\{4,6\}$

Son aquellos valores que pertenecen al suceso A y al suceso B a la vez, es decir, son pares y mayores que 3.

© RC Libros

La unión de dos sucesos es otro favorable al primer suceso, al segundo o a ambos a la vez. Por ejemplo:

A U B = {2,3,4,5,6}

Son aquellos valores que pertenecen a A, a B o a ambos a la vez. O sea, está compuesto por aquellos valores pares y a la vez por aquellos valores mayores o iguales que 3.

Pero ¿cuál es la probabilidad de sacar un número par o de sacar un número mayor o igual que 3? La probabilidad de un suceso es la tendencia que tiene ese suceso a darse y su valor siempre estará comprendido entre 0 y 1. Podemos calcularlo usando la **regla de Laplace**.

$$P(suceso) = \frac{card(suceso)}{card(\Omega)}$$

O lo que es lo mismo, el número de resultados favorables al suceso dividido entre el número total de posibles resultados. Ahora podemos contestar a las preguntas que nos estábamos haciendo.

$$P(A) = \frac{3}{6} = 0.5$$

$$P(B) = \frac{4}{6} = 0.66$$

Observamos que el valor de la probabilidad de ambos sucesos está entre 0 y 1. Para pasarlo a un valor porcentual simplemente multiplicamos por 100. Así, la probabilidad de A es del 50%, mientras que la probabilidad de B es del 66%. Como la probabilidad de un suceso está acotada entre 0 y 1, la probabilidad complementaria de un suceso A será 1-P(A). Por ejemplo:

$$P(B^c) = 1 - 0.66 = 0.33$$

También podemos preguntarnos cuál es la probabilidad de que el valor sea par y a la vez mayor o igual que 3.

© RC Libros

$$P(A \cap B) = \frac{2}{6} = 0.33$$

Ahora que tenemos una noción de cómo calcular la probabilidad de un suceso, vamos a introducir el concepto de **probabilidad condicionada**. Decimos que la probabilidad condicionada del suceso A sabiendo que se da el suceso B es:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Despejando de la fórmula, podemos comprobar que se da la siguiente igualdad:

$$P(A \cap B) = P(A|B) \times P(B) = P(B|A) \times P(A)$$

Veámoslo con un sencillo ejemplo que nos facilitará entender el concepto. Retomemos los sucesos A y B que son obtener un valor par y obtener un valor mayor o igual a 3.

A = {2,4,6}

B = {3,4,5,6}

¿Cuál es la probabilidad de haber sacado un número par tras el lanzamiento de un dado sabiendo que dicho valor es mayor o igual a 3? O dicho de otra forma, ¿cuál es la probabilidad de que se dé A sabiendo que se da B? Esto lo expresamos como P(A|B) o probabilidad de A condicionada a B. Vamos a calcularla con la fórmula de que hemos visto antes.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{0.33}{0.66} = 0.5$$

Ya habíamos calculado más arriba los valores de $P(A \cap B) = 0.33$ y $P(B) = 0.66$, así que solo hemos tenido que sustituir los valores en la fórmula para obtener el valor 0.5, que puede expresarse en forma de fracción como $1/2$. Tiene todo el sentido haber obtenido este valor, ya que de los posibles 4 valores de B solo 2 (la mitad) son de A (son pares).

Cuando se dan dos sucesos, no siempre ocurre que cuando se da uno de ellos se vea afectada la probabilidad del otro. Vamos a exponer un ejemplo un poco exagerado pero que nos permite entender este concepto fácilmente. Imaginemos que queremos calcular la probabilidad de sacar un valor par al lanzar un dado

© RC Libros

sabiendo que hoy el día está soleado. Esto es, P(Par|Soleado). ¿Cuál será la probabilidad de P(Par)? Es evidente que el hecho de que el día esté soleado no influye para nada en la probabilidad de sacar número par, es decir, que P(Par|Soleado) = P(Par). Cuando esto ocurre decimos que ambos sucesos son independientes y se cumple que:

$$P(A \cap B) = P(A) \times P(B)$$

$$P(A|B) = P(A)$$

$$P(B|A) = P(B)$$

Ya estamos casi preparados para enfrentarnos al teorema de Bayes, que es la base para construir los clasificadores que vamos a describir en este capítulo, pero antes nos falta presentar un último concepto: **el teorema de las probabilidades totales**. Este teorema dice que si dividimos el espacio muestral en n partes tal que $\Omega = \{A_1, A_2, A_3, \ldots, A_n\}$, y siendo B cualquier suceso, tenemos que

$$P(B) = P(B \cap A_1) + P(B \cap A_2) + \cdots + P(B \cap A_n)$$

Que según vimos antes, podemos reescribir como:

$$P(B) = P(B|A_1) \times P(A_1) + P(B|A_2) \times P(A_2) + \cdots + P(B|A_n) \times P(A_n)$$

Siguiendo con nuestro ejemplo del dado, vamos a dividir el espacio muestral en dos. Vamos a considerar por un lado, los números pares y, por otro, los impares. Hay que tener en cuenta que la suma de todas las partes debe dar lugar a todo el espacio muestral. Así que tenemos:

$$A_1 = \{1,3,5\}$$

$$A_2 = \{2,4,6\}$$

$$B = \{3,4,5,6\}$$

¿Cuál es la probabilidad de B si sabemos que A_1 y A_2 forman la totalidad del espacio muestral? Aplicando la fórmula tenemos:

$$P(B) = P(B \cap A_1) + P(B \cap A_2) = \frac{2}{6} + \frac{2}{6} = \frac{4}{6} = 0.66$$

Que coincide con lo que habíamos calculado al principio.

© RC Libros

Ahora sí estamos listos para entender el famoso **teorema de Bayes**. Este nos dice que si tenemos el espacio muestral dividido en n partes tal que $\Omega = \{A_1, A_2, A_3, \ldots, A_n\}$, conociendo la probabilidad de cada una de las particiones A_i y la probabilidad de un suceso B condicionada a cada una de las particiones A_i, es decir $P(B|A_i)$, podemos calcular $P(A_i|B)$ de la siguiente manera.

$$P(A_i|B) = \frac{P(A_i \cap B)}{P(B)}$$

Que según lo visto hasta ahora podemos reescribir como:

$$P(A_i|B) = \frac{P(B|A_i) \times P(A_i)}{P(B|A_1) \times P(A_1) + P(B|A_2) \times P(A_2) + \cdots + P(B|A_n) \times P(A_n)}$$

En el denominador hemos aplicado el teorema de las probabilidades totales y en el numerador hemos usado la propiedad que vimos al introducir las probabilidades condicionadas.

Siguiendo con el ejemplo de los dados, ahora podríamos preguntarnos la probabilidad de sacar número par sabiendo que el valor obtenido es mayor o igual que 3. Es decir, si tenemos los sucesos

$A_1 = \{1,3,5\}$

$A_2 = \{2,4,6\}$

$B = \{3,4,5,6\}$

¿Cuál es la probabilidad A_1 sabiendo que se da B?

$$P(A_1|B) = \frac{P(A_1 \cap B)}{P(B)} = \frac{0.33}{0.66} = 0.5$$

También podemos calcularlo haciendo uso de la fórmula alternativa, con la que obtenemos el mismo resultado:

$$P(A_1|B) = \frac{P(B|A_1) \times P(A_1)}{P(B|A_1) \times P(A_1) + P(B|A_2) \times P(A_2)} = \frac{\frac{2}{3} \times \frac{3}{6}}{\left(\frac{2}{3} \times \frac{3}{6}\right) + \left(\frac{2}{3} \times \frac{3}{6}\right)} = \frac{0.33}{0.66} = 0.5$$

Al ser A_1 y A_2 una partición de todo el espacio muestral completo, es evidente que

$P(A_2|B) = 1 - P(A_1|B) = 0.5$

© RC Libros

Clasificación bayesiana ingenua

Como ya hemos apuntado, un clasificador bayesiano ingenuo es un clasificador probabilístico basado en la aplicación del teorema de Bayes. Es posible aplicar este tipo de clasificación a múltiples campos, por ejemplo, un clasificador de este tipo podría clasificar una serie de síntomas en una enfermedad concreta, basándose en la probabilidad de que dichos síntomas se den al tener la enfermedad.

El apellido de ingenuo es debido a que al crear el clasificador asumimos que las variables que vamos a usar para clasificar, por ejemplo, los síntomas, son totalmente independientes unos de otros. Es decir, que la probabilidad de que se dé un síntoma es independiente de la probabilidad de que se dé otro. Hacemos esta suposición aunque sepamos de hecho que no lo sean totalmente, y aun así este modelo probabilístico funciona sorprendentemente bien. En la sección anterior ya introdujimos el concepto de independencia en Probabilidad.

Podemos usar el teorema de Bayes para crear un modelo causa-efecto basado en probabilidad. Vamos a escribir el teorema de Bayes cambiando las nomenclaturas.

$$P(causa|efecto) = \frac{P(efecto|causa) \times P(causa)}{P(efecto)}$$

La probabilidad de que se dé una causa, condicionada a que se dé el efecto observado, puede calcularse mediante esta fórmula. Siguiendo con el símil del diagnóstico médico supongamos que un paciente tiene fiebre y queremos saber la probabilidad de que, al tener fiebre, el paciente tenga una gripe, es decir, queremos calcular P(gripe|fiebre), que es precisamente lo que hacen los médicos: a partir de unos síntomas dan un diagnóstico. Supongamos que el doctor sabe que la gripe causa fiebre en el paciente el 60% de las veces. Además, conocemos que el porcentaje de fiebre en la población general (independientemente de la causa) es del 15% y la de la gripe para esta época alcanza el 5%. Por lo tanto, tenemos los siguientes datos:

$P(fiebre|gripe) = 0.6$

$P(fiebre) = 0.15$

$P(gripe) = 0.05$

© RC Libros

A partir de estos datos podemos preguntarnos por la probabilidad de que la causa de la fiebre de un paciente sea la gripe.

$$P(gripe|fiebre) = \frac{P(fiebre|gripe) \times P(gripe)}{P(fiebre)} = \frac{0.7 \times 0.05}{0.15} = 0.233$$

Esto indica que, según los datos que manejamos, hay una probabilidad del 23,3% de que la fiebre haya sido provocada por una gripe.

En este ejemplo hemos trabajado con un solo síntoma o efecto. Para extender el cálculo a múltiples causas, usamos la siguiente expresión.

$$P(causa|efecto_1, efecto_2, \dots, efecto_n) = P(causa) \times \prod_i P(efecto_i|causa)$$

Con esto pretendemos que, dada una serie de efectos, seamos capaces de clasificar y, por lo tanto, encontrar la causa que los provoca. Vamos a desarrollar un ejemplo algo más complejo que el anterior y más enfocado al marketing. Supongamos que un concesionario tiene a la venta un coche familiar. Este coche no se vende mucho y quieren mejorar sus ventas. Son conscientes de que no es un coche dirigido al público general, así que quieren poder detectar, de entre los nuevos clientes, a quiénes de ellos ofrecérselo con ciertas garantías de que se interesarán por él. A partir de los datos de venta de los que disponen para ese modelo de coche, nos solicitan que hagamos una clasificación de sus nuevos clientes para ver a quiénes le ofrecen el coche y a quiénes no. Por lo general, una base de datos de clientes contendrá muchísima información. Aquí vamos a simplificar y supondremos que los datos que nos facilitan son los contenidos en la siguiente tabla en la que nos informan de las características del cliente y de si compró no ese modelo.

Sexo	Estado civil	Tiene hijos	¿Compró el coche?
Hombre	Casado	Sí	Sí
Hombre	Casado	No	Sí
Hombre	Soltero	No	No
Mujer	Casada	Sí	No
Mujer	Casada	No	Sí

Hombre	Casado	No	Sí
Hombre	Casado	No	No
Mujer	Soltera	No	No
Hombre	Soltero	No	No
Mujer	Casada	Sí	Sí

Partiendo de estos datos, queremos saber si una cliente que es mujer, soltera pero con hijos, podría estar interesada en comprar el coche. Fijémonos en que se trata de un caso que no se encuentra en la tabla, por lo que a priori, no tenemos una experiencia previa en usuarios con estas mismas características. Sin embargo, vamos a tratar de clasificar a la clienta como compradora o no compradora de este vehículo. Para ello, calcularemos las siguientes probabilidades:

$P(compra|mujer, soltera, con_hijos)$

$P(no_compra|mujer, soltera, con_hijos)$

Una vez calculadas ambas probabilidades y comparándolas podremos ver cuál de las dos opciones es más probable. Calcularemos primero las probabilidades para el caso en que la clienta compra. Empezaremos calculando $P(mujer|compra)$.

$$P(mujer|compra) = \frac{P(compra|mujer) \times P(mujer)}{P(compra)} = \frac{0.5 \times 0.4}{0.6} = 0.33$$

El resto de probabilidades se calcula de la misma manera aplicando la fórmula, por lo que sOlo dejamos escritos los resultados.

$P(soltera|compra) = 0.16$

$P(con_hijos|compra) = 0.33$

Calculamos ahora los valores para el caso en el que el cliente no compra el coche.

$P(mujer|no_compra) = 0.5$

© RC Libros

$P(soltera|no_compra) = 0.5$

$P(con_hijos|no_compra) = 0.25$

Ahora, usando la fórmula para el cálculo de la probabilidad cuando tenemos varios efectos, obtenemos ambos valores.

$P(compra|mujer, soltera, con_hijos) = 0.6 \times 0.33 \times 0.16 \times 0.33 = 0.01$

$P(no_compra|mujer, soltera, con_hijos) = 0.4 \times 0.5 \times 0.5 \times 0.25 = 0.025$

Quedándonos con el valor más alto (0.025), inferimos que la mujer no comprará el coche basándonos en los datos de los que disponemos.

Como hemos dicho al principio, este método de clasificación supone la independencia entre todas las variables aunque de hecho no sea así, por ejemplo, estar casado y tener hijos están fuertemente relacionados, ya que si se da la primera, la probabilidad de la segunda aumenta. Aun así, la clasificación bayesiana funciona bastante bien y es una herramienta muy poderosa para usarla en el campo de la **minería de datos**.

Una de las aplicaciones más útiles que tiene este método de clasificación en nuestra vida diaria es la clasificación de textos y más concretamente en el filtrado de spam o correo basura.

A modo de ejemplo vamos a implementar un programa de filtrado de spam usando el clasificador bayesiano que hemos presentado. Para ello, si en el ejemplo tomábamos como elementos caracterizadores el sexo, el estado civil y el haber sido padres, ahora necesitaremos otros que nos ayuden a clasificar los textos del correo electrónico. Hay multitud de características que nos pueden ayudar a clasificar el spam, como el uso de mayúsculas, el idioma del texto y la procedencia. Para nuestro ejemplo, y para mantener las cosas simples solo nos fijaremos en las palabras de más de dos letras que componen el correo electrónico.

Al igual que en el ejemplo de la compra del coche, vamos a usar la fórmula de Bayes pero ahora calculando la probabilidad condicionada de cada palabra.

$$P(palabra|categoria) = \frac{P(categoria|palabra) \times P(palabra)}{P(categoria)}$$

Donde la categoría en este caso son dos: SPAM o NOSPAM para indicar la probabilidad de la palabra sabiendo que el mensaje es o no spam.

Una vez calculada la probabilidad de cada palabra condicionada, nos resta multiplicar sus probabilidades para cada categoría tal y como hicimos en el ejemplo.

$$P(categoria|palabra_1, palabra_2, \dots, palabra_n) = P(categoria) \times \prod_i P(palabra_i|categoria)$$

Es decir, que multiplicamos cada probabilidad P(palabra$_i$|categoría) del texto que queremos clasificar y además también P(categoría).

Antes de poder clasificar un texto tendremos que poder analizar una serie de datos previos. Al igual que en el ejemplo de la venta de coches, teníamos una tabla con el comportamiento de anteriores clientes, el analizador de correo necesita ejemplos previos de correo que estén previamente clasificados como SPAM o NOSPAM. En general, cuantos más ejemplos proveamos, mejor comportamiento tendrá el clasificador. La fase en la que introducimos ejemplos en el sistema se llama entrenamiento, y permite almacenar características y estadísticas de palabras y categorías para luego usarlos en la clasificación de un nuevo texto.

En el ejemplo siguiente utilizaremos la función *entrenar()* para realizar el entrenamiento. La función espera una lista como la siguiente con el texto del mensaje y si es o no spam:

["Quedamos mañana lunes para ir al cine", "nospam"]

Se hace uso de la función auxiliar *lista_palabras()* que devuelve una lista con todas las palabras del texto mayores de dos caracteres, en minúsculas y sin repeticiones.

La función de entrenamiento genera cuatro variables que almacenan los parámetros necesarios para la posterior clasificación.

La variable *c_palabras* es un diccionario con las palabras del texto y el número de veces que aparece en cada categoría. Por ejemplo, si tenemos la palabra *interesante*, que aparece dos veces en correos que no son spam y una vez en un texto que es spam, se almacenaría de la siguiente manera:

{'interesante': {'nospam': 2, 'spam': 1}, 'relojes': {'nospam': 0, 'spam': 1}}

La variable *c_categorias* almacena en un diccionario cada categoría con el número de textos que pertenecen a ella.

{'nospam': 2, 'spam': 3}

© RC Libros

Finalmente, las variables *c_textos* y *c_tot_palabras* guardan el número de textos totales que hay de ejemplo y el número de palabras totales, respectivamente. Necesitaremos estos datos al hacer la clasificación.

La función *clasificar()* es la que hace los cálculos y determina a qué categoría pertenece el texto que le asamos como parámetro. También espera que pasemos las cuatro variables anteriores.

```
# clasificador spam con naive bayes
def lista_palabras(texto):
  palabras=[]
  palabras_tmp=texto.lower().split()
  for p in palabras_tmp:
    if p not in palabras and len(p)>2:
      palabras.append(p)

  return palabras



def entrenar(textos):
  c_palabras={}
  c_categorias={}
  c_textos=0
  c_tot_palabras=0
  # añadir al diccionario las categorías
  for t in textos:
    c_textos=c_textos+1
    if t[1] not in c_categorias:
      c_categorias[t[1]]=1
    else:
      c_categorias[t[1]]=c_categorias[t[1]]+1

  # añadir palabras al diccionario
  for t in textos:
    palabras=lista_palabras(t[0])
```

```
    for p in palabras:
      if p not in c_palabras:
        c_tot_palabras=c_tot_palabras+1
        c_palabras[p]={}
        for c in c_categorias:
          c_palabras[p][c]=0
      c_palabras[p][t[1]]=c_palabras[p][t[1]]+1

  return (c_palabras,c_categorias, c_textos, c_tot_palabras)



def clasificar(texto, c_palabras, c_categorias, c_textos,
c_tot_palabras):
  categoria=""
  prob_categoria=0
  for c in c_categorias:
    # probabilidad de la categoría
    prob_c=float(c_categorias[c])/float(c_textos)
    palabras=lista_palabras(texto)
    prob_total_c=prob_c
    for p in palabras:
      # probabilidad de la palabra
      if p in c_palabras:
        prob_p=float(c_palabras[p][c])/float(c_tot_palabras)
        # probabilidad P(categoria|palabra)
        prob_cond=prob_p/prob_c
        # probabilidad P(palabra|categoria)
        prob=(prob_cond*prob_p)/prob_c
        prob_total_c=prob_total_c*prob

    if prob_categoria<prob_total_c:
      categoria=c
      prob_categoria=prob_total_c

  return (categoria,prob_categoria)
```


© RC Libros

```
if __name__ == "__main__":
  textos=[
    ["Viagra a buen precio", "spam"],
    ["Quedamos mañana lunes para ir al cine", "nospam"],
    ["Replicas de relojes y viagra a precios de risa", "spam"],
    ["Disponga de sus productos farmaceuticos en 24 horas", "spam"],
    ["La Inteligencia Artificial es una disciplina muy interesante",
"nospam"]
  ]

  p,c,t,tp = entrenar(textos)
  clase = clasificar("Pedidos online de viagra servidos en 24 horas",
p, c, t, tp)
  print "Texto clasificado como: "+str(clase)
```

Listado. 7-1 antispam.py

Evidentemente, con tan pocos ejemplos no es posible hacer un buen entrenamiento del clasificador.

En este ejemplo hemos clasificado los textos en dos categorías: spam y no spam, pero nada impide hacer otras clasificaciones. Por ejemplo, si sustituimos los textos de entrenamiento por los siguientes, tendremos un clasificador capaz de distinguir entre los idiomas español, inglés y francés, o sea, tres categorías diferentes en vez de dos.

```
  textos=[
    ["Allez tout droit et prenez la prochaine rue", "frances"],
    ["Go straight and take the next street", "ingles"],
    ["Siga recto y gire en la siguiente calle", "espanol"],
    ["Intelligence Artificielle est une discipline \
    très intéressante","frances"],
    ["Artificial Intelligence is a very interesting \
    discipline", "ingles"],
    ["La Inteligencia Artificial es una disciplina \
    muy interesante", "espanol"]
  ]
```

© RC Libros

REDES NEURONALES ARTIFICIALES

La observación de la naturaleza y la de nosotros mismos son muy inspiradores a la hora de ayudarnos a crear técnicas y algoritmos aplicables en Inteligencia Artificial. En el capítulo 4 vimos cómo la aplicación del mecanismo biológico de la herencia y el cruce de genes dieron lugar a los algoritmos genéticos. De la misma manera, la comprensión del funcionamiento de nuestro propio cerebro puede ayudarnos en la creación de programas inteligentes.

Son muchos los autores que establecen un paralelismo entre los microprocesadores que hacen funcionar nuestros ordenadores y el cerebro. De hecho llegan a hacer comparaciones entre el número de transistores del microprocesador y del número de neuronas del cerebro. No hace muchos años, algunos aventuraban que cuando el número de transistores de los microprocesadores igualaran al número de neuronas del cerebro, podríamos crear ordenadores tan inteligentes como nosotros mismos.

Lo cierto es que no estamos tan lejos de construir procesadores con un número similar de transistores al número de neuronas del cerebro de algunos animales; sin embargo, estamos muy lejos de poder imitar su cerebro y la complejidad de sus capacidades. De hecho, la velocidad a la que se activan los transistores de los micros actuales es muy superior a la velocidad de activación de las neuronas humanas. ¿Por qué estamos tan lejos entonces de poder emular incluso las capacidades más simples del razonamiento humano?

La respuesta se halla en el diferente modelo de procesamiento en los que están basados. Mientras que un microprocesador procesa la información de forma secuencial, el cerebro es una estructura que procesa información de forma paralela y concurrente. No es trivial hacer que un microprocesador se comporte como una máquina capaz de procesar información en paralelo, de hecho es uno de los campos de la computación donde más esfuerzos se invierten actualmente.

Las **redes neuronales artificiales** o RNAs son un intento de emular la forma de trabajar del cerebro humano, y aunque estamos lejos de alcanzar su misma capacidad, son un instrumento de gran potencia para gran cantidad de aplicaciones y un campo de investigación muy prometedor.

Una neurona animal está compuesta por el núcleo, que no es más que una célula especializada, rodeada millones de conexiones que la unen a otras neuronas. Estas conexiones se denominan sinapsis. Cada neurona está conectada de media a otras mil, aunque pueden ser muchas más. Se estima que un niño tiene alrededor de 1.000

© RC Libros

billones de conexiones, que con el tiempo se van perdiendo hasta alcanzar alrededor de los 500 billones en un adulto.

Las conexiones se realizan mediante dos tipos de neurotransmisores: las dendritas y los axones. Según la neurociencia actual, parece que una neurona funciona de forma muy similar a un transistor, es decir, en un momento dado puede estar activa o no activa. En la realidad no es exactamente así, pero podemos decir que se acerca mucho a este modelo. Quizás, el decir que una neurona puede estar muy activa o poco activa describa mejor la realidad. En cualquier caso, las neuronas se disparan (se activan) en función de las dendritas. Las dendritas transportan señales eléctricas desde otras neuronas. Cuando la cantidad de dendritas que envían señales alcanzan un umbral determinado (umbral de disparo), la neurona se activa y envía señales eléctricas a otras neuronas a través de los axones. Es decir, funcionan como una auténtica red con billones de conexiones que trabajan de forma paralela.

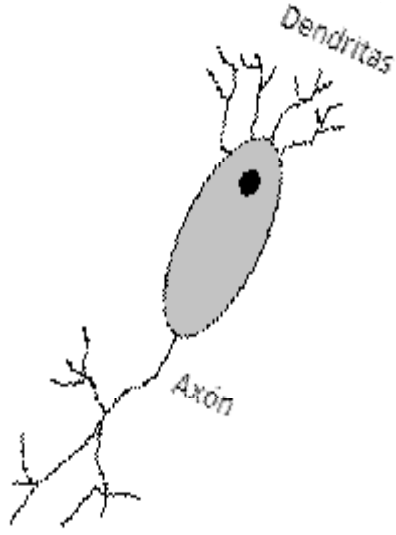


Fig. 7-1 Representación de una neurona

© RC Libros

El perceptrón simple

El **perceptrón** es un modelo simple de neurona que nos va a permitir presentar los conceptos básicos para luego ir profundizando en modelos más complejos. Al igual que una neurona real, a nuestro perceptrón llegarán señales de entrada (como si fueran dendritas de una neurona real) y saldrá una salida (simulando un axón). Las entradas de nuestro perceptrón podrán tomar los valores uno y cero. Además, a cada una de las entradas (x) se le asigna un valor al que llamaremos peso (w). La salida (z) del perceptrón es una función que depende del valor de las entradas, de sus pesos y del umbral de disparo de la neurona (θ) y su valor siempre está entre 0 y 1.

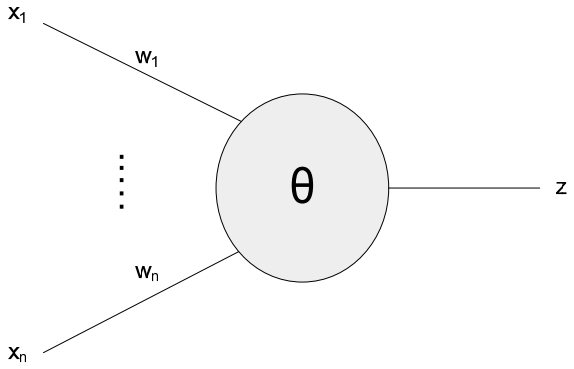


Fig. 7-2 Esquema de una neurona artificial

En la figura vemos el esquema de un perceptrón con sus entradas $x_1,\ldots,x_n$, los respectivos pesos de cada entrada $w_1,\ldots,w_n$, y el umbral de disparo a partir del cual la neurona se activa. La salida z se calcula según la siguiente función:

$$z = \sum_i x_i \times w_i$$

Es decir, la suma del producto de cada entrada por su correspondiente peso. Si el valor de z es mayor que θ, entonces el perceptrón se dispara. También es habitual hacer que se dispare con valor cero y añadir una nueva entrada con el valor –θ tal y como se ve en la siguiente figura. En cualquier caso, ambos perceptrones son equivalentes.

© RC Libros

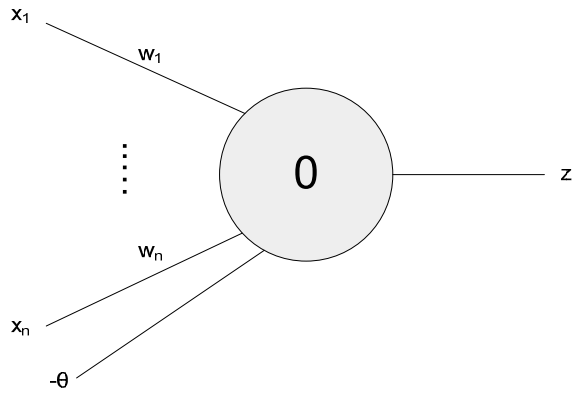


Fig. 7-3 Esquema alternativo de una neurona artificial

Nuestra tarea será ir ajustando los valores de los pesos $w_1,\ldots,w_n$ y de θ a través de un proceso llamado **entrenamiento**. En este tipo de redes neuronales las llamamos de **aprendizaje supervisado**, ya que durante el entrenamiento vamos a ir proveyendo de ejemplos a la red y según la respuesta de la red comparada con la respuesta esperada, vamos a ir ajustando los valores correspondientes. Dicho de otra forma, para cada ejemplo habrá que indicar a la red neuronal cuál es el resultado que debería darse en la salida. El clasificador bayesiano que hemos visto en la primera parte del capítulo también es un ejemplo de aprendizaje supervisado.

A modo de ejemplo, supondremos que queremos crear una red neuronal que clasifique clientes de un banco para ver si se les concede o no una hipoteca dependiendo de dos factores: x_1 = Sueldo bruto y x_2 = Nivel de deuda. Tras entrenar a un perceptrón con muchos casos previos hemos obtenido los siguientes parámetros de configuración de la neurona:

w_1 = -0.2

w_2 = 0.5

θ = 0.6

Sustituyendo en la fórmula tenemos $z = -0.2x_1 + 0.5x_2 - 0.6$. Esta ecuación nos recuerda bastante a la de una recta. En concreto a la ecuación $-0.2x_1 + 0.5x_2 - 0.6 = 0$ cuya representación es la siguiente.

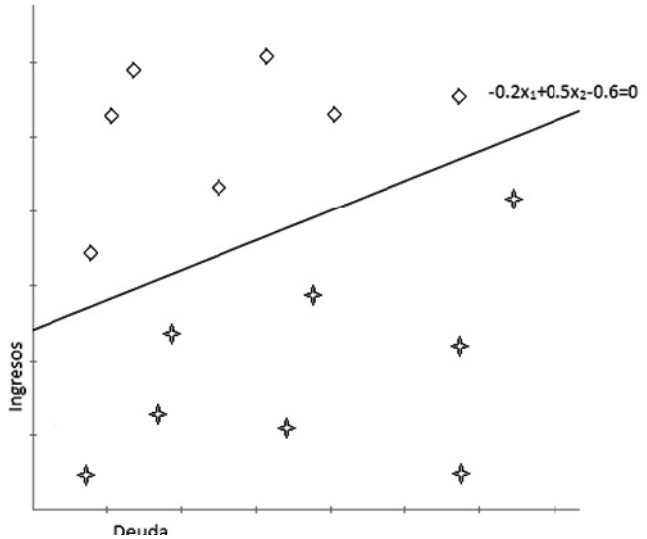


Fig. 7-4 Separación de clientes en dos clases diferentes

Como vemos, el resultado es una recta que parte en dos el espacio de un plano. Idealmente, tras el entrenamiento, habremos conseguido separar a los clientes con más riesgo, que son aquellos que quedan por debajo de la recta, representados por una estrella, de los clientes con menos riesgo que son los que han quedado por encima de la recta, representados por un rombo. En resumen, la neurona se dispara para valores que están por encima de la recta.

El proceso de entrenamiento general que nos va a permitir ajustar los pesos $w_1,\ldots,w_n$ y θ es el siguiente:

- Ajustar los pesos $w_1,\ldots,w_n$ y θ de forma aleatoria.
- Aplicar las entradas $x_1,\ldots,x_n$ a la red.
- Calcular la salida z.
- Comparar el resultado obtenido con el deseado. A la diferencia entre ambos resultados la llamamos error.
- Modificar los pesos $w_1,\ldots,w_n$ y el valor umbral θ según el error obtenido.
- Repetir el proceso hasta que el error esté dentro de un rango aceptable (habitualmente <1%).

© RC Libros

Vamos a añadir un factor más llamado factor de aprendizaje que representaremos con λ y que indica con qué velocidad va a aprender la red. Un valor muy alto puede hacer que no termine de ajustar correctamente los pesos, mientras que un valor muy bajo hará que se converja muy lentamente hasta los valores correctos. Un valor de 0.2 suele ser habitual.

Vemos en detalle cómo calcular el error y hacer los reajustes de los pesos y del umbral de la neurona. Al entrenar la neurona con una entrada $x_1,\ldots,x_n$, obtendremos una entrada z. Suponiendo que la salida deseada sea y, el error es:

$$e = (y - z)$$

Este error nos permite ajustar θ sumándole el siguiente resultado:

$$\Delta\theta = -(\lambda \times \mathrm{e})$$

Y los pesos se ajustan sumándoles a cada uno el resultado de:

$$\Delta w_i = \lambda \times \mathrm{e} \times x_i$$

A modo de resumen, vamos a desarrollar un ejemplo completo de todo el proceso de entrenamiento de un perceptrón simple. Queremos hacer que nuestra neurona sea capaz de comportarse como una puerta lógica OR, es decir, que dadas dos entradas X_1 y X_2 el resultado z sea el valor de la función lógica OR cuya tabla de verdad es:

X_1	X_2	X_1 OR X_2
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

Si representamos las entradas x_1, x_2 y la salida z en una gráfica, observamos que en realidad lo que buscamos es clasificar las entradas de forma que la recta separe aquellas combinaciones de las entradas que dan lugar al resultado 1 de aquellas que dan lugar al resultado 0.

© RC Libros

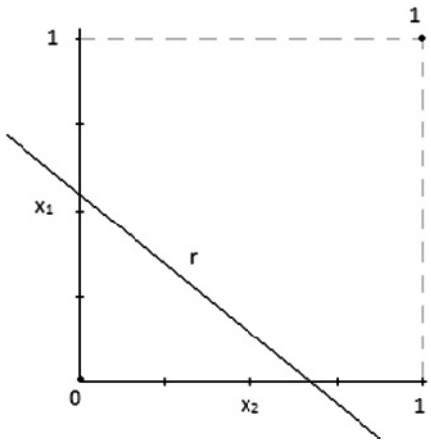


Fig. 7-5 Recta que separa valores de la función OR

Comenzaremos dando valores iniciales a los pesos, a θ y a λ.

$\Lambda = 0.2$

$w_1 = 0.2$

$w_2 = 0.6$

$\theta = 0.4$

Según estos valores, la recta que divide el plano se muestra en la figura siguiente en la que podemos observar cómo la división no es correcta.

© RC Libros

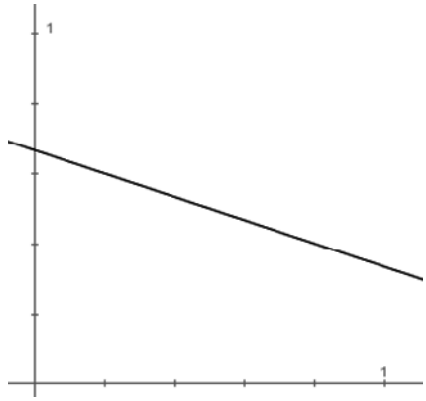


Fig. 7-6 Primer intento de buscar la función OR

Vamos a realizar el entrenamiento de las cuatro entradas posibles de forma iterativa hasta obtener el resultado deseado.

Iteración 1.

Entrenamos la entrada $x_1 = 0$ y $x_2 = 1$:

$$z = 0 \times 0.2 + 1 \times 0.6 - 0.4 = 0.2$$

Como z ≥ 0 se dispara y por lo tanto z = 1

Valor esperado y = 1 por lo tanto, como z=y, no hacemos nada.

Entrenamos la entrada $x_1 = 1$ y $x_2 = 0$:

$$z = 1 \times 0.2 + 0 \times 0.6 - 0.4 = -0.2$$

Como z < 0 no se dispara y por lo tanto z = 0

Valor esperado y = 1 por lo tanto, como z ≠ y, ajustamos pesos.

$$e = 1 - 0 = 1$$

© RC Libros

Ajustamos parámetros:

$\Delta\theta = -(0.2 \times 1) = -0.2$

$\Delta w_1 = 0.2 \times 1 \times 1 = 0.2$

$\Delta w_2 = 0.2 \times 1 \times 0 = 0$

Aplicando los incrementos obtenemos:

$\theta = 0.4 + (-0.2) = 0.2$

$w_1 = 0.2 + 0.2 = 0.4$

$w_2 = 0.6 + 0 = 0.6$

Entrenamos la entrada $x_1 = 0$ y $x_2 = 0$:

$z = 0 \times 0.2 + 0 \times 0.6 - 0.2 = -0.2$

Como z < 0 no se dispara y por lo tanto z = 0

Valor esperado y = 0 por lo tanto, como z = y, no hacemos nada.

Entrenamos la entrada $x_1 = 1$ y $x_2 = 1$:

$z = 1 \times 0.4 + 1 \times 0.6 - 0.2 = 0.8$

Como z ≥ 0 se dispara y por lo tanto z = 1

Valor esperado y = 1 por lo tanto, como z = y, no hacemos nada.

Iteración 2.

Entrenamos la entrada $x_1=0$ y $x_2=1$:

$z = 0 \times 0.4 + 1 \times 0.6 - 0.2 = 0.4$

© RC Libros

Como z ≥ 0 se dispara y por lo tanto z = 1

Valor esperado y = 1 por lo tanto, como z = y, no hacemos nada.

Entrenamos la entrada $x_1 = 1$ y $x_2 = 0$:

$$z = 1 \times 0.4 + 0 \times 0.6 - 0.2 = 0.2$$

Como z ≥ 0 se dispara y por lo tanto z = 1

Valor esperado y = 1 por lo tanto, como z = y, no hacemos nada.

Entrenamos la entrada $x_1 = 0$ y $x_2 = 0$:

$$z = 0 \times 0.4 + 0 \times 0.6 - 0.2 = -0.2$$

Como z < 0 no se dispara y por lo tanto z = 0

Valor esperado y = 0 por lo tanto, como z = y, no hacemos nada.

Entrenamos la entrada $x_1 = 1$ y $x_2 = 1$:

$$z = 1 \times 0.4 + 1 \times 0.6 - 0.2 = 0.8$$

Como z ≥ 0 se dispara y por lo tanto z = 1

Valor esperado y = 1 por lo tanto, como z = y, no hacemos nada.

En este punto, todos los ejemplos usados para entrenar al perceptrón han coincidido con el valor esperado, por lo que ya hemos terminado el entrenamiento. La recta que definen los pesos y el valor umbral es la representada en la siguiente figura, que esta vez sí que divide el plano en dos: los valores para los que la salida vale 0 y los valores para los que la salida vale 1.

© RC Libros

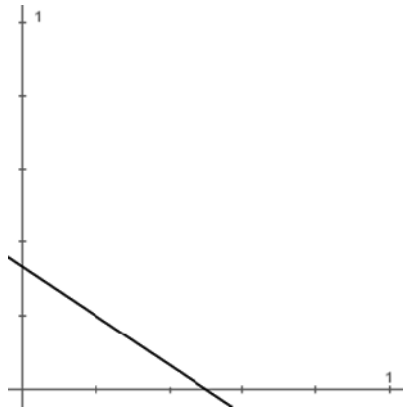


Fig. 7-7 Encontrada la función OR

El siguiente listado es una implementación del perceptrón simple para el cálculo de función lógica OR, aunque cambiando la variable datos_ent podremos entrenar al perceptrón para otras funciones lógicas.

```
# perceptrón simple para función OR
def salida(k, pesos, t):
  z=-t
  for i in range(len(k)):
    z=z+(k[i]*pesos[i])
  if z>=0:
    return 1
  else:
    return 0



def entrenar_perceptron(datos_ent, pesos, t, l):
  errores=True
  while errores:
    errores=False
    # entrenar perceptrón
```


© RC Libros

```
    for k,y in datos_ent.iteritems():
      z=salida(k, pesos, t)
      if z!=y:
        errores=True
        # error
        e=(y-z)
        # calcular ajustes
        delta_t=-(l*e)
        t=t+delta_t
        for i in range(len(k)):
          delta_w=l*e*k[i]
          pesos[i]=pesos[i]+delta_w

  return pesos, t


def clasificar(entrada, pesos, t):
  return salida(entrada, pesos, t)


if __name__ == "__main__":
  datos_ent={(0,0):0, (0,1):1, (1,0):1,(1,1):1}
  pesos=[0.2,-0.5]
  t=0.4
  l=0.2
  pesos, t = entrenar_perceptron(datos_ent, pesos, t, l)
  print clasificar((0,1), pesos, t)
```

Listado. 7-2 perceptron.py

© RC Libros

Redes neuronales multicapa

Para tareas simples, el perceptrón que acabamos de presentar puede tener cierta utilidad, aunque si necesitamos enfrentarnos a tareas más complejas no será lo suficientemente efectivo. Hemos visto cómo la neurona puede entrenarse para que sea capaz de calcular la función OR. Si probamos con la función AND o NOT, veremos que también funciona, pero probaremos con la función XOR. Para ello, sustituimos la línea siguiente en el listado anterior:

```
datos_ent={(0,0):0, (0,1):1, (1,0):1,(1,1):0}
```

Recordemos que la tabla de verdad para la función XOR es la siguiente:

X_1	X_2	X_1 XOR X_2
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Si ejecutamos el programa anterior con la intención de que la neurona sea capaz de resolver la función XOR, veremos cómo entra en un bucle infinito y nunca es capaz de terminar el entrenamiento, es decir, que no consigue encontrar una recta que divida en el planos los ceros y los unos. En la siguiente figura puede observarse mejor el problema, que se reduce a la siguiente cuestión. ¿Se puede encontrar una recta que parta el plano en dos separando los puntos con valor 0 y los puntos con valor 1?

© RC Libros

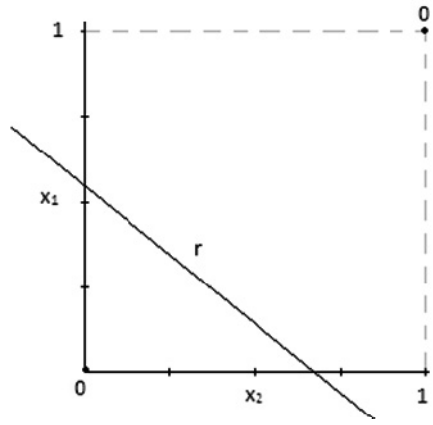


Fig. 7-8 ¿Hay alguna recta que resuelva XOR?

La realidad es que no es posible encontrar esa recta, por lo tanto, el perceptrón simple no es capaz de resolver este problema. Podemos suponer –y de hecho así es– que habrá otros problemas que el perceptrón simple no pueda tratar. Si podemos dividir plano en dos usando una recta (o el espacio en dos usando un plano, etc.) decimos que se trata de un problema **linealmente separable**. En otro caso, como ocurre con XOR, hablamos de problemas que **no linealmente separables**.

Una posible solución es usar el resultado de una neurona como entrada de otra. Por ejemplo, en la siguiente figura se observa cómo una neurona puede dividir el espacio usando la recta r1 de forma que lo que hay bajo esta línea sabemos que tiene el valor cero. A su vez, lo que queda por encima puede volver a dividirse con otra neurona tal y como lo hace la recta r2. Sabemos que lo que hay debajo tiene el valor 1 (r1 ya había resuelto en cero que también queda por debajo de r2, por lo que ya no lo tenemos en cuenta). Finalmente, r3 particiona el espacio restante que queda sobre r2 para separar el cero y el uno que faltan por clasificar.

© RC Libros

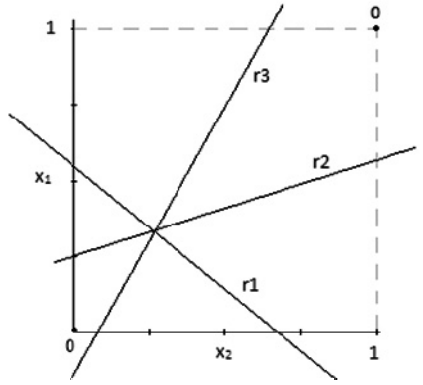


Fig. 7-9 Separando el plano con multiples neuronas

El proceso es plausible, pero bastante engorroso. ¿Y si fuera posible separar el plano usando una curva en vez de una recta como en la siguiente figura?

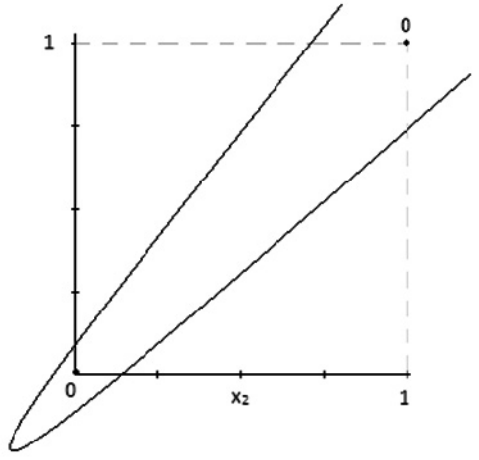


Fig. 7-10 Separando el plano con una curva

© RC Libros

Sin entrar en complejos detalles ni en demostraciones matemáticas, en general podemos decir que si organizamos un conjunto de neuronas formando una red conseguiremos diferentes niveles de complejidad en las particiones del plano (y particiones del espacio o cualquier otra dimensión como veremos seguidamente). Es lo que se llama una red neuronal. Hay diferentes formas de organizar las redes, aunque aquí nos centraremos en dos muy concretas que son representativas de dos tipos de redes neuronales: la **red de retropropagación** o ***back propagation network*** y la **red de Hopfield**. La primera pertenece al grupo de redes de aprendizaje supervisado, que entrenamos, como ya hemos visto, dando entradas y la respuesta que debemos esperar. La segunda es una red de **aprendizaje no supervisado** en la que, como veremos un poco más adelante, no es necesario decirle cuál es la respuesta correcta a una entrada concreta.

La red de de retropropagación se organiza en capas que pueden tener un número arbitrario de neuronas, con la particularidad de que cada neurona está conectada a todas las de la capa anterior (excepto las de la primera capa). A las neuronas de la primera capa se las conoce como neuronas de entrada, y a las de la última capa las llamamos neuronas de salida. Evidentemente, el número de entradas y de salidas dependerá del tipo de problema a resolver. Por ejemplo, para crear una red que calcula la función XOR necesitaremos dos entradas y una salida.

En la figura siguiente está representada una red de dos capas con dos neuronas en cada una. Observamos cómo cada una de las neuronas en la capa de salida se conecta a todas las neuronas de la capa anterior a través de un valor $W_{i,j}$ que llamamos peso y que ya utilizábamos en el perceptrón simple.

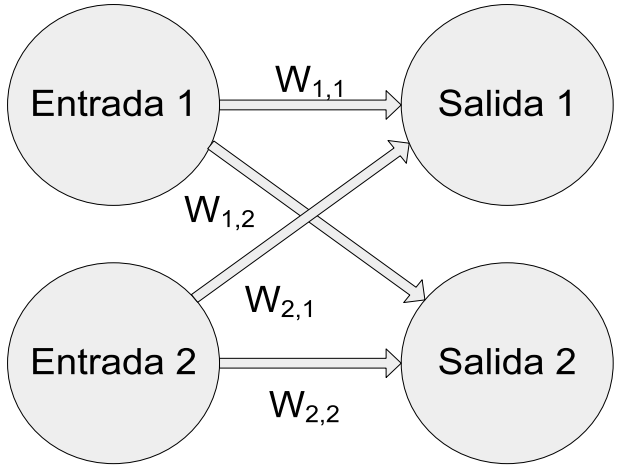


Fig. 7-11 Ejemplo de red con dos capas

Como hemos dicho, la red puede organizarse en varias capas. Las que quedan entre las neuronas de entrada y las de salida las llamamos capas ocultas, siendo posible tener más de una capa de neuronas ocultas.

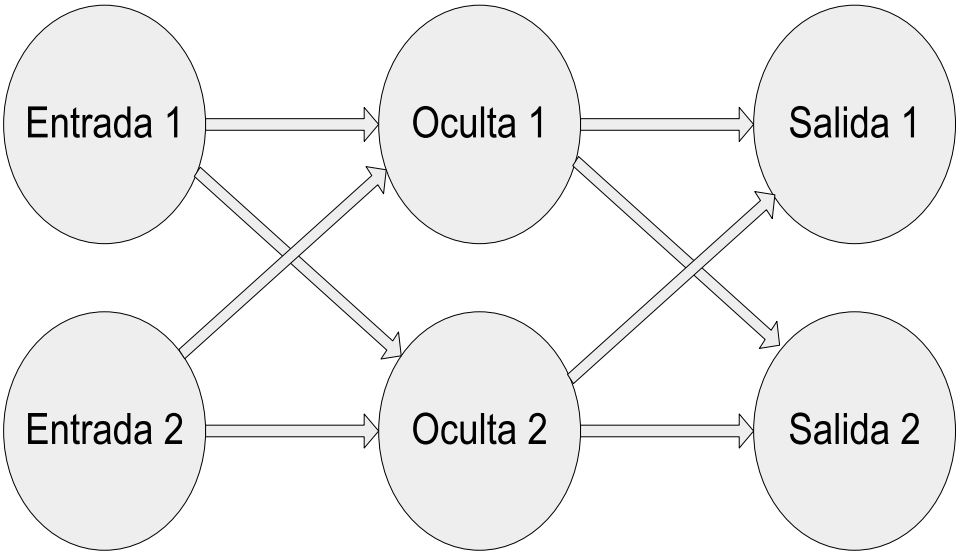


Fig. 7-12 Ejemplo de red con tres capas (una oculta)

Del número de capas ocultas que tenga la red va a depender el tipo de problemas que podemos resolver. Sin capas ocultas, seremos capaces de resolver problemas linealmente separables. Con una capa oculta podremos resolver problemas que son separables mediante una curva, y con dos capas ocultas podemos resolver problemas en los que se den separaciones arbitrarias, por lo que en principio, no tiene mucho sentido usar más de dos capas ocultas en una red. Por simplificar representamos las separaciones en un plano de dos dimensiones, pero se puede generalizar a cualquier número de dimensiones (que depende del número de neuronas en la capa de entrada).

© RC Libros

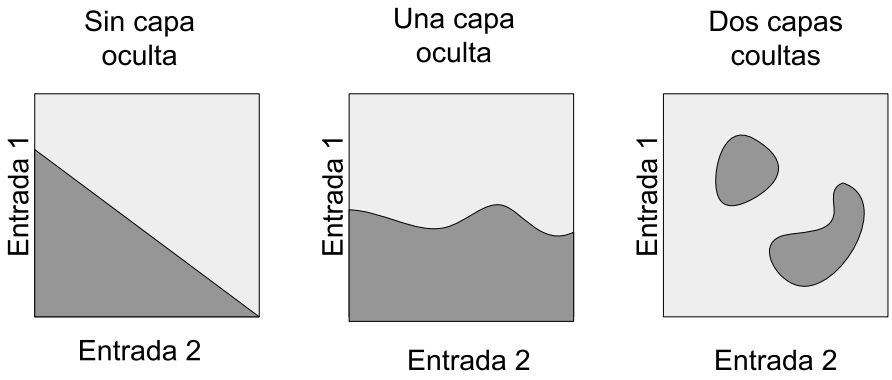


Fig. 7-13 Capacidad de separación del espacio según las capas ocultas

Con el perceptrón simple, considerábamos que la neurona se activaba si la suma de los pesos multiplicados por las entradas superaban un umbral determinado que llamábamos θ. Para las redes de capa oculta, necesitamos una función de activación diferente, ya que el algoritmo que vamos a usar para transmitir los errores de una capa a otra necesitan que dicha función sea derivable, y la función umbral, que tiene forma de escalón no lo es. En su lugar usaremos una función de activación para las neuronas de tipo sigmoide como la siguiente.

$$sigmoide(z) = \frac{1}{1 + e^{-z}}$$

Aunque podemos usar otras como la tangente hiperbólica que es muy similar y que es quizás más cómoda de usar en un programa (usaremos esta última en el programa de ejemplo). Lo interesante de esta función es que está acotada entre -1 y 1 y que crece muy rápidamente hasta 1 para valores positivos y decrece igual hasta el valor -1 para valores negativos.

© RC Libros

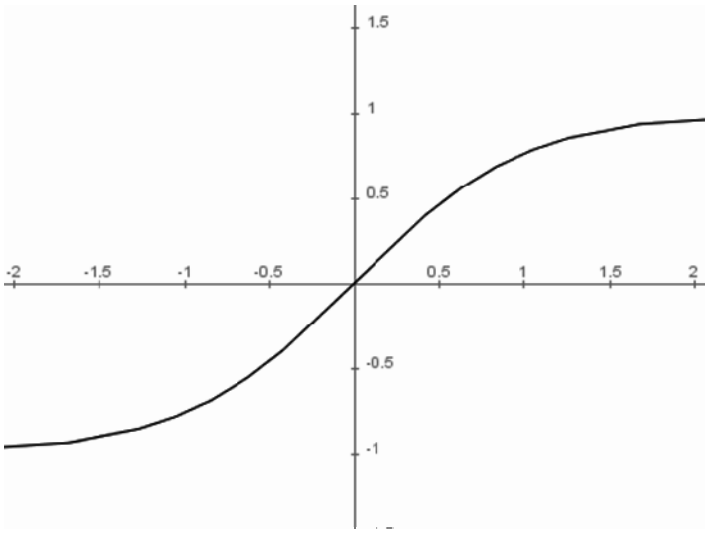


Fig. 7-14 Ejemplo de función de tipo sigmoide (tangente hiperbólica)

Así pues, las neuronas de la red se activarán ahora según la siguiente función:

$$z = Sigmoide(\sum_i x_i \times w_i)$$

Es decir, para la red de dos capas de la figura 5-11 tendríamos que:

$$salida\ 1 = Sigmoide(Entrada\ 1 \times P_{1,1} + Entrada\ 2 \times P_{2,1})$$

$$salida\ 2 = Sigmoide(Entrada\ 1 \times P_{1,2} + Entrada\ 2 \times P_{2,2})$$

Ahora las salidas no serán valores discretos 0 y 1 como en el caso del perceptrón simple, sino que serán valores reales. Consideraremos que una neurona se activa si su valor de salida es cercano a 1, por ejemplo, un valor de 0.963 puede ser considerado como un valor que activa la neurona.

El algoritmo de retropropagación tiene dos fases. En la primera se actualizan todos los nodos de la red, empezando por los nodos de entrada, según acabamos de ver. En una segunda fase, calculamos el error obtenido en la salida y lo propagamos hacia atrás para actualizar los pesos de las conexiones de acuerdo al error (de ahí el nombre del algoritmo).

© RC Libros

En la capa de salida, el error se calcula igual que en el perceptrón simple: restando el valor obtenido del valor esperado.

$$e = (y - z)$$

A partir del error se calcula el incremento corrector necesario a aplicar en los pesos de las conexiones de esta capa. Este valor se calcula con:

$$\Delta_{salida\ i} = \text{dsigmoide}(x_{salida\ i}) \times e$$

Donde dsigmoide se refiere a la derivada de la función sigmoide que estamos usando. En nuestro caso la tangente hiperbólica cuya derivada es $1\text{-}x^2$. Necesitamos la derivada ya que nos indica el ritmo de crecimiento de la función sigmoide para el valor dado, y por lo tanto la variación que hay que hacer en ese punto para realizar el ajuste de pesos.

Ahora necesitamos propagar el error hacia las capas ocultas, donde el error se calcula con:

$$e = \sum_{i,j} \Delta_{salida\ j} \times W_{i,j}$$

Es decir, sumamos los incrementos calculados desde los nodos de salida y los multiplicamos por los pesos de las conexiones. Recordemos que cada nodo de salida tiene una conexión con cada uno de los nodos de la capa anterior. Finalmente, con el error calculado obtenemos el incremento de ajuste para la capa oculta.

$$\Delta_{oculta\ i} = \text{dsigmoide}(x_{oculta\ i}) \times e$$

Para ajustar los pesos calculamos el cambio que hay que hacer en ellos y ajustamos los pesos. Para cada par i,j de neuronas calculamos el peso de su conexión como:

$$cambio = \Delta_{ni} \times \mathrm{x_{nj}}$$

$$W_{i,j} = W_{i,j} + (\lambda \times \text{cambio})$$

Siendo n el número de la capa (que puede ser de salida u oculta) y λ el factor de aprendizaje.

El siguiente código de ejemplo implementa una red neuronal con una capa oculta para resolver la función XOR. La función *actualiza_nodos()* es la encargada de realizar

la primera fase del algoritmo, que es la de actualizar los nodos. La función *retropropagacion()* es la que calcula los errores, los propaga hacia atrás en la red y finalmente ajusta los pesos.

```
# Perceptrón multicapa
# para cálculo de función XOR
import math
import random
import string

# crea una matriz para almacenar los pesos
def matriz(x, y):
  m = []
  for i in range(x):
      m.append([0.0]*y)
  return m

# función de tipo sigmoide
def sigmoide(x):
  return math.tanh(x)

# derivada de función de tipo sigmoide
def dsigmoide(x):
  return 1.0 - x**2

# inicialización
def iniciar_perceptron():
  global nodos_ent, nodos_ocu, nodos_sal, pesos_ent, pesos_sal
  global act_ent, act_ocu, act_sal
  random.seed(0)

  # sumamos uno para umbral en nodos entrada
  nodos_ent = nodos_ent + 1
```


© RC Libros

```
  # activación de los nodos
  act_ent = [1.0]*nodos_ent
  act_ocu = [1.0]*nodos_ocu
  act_sal = [1.0]*nodos_sal

  # crear matrices de pesos
  pesos_ent = matriz(nodos_ent, nodos_ocu)
  pesos_sal = matriz(nodos_ocu, nodos_sal)
  # inicializar pesos a valores aleatorios
  for i in range(nodos_ent):
      for j in range(nodos_ocu):
          pesos_ent[i][j] = random.uniform(-0.5, 0.5)
  for j in range(nodos_ocu):
      for k in range(nodos_sal):
          pesos_sal[j][k] = random.uniform(-0.5, 0.5)


# actualizar valor de los nodos
def actualiza_nodos(entradas):
  global nodos_ent, nodos_ocu, nodos_sal, pesos_ent, pesos_sal
  global act_ent, act_ocu, act_sal
  if len(entradas) != nodos_ent-1:
      raise ValueError('Numero de nodos de entrada incorrectos')

  # activación en nodos de entrada
  for i in range(nodos_ent-1):
      act_ent[i] = entradas[i]

  # activación en nodos ocultos
  for j in range(nodos_ocu):
      sum = 0.0
      for i in range(nodos_ent):
          sum = sum + act_ent[i] * pesos_ent[i][j]
      act_ocu[j] = sigmoide(sum)
```

© RC Libros

```
  # activación en nodos de salida
  for k in range(nodos_sal):
      sum = 0.0
      for j in range(nodos_ocu):
          sum = sum + act_ocu[j] * pesos_sal[j][k]
      act_sal[k] = sigmoide(sum)

  return act_sal[:]


# retropropagación de errores
def retropropagacion(objetivo, l):
  global nodos_ent, nodos_ocu, nodos_sal, pesos_ent, pesos_sal
  global act_ent, act_ocu, act_sal
  if len(objetivo) != nodos_sal:
      raise ValueError('numero de objetivos incorrectos')

  # error en nodos de salida
  delta_salida = [0.0] * nodos_sal
  for k in range(nodos_sal):
      error = objetivo[k]-act_sal[k]
      delta_salida[k] = dsigmoide(act_sal[k]) * error

  # error en nodos ocultos
  delta_oculto = [0.0] * nodos_ocu
  for j in range(nodos_ocu):
      error = 0.0
      for k in range(nodos_sal):
          error = error + delta_salida[k]*pesos_sal[j][k]
      delta_oculto[j] = dsigmoide(act_ocu[j]) * error

  # actualizar pesos de nodos de salida
  for j in range(nodos_ocu):
```


© RC Libros

```
      for k in range(nodos_sal):
          cambio = delta_salida[k]*act_ocu[j]
          pesos_sal[j][k] = pesos_sal[j][k] + l*cambio

  # actualizar pesos de nodos de entrada
  for i in range(nodos_ent):
      for j in range(nodos_ocu):
          cambio = delta_oculto[j]*act_ent[i]
          pesos_ent[i][j] = pesos_ent[i][j] + l*cambio

  # calcular error
  error = 0.0
  for k in range(len(objetivo)):
      error = error + 0.5*(objetivo[k]-act_sal[k])**2
  return error


# clasificar patrón de entrada
def clasificar(patron):
  for p in patron:
      print p[0], '->', actualiza_nodos(p[0])


# entrenamiento del perceptrón
def entrenar_perceptron(patron, l, max_iter=1000):
  for i in range(max_iter):
    error = 0.0
    for p in patron:
      entradas = p[0]
      objetivo = p[1]
      actualiza_nodos(entradas)
      error = error + retropropagacion(objetivo, l)
    # salir si alcanzamos el límite inferior de error deseado
    if error < 0.001:
```

```
      break


if __name__ == '__main__':
  datos_ent = [[[0,0], [0]],
    [[0,1], [1]],
    [[1,0], [1]],
    [[1,1], [0]]]

  nodos_ent=2  # dos neuronas de entrada
  nodos_ocu=2  # dos neuronas ocultas
  nodos_sal=1  # una neurona de salida
  l=0.5
  iniciar_perceptron()
  entrenar_perceptron(datos_ent, l)
  clasificar(datos_ent)
```

Listado. 7-3 perceptron_multicapa.py

La ejecución del programa da como resultado:

```
[0, 0] -> [0.004343095187125427]

[0, 1] -> [0.9693877479773788]

[1, 0] -> [0.9687104891927546]

[1, 1] -> [-0.00743348049585324]
```

Donde podemos considerar que los valores que están en el entorno de 1 implican que la neurona se activa (valor 1) y los valores cercanos a 0 hacen que la neurona no se dispare (valor 0).

© RC Libros

Red de Hopfield

La red neuronal que acabamos de ver tiene mucho potencial de aplicación, pero tiene un pequeño problema, y es que hay que entrenarla dándole de antemano las salidas correctas, lo que para algunas aplicaciones puede ser engorroso, o simplemente puede ocurrir que no las conocemos. Si estuviéramos interesados en crear una red neuronal capaz de predecir las subidas y bajadas de la bolsa o el clima que va a hacer mañana, el modelo de red multicapa que hemos visto no sería demasiado cómodo de usar (lo que no quiere decir que no se pueda).

La red que vamos a presentar a continuación se llama **red de Hopfield** –en honor a su inventor John Hopfield– y pertenece a la clase de redes de aprendizaje no supervisado, ya que durante el entrenamiento solo introduciremos entradas en el sistema, pero no las salidas que deben producirse en respuesta a estos.

Este tipo de red parece recrear mejor el comportamiento del cerebro humano e imitar en cierto grado la forma de trabajar del neocórtex, que parece almacenar los recuerdos usando una **memoria asociativa** (las redes de Hopfield son denominadas memorias asociativas por algunos autores).

En efecto, el escuchar una canción de nuestra infancia, percibir algún olor o ver cierto paisaje nos evoca recuerdos de los que no éramos conscientes. Es como si un determinado suceso pudiera desencadenar la recuperación de recuerdos que teníamos latentes. Los recuerdos parecen almacenarse asociados unos a otros, de forma que al evocar uno, los demás van surgiendo incluso si tienen bastante tiempo.

Aunque a veces los recuerdos nos parezcan muy claros, ocurre que pueden no ser exactos o estar distorsionados aunque no seamos conscientes de ello. El cerebro tiende a hacer generalizaciones de las experiencias que al ser rescatados sufren cierta interpretación. Esto, aunque pueda parecer un inconveniente, es una ventaja que nos permite reconocer caras, personas, objetos, etc. Por ejemplo, aunque veamos un modelo de coche por primera vez, somos capaces de saber que es un choche porque el cerebro es capaz de generalizar dicho concepto. Lo mismo ocurre si vemos a un conocido. Aunque lleve gafas de sol o media cara tapada con una bufanda podemos reconocerlo. A partir de unos datos sesgados –como la imagen de media cara– el cerebro evoca el rostro completo.

De este fenómeno se aprovechan sistemas de seguridad como los ***capcha*** (*Completely automated public Turing test to tell computers and humans apart*) o algo así como: prueba pública de Turing para diferenciar ordenadores de humanos. Los

capchas son muy usados en internet para evitar que robots de software suplanten a humanos a la hora de rellenar formularios o hacer peticiones.

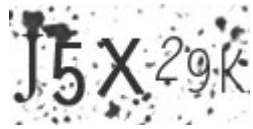


Fig. 7-15 Ejemplo de capcha

La fortaleza de los *capchas* se basa en que el cerebro humano es capaz de reconocer las letras como patrones y diferenciarlos del resto de ruido de la imagen. Un ordenador, a priori, tiene mucha dificultad para realizar este tipo de trabajos.

En cualquier caso hay técnicas, como las redes de Hopfield entre otras, que son capaces de reconocer diferentes tipos de *capchas* si no son demasiado complejos (para los más complejos es necesario hacer ciertos tratamientos a la imagen, como eliminación de ruidos, segmentación, etc.). Desde hace años, se utilizan con éxito estas redes para el reconocimiento óptico de caracteres (OCR) que combinado a veces con otras técnicas, como la clasificación bayesiana, permiten transformar imágenes escaneadas de libros a texto editable por el ordenador. Si tenemos en cuenta que existen diferentes tipos de tipografía, que la calidad de las impresiones no son siempre buenas y que los caracteres pueden estar distorsionados o medio borrados, entenderemos que no es una tarea sencilla.

Para entender mejor cómo funciona una red de Hopfield, vamos a hacer una analogía con un sistema físico bien conocido. Si tenemos un bol de cocina y dejamos deslizar una canica sobre su superficie, esta se moverá de un lado a otro hasta que finalmente se pare en el fondo. Esta se detiene cuando su energía cinética y potencia es cero (la energía se ha invertido en generar calor al rozar con la superficie del bol). Podemos describir el sistema en términos de energía.

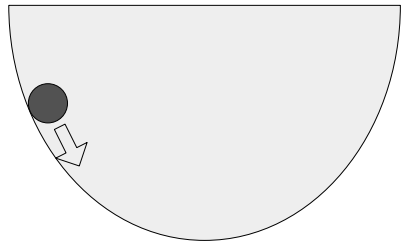


Fig. 7-16 Analogía con un bol

© RC Libros

La bola siempre se parará en el mismo lugar al que llamamos estado estable. De alguna manera podemos decir que el sistema "recuerda" cuál es el estado estable. Da igual donde dejemos la bola, que siempre caerá en este estado estable.

Si en lugar de un bol tuviéramos una superficie ondulada como la de la figura 7-17 y dejáramos caer una bola en cualquier sitio, siempre acabaría en el fondo de la ondulación más cercana. Es decir, en el mínimo local más próximo. Podríamos decir que este sistema tiene cuatro estados estables y por lo tanto es capar de "recordar" cuatro estados.

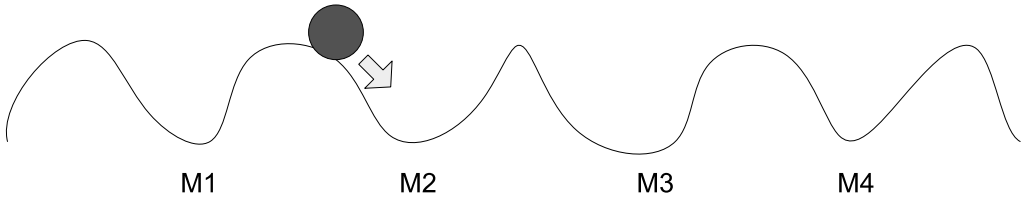


Fig. 7-17 Analogía con una superficie ondulada

Esta analogía nos permite entender cómo la red de Hopfield es capaz de reconocer un estado (por ejemplo, un carácter) a partir de una aproximación (por ejemplo, un carácter distorsionado). Si introducimos en la red de Hopfield el siguiente símbolo:



Será como si pusiéramos la bola sobre la superficie del bol (o la ondulación) correspondiente el número 5, y esta buscará su estado estable o de mínima energía, que será la representación canónica y "limpia" del número 5 con el que entrenamos inicialmente a la red.

Estructuralmente, una red de Hopfield está compuesta por una serie de neuronas interconectadas todas entre sí y relacionadas a través de un peso w, de la misma forma que ocurre en las redes multicapa, pero ahora la información no se va propagando por las diferentes capas, sino que la activación de una neurona influye

© RC Libros

en todas las demás y, por lo tanto, sobre sí misma. Es por ello que este tipo de redes se denominan **recurrentes**.

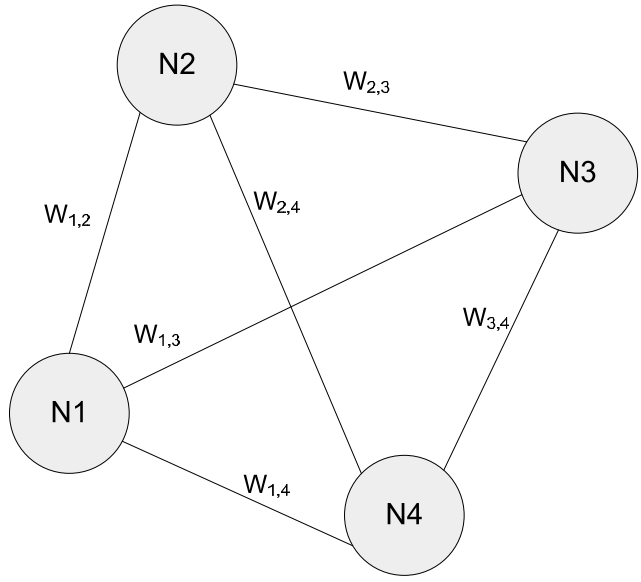


Fig.7-18 Red de Hopfield

Las redes de aprendizaje no supervisado se entrenan a partir de una serie de patrones que son introducidos en el sistema. Se espera que, una vez entrenadas, al introducir un patrón cualquiera, la red será capaz de identificar el más parecido y devolver en la salida el patrón original. Es por eso que se utilizan en aplicaciones como el reconocimiento de caracteres, reconocimiento de caras en fotografías, clasificación de patrones de conducta, etc.

Desde el punto de vista de la implementación es muy cómodo representar los pesos entre neuronas mediante una matriz bidimensional W[m,n] que será simétrica, ya que el peso entre ambas neuronas es el mismo con independencia del sentido en que se considere la relación. Además, la diagonal principal se mantiene vacía ya que una neurona no está conectada consigo misma.

© RC Libros

	N1	N2	N3	N4
N1		W1,2	W1,3	W1,4
N2	W1,2		W2,3	W2,4
N3	W1,3	W2,3		W3,4
N4	W1,4	W2,4	W3,4	

Fig. 7-19 Representación de la matriz de pesos

Haciendo uso de la analogía física presentada más arriba, definiremos la energía entre dos neuronas como:

$$energía_{i,j} = W_{i,j} \times x_i \times x_j$$

Donde W es el peso entre las dos neuronas y x el valor actual de activación de cada neurona, que puede tener dos valores posibles: +1 para indicar que la neurona está activada y -1 para indicar que no está activada. La energía total del sistema es la suma de todos los pares de neuronas y cuando entrenamos la red con una entrada, se busca un mínimo local de energía cercano a dicha entrada.

En el listado 7-4 se muestra la implementación de una red de Hopfield capaz de reconocer los caracteres 0, 1 y 2. En un programa real deberíamos ser capaces de leer los caracteres a partir de una imagen, pero para simplificar el programa introduciremos los caracteres representando los píxeles encendidos con la letra o y los píxeles apagados con un punto en una matrix de 5x4. El carácter 1 tiene el siguiente aspecto:

© RC Libros

```
.oo..

..o..

..o..

..o..
```

Para representar los caracteres reconocibles usamos una cadena de 20 caracteres en la que cada 5 de ellos corresponden con una fila. Usamos las funciones *graf_a_valores()* y *valores_a_graf()* para convertir la cadena de caracteres a valores manejables por el programa, y viceversa.

La función *discretizar()* nos asegura que los valores de entrada siempre tendrán un valor +1 o -1. La función principal es *entrenar()* que recibe una serie de patrones con los que crea la matriz de pesos. Esta función también crea un array con las sumas de las líneas de la matriz de pesos, que será usada a la hora de calcular el diferencial de energía entre la entrada y los valores usados durante el entrenamiento. Esto se hace en la función *denergia()*. Finalmente, la función *clasificar()* utiliza la función *denergía()* para recuperar el patrón correspondiente.

```
# Red de Hopfield
# reconoce los caracteres 0,1 y 2

# transformar representación gráfica en valores -1 y +1
def graf_a_valores(datos):
  datos=datos.replace(' ', '') # quitar espacios
  salida=[]
  for i in range(len(datos)):
   if datos[i]=='.':
    salida.append(-1.0)
   else:
    salida.append(1.0)
  return salida

# transformar valores -1 y +1 a representación gráfica
def valores_a_graf(datos):
  salida=''
```


© RC Libros

```
  cont=0
  for i in datos:
    cont=cont+1
    if i==-1.0:
      salida=salida+'.'
    else:
      salida=salida+'o'
    if cont%5==0:
      salida=salida+'\n' # salto de línea
  return salida


# calcula diferencial de energía
def denergia(linea):
  global n_entradas, pesos, nodos_entrada, sum_lin_pesos
  temp=0.0
  for i in range(n_entradas):
    temp=temp+(pesos[linea][i])*(nodos_entrada[i])
  return 2.0*temp-sum_lin_pesos[linea]

# devuelve el valor -1 o 1
def discretizar(n):
  if n<0.0:
    return -1.0
  else:
    return 1.0

# entrenamiento de la red
def entrenar(datos_ent):
  global n_entradas, pesos, nodos_entrada, sum_lin_pesos
  # actualizamos pesos
  for i in range(1,n_entradas):
    for j in range(i):
      for k in range(len(datos_ent)):
```

```
        datos=datos_ent[k]
        t=discretizar(datos[i])*discretizar(datos[j])
        temp=float(int(t+pesos[i][j])) # truncar decimales
        pesos[i][j]=temp
        pesos[j][i]=temp  # es una matriz simetrica

  # actualizamos suma de las líneas de la matriz de pesos
  for i in range(n_entradas):
    sum_lin_pesos[i]=0.0
    for j in range(i):
      sum_lin_pesos[i]=sum_lin_pesos[i]+pesos[i][j]


# clasificar patrón de entrada
def clasificar(patron, iteraciones):
  global n_entradas, pesos, nodos_entrada, sum_lin_pesos
  nodos_entrada=patron[:]
  for i in range(iteraciones):
    for j in range(n_entradas):
      if denergia(j)>0.0:
        nodos_entrada[j]=1.0
      else:
        nodos_entrada[j]=-1.0
  return nodos_entrada


if __name__ == '__main__':
  datos_ent = [
    graf_a_valores('.ooo. \
                    o...o \
                    o...o \
                    .ooo.'),
```


© RC Libros

```
  graf_a_valores('.oo.. \
                  ..o.. \
                  ..o.. \
                  ..o..'),

  graf_a_valores('ooooo \
                  ...o. \
                  .o... \
                  ooooo')
]

n_entradas=20  # número nodos en la red
nodos_entrada=[0.0]*n_entradas
sum_lin_pesos=[0.0]*n_entradas
# crear matriz de pesos
pesos=[]
for id in range(n_entradas):
  pesos.append([0.0]*n_entradas)

entrenar(datos_ent)
# intentar reconocer el carácter distorsionado
caracter=graf_a_valores('.oo.o \
                         ..o.. \
                         o.o.. \
                         ..o..')
reconocido=clasificar(caracter, 5)

print 'Caracter introducido:'
print valores_a_graf(caracter)
print 'Caracter reconocido:'
print valores_a_graf(reconocido)
```

Listado. 7-4 hopfield.py

© RC Libros

```
Caracter introducido:

.oo.o

..o..

o.o..

..o..


Caracter reconocido:

.oo..

..o..

..o..

..o..
```

Es decir, que a pesar del ruido introducido en la imagen, el programa puede reconocer que se trata del carácter correspondiente al número 1.

El programa puede entrenarse con todas las letras del alfabeto, e incluso usar diferentes tipos de letras para que sea capaz de reconocer cualquier texto. Evidentemente, habría que crear un procedimiento que fuera capaz de extraer los caracteres desde una imagen tanto para realizar el entrenamiento con texto real como para introducir el texto que se quiere reconocer.

© RC Libros

A El lenguaje Python

INTRODUCCIÓN

En principio, cualquier lenguaje puede utilizarse en Inteligencia Artificial, sin embargo, algunos se adaptan mejor que otros, sobre todo aquellos que son capaces de manejar complejas estructuras de datos. En este libro se ha decidido finalmente usar Python, un lenguaje creado por Guido van Rossum, ya que nos ofrece la flexibilidad necesaria y a la vez es un lenguaje con una curva de aprendizaje muy rápida. Frente a otros lenguajes, como Java o C, Python tiene la ventaja de ser un lenguaje interpretado y no necesita una compilación previa, lo que nos facilita el ciclo de programación y pruebas. A cambio, el rendimiento de la aplicación puede verse afectado. La primera vez que Python ejecuta una aplicación, genera un código intermedio al estilo de Java que hace que en subsiguientes ejecuciones el rendimiento y la velocidad de ejecución sean mejores.

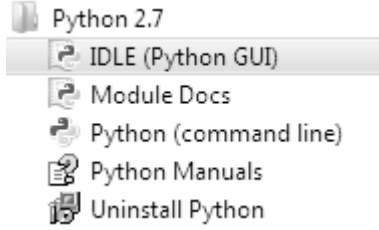
Python es un lenguaje de tipado dinámico, es decir, que no necesitamos declarar el tipo de cada variable que utilizaremos, sino que se inferirá automáticamente en el momento de utilizarla por primera vez. También es un lenguaje fuertemente tipado, lo que significa que, una vez que una variable ha sido utilizada y tiene asignado un tipo concreto, ya solo puede contener datos de ese tipo.

A pesar de ser un lenguaje muy utilizado para crear scripts de sistemas, Python es un lenguaje de propósito general que puede utilizarse también para crear complejos programas orientados a objeto y con interfaces gráficas de usuario. Al ser un lenguaje multiplataforma podremos crear aplicaciones que funcionarán en multitud de sistemas operativos, incluso en más plataformas que el propio lenguaje Java.

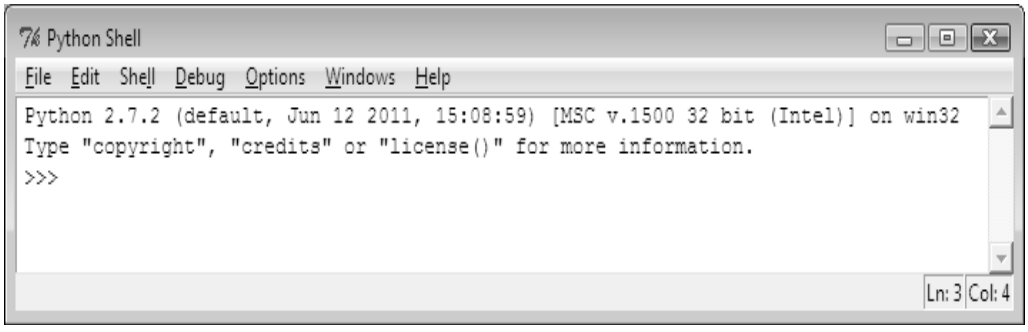
La instalación de Python es de lo más sencillo, solo hay que visitar la web http://www.python.org/download y descargar la versión correspondiente a su sistema operativo.

El intérprete interactivo

Una de las ventajas de Python es que es posible utilizarlo de modo inmediato desde su intérprete interactivo llamado IDLE, que puede ser lanzado desde el menú de inicio de Windows. Desde Linux u otro sistema operativo puede ser ejecutado desde la línea de comando simplemente tecleando el comando *python*.



El intérprete interactivo es una herramienta muy poderosa que nos permite ejecutar instrucciones directamente sin necesidad de escribirlas dentro de un programa. En la siguiente figura vemos el aspecto de la herramienta IDLE en Windows.



© RC Libros

Para probar el intérprete teclearemos la siguiente instrucción dentro de IDLE o del intérprete interactivo y pulsaremos Enter.

```
print "Saludos desde Python"
```

El intérprete responderá con: *Saludos desde Python*, lo que querrá decir que todo funciona correctamente.

También podemos añadir las instrucciones a un programa completo, para lo que será necesario añadirlas a un archivo con extensión .py y ejecutarlas con el comando *python*. Si añadimos la instrucción anterior a un archivo llamado saludo.py, podremos ejecutarlo haciendo doble clic sobre el archivo si estamos en Windows, aunque es preferible ejecutarlo desde la línea de comandos del sistema operativo escribiendo:

```
python saludo.py
```

Desde Linux podemos ejecutarlo escribiendo directamente ./saludo.py si añadimos la siguiente línea al principio del programa.

```
#!/usr/bin/env
```

Y además hacemos el archivo ejecutable con

```
chmod +x saludo.py
```

El primer programa

Python tiene una forma peculiar de organizar el código fuente, ya que la indentación de las líneas forma parte de la semántica del lenguaje. Esto puede parecer algo confuso al principio, pero a la postre es una ventaja, ya que obliga a identar y estructurar el código fuente de una manera adecuada.

Vamos a comenzar con el siguiente programa que nos permitirá echar un breve vistazo a los elementos más importantes del lenguaje.

© RC Libros

```
# ejemplo 1


print "Introduce edad:"
edad = input()
dias=edad*365
print "Tienes " + str(dias) + " dias"
```

Listado. A-1 ejemplo1.py

Este programa pide que introduzca su edad por el teclado y seguidamente se la muestra en días (aproximadamente, sin contar años bisiestos). Es un programa muy simple pero que nos ayudará a empezar con Python.

La primea línea es un comentario. Todo lo que hay escrito tras el carácter # es ignorado por el lenguaje, pero permite poner comentarios al programa comprensibles por humanos.

A *print* ya lo conocemos, nos permite mostrar por pantalla una cadena de caracteres o *string* como se conoce en Python. Hablaremos de los strings unas líneas más abajo.

La función *input()* permite introducir información por teclado (solo valores válidos para Python). Esta función recoge la edad tecleada y la almacena en la variable *edad*.

Podemos imaginar una variable como una pequeña zona de memoria donde se almacena un dato para poder usarlo más tarde. A cada una de estas zonas de memoria le asignamos un nombre (edad) y un valor (el valor introducido por teclado). Cada variable tiene asociado un tipo, que define qué tipo de información puede almacenar. Los tipos básicos en Python son:

© RC Libros

Nombre	Descripción	Ejemplo
String	Cadena de caracteres	“python”
Int	Número entero	-100
Long	Número entero grande. Se debe poner L tras el número.	3987268483L
Float	Número real	3.141592
Complex	Número complejo	5+3j

Las operaciones que podemos realizar con cada tipo de dato dependen de la clase a la que pertenezcan. En general podemos operar matemáticamente con números de diferente tipo y Python realizará las conversiones pertinentes. Aunque como veremos seguidamente, es posible indicar a Python qué conversiones hacer explícitamente.

El número de días se ha calculado multiplicando la variable *edad* por 365 y asignando el resultado de la operación a la variable *dias*. Las operaciones matemáticas básicas que podemos realizar en Python son:

Operador	**Descripción**	**Ejemplo**
+	Adición.	10 + 15 = 25
-	Substracción.	15 – 10 = 5
*	Multiplicación.	3 * 4 = 12
/	División.	10 / 5 = 2
%	Módulo (resto).	10 % 2 = 0
**	Exponente.	5 ** 2 = 25
//	División con trucado de decimales. Se quitan los decimales tras la coma.	9 // 2 = 4

© RC Libros

El signo = también es un operador, en este caso de asignación, ya que almacena el resultado de la operación en una variable. Además del signo igual, hay otros operadores muy utilizados que se resumen en la siguiente tabla.

Operador	Descripción	Ejemplo
=	Asignación simple. Asigna el valor de la expresión de la derecha a la variable de la izquierda.	v = 3+5
+=	Asigna la suma del valor de la expresión de la derecha y de la variable de la izquierda.	v += 5 (equivale a v=v+5)
-=	Asigna la diferencia del valor de la expresión de la derecha y de la variable de la izquierda.	v -= 5 (equivale a v=v-5)
*=	Asigna el producto del valor de la expresión de la derecha y de la variable de la izquierda.	v *= 5 (equivale a v=v*5)
/=	Asigna el cociente del valor de la expresión de la derecha y de la variable de la izquierda.	v /= 5 (equivale a v=v/5)
%=	Asigna el módulo (resto) del valor de la expresión de la derecha y de la variable de la izquierda.	v %= 5 (equivale a v=v%5)
**=	Asigna la potencia del valor de la expresión de la derecha y de la variable de la izquierda.	v **= 5 (equivale a v=v**5)
//=	Asigna el cociente truncado del valor de la expresión de la derecha y de la variable de la izquierda.	v //= 5 (equivale a v=v//5)

© RC Libros

En la línea siguiente se utiliza de nuevo *print* para mostrar el número de días que hay almacenado en la variable *días*. Hay dos aspectos interesantes en los que debemos fijarnos.

Se ha utilizado de nuevo el simbolo +, pero en esta ocasión no con la intención de sumar dos valores, sino que, cuando se usa con cadenas de caracteres, su función es concatenarlas (unirlas) en un solo *string*. Por ejemplo:

```
saludo="hola "+"mundo"
```

Si mostramos el contenido de la variable saludo, observaremos que contiene el *string* "hola mundo". Hay que tener en cuenta que el operador + solo funciona si ambos operandos son cadenas de caracteres. No se puede concatenar un *string* y un número. La siguiente operación debería dar un error.

```
saludo="salu"+2
```

La única manera de hacerlo es transformando el número 2 en una cadena de caracteres. Esto puede realizarse mediante la función str(), que transforma el número entero en un string.

```
saludo="salu"+str(2)
```

Ahora, la variable saludo contiene la cadena "salu2". A la operación de transformar una variable de un tipo de dato a otro la denominamos moldeado o, en inglés, *casting*. Como vemos en la siguiente tabla, existen funciones específicas para transformar a diferentes tipos de datos.

función	**Descripción**
int(x [,base])	Convierte x a entero. Podemos especificar la base de numeración si x es una cadena. La base será decimal si no se indica.
long(x [,base])	Convierte x a entero largo. Podemos especificar la base de numeración si x es una cadena. La base será decimal si no se indica.
float(x)	Convierte x a un número real.
str(x)	Convierte x a un string.
tuple(x)	Convierte x a una tupla.

© RC Libros

list(x)	Convierte x a una lista.
set(x)	Convierte x a un conjunto.
chr(x)	Convierte un entero a un carácter.
unichr(x)	Convierte un entero a un carácter unicode.
ord(x)	Convierte un carácter a su correspondiente valor ASCII.
hex(x)	Convierte un entero en una cadena hexadecimal.
oct(x)	Convierte un entero en una cadena octal.

He aquí algunos ejemplos de conversión de valores:

```
>>> float(4)

4.0

>>> int (3.3)

3

>>> str(5.25)

'5.25'

>>> ord('a')

97

>>> hex(12)

'0xc'
```

Cadenas de caracteres

Las cadenas de caracteres o *strings* permiten almacenar texto, pero además facilitan realizar operaciones sobre él. Podemos definir el texto de una cadena de cualquiera de las siguientes tres formas:

© RC Libros

```
texto = 'soy un string'

texto = "soy un string"

texto = """soy

   un

   string multilínea"""
```

En las dos primeras usamos comillas simples o dobles, y son equivalentes. La tercera forma usa tres comillas dobles seguidas para marcar el inicio y el final de la cadena, y permite crear strings que ocupan más de una línea.

La elección de una u otra forma depende de la situación. Por ejemplo, si prevemos que dentro del mismo texto se van a usar comillas dobles, es más interesante usar comillas simples, ya que la siguiente línea es errónea para Python:

```
texto = "me dijo: "a ver si vienes""
```

En este caso habría que usar comillas simples:

```
texto = 'me dijo: "a ver si vienes"'
```

A la inversa ocurre igual. Si se van a usar comillas simples, conviene crear la cadena con comillas dobles. Otra opción es usar una secuencia de escape dentro de la cadena, lo que nos permite añadir caracteres especiales. Las secuencias de escape empiezan con la barra invertida \. Por ejemplo \" significa que queremos poner una comilla doble dentro de la cadena. De esta manera, la siguiente línea es completamente válida para Python:

```
texto = "me dijo: \"a ver si vienes\""
```

Ahora, la variable texto contiene la cadena: me dijo "a ver si vienes".

En la siguiente tabla podemos ver las secuencias de escape válidas en las cadenas de caracteres de Python.

© RC Libros

Secuencia de escape	Significado
[enter]	Usada para continuar en la siguiente línea
\\	Barra invertida (\)
\'	Comilla simple (')
\"	Comilla doble(")
\a	Código ASCII 0x08 (BEL)
\b	Código ASCII borrado (BS)
\f	Código ASCII alimentación formulario (FF)
\n	Código ASCII alimentación de línea (LF)
\N{*nombre*}	Carácter llamado *nombre* en Unicode
\r	Código ASCII retorno de carro (CR)
\t	Código ASCII tabulador horizontal (TAB)
\u*xxxx*	Carácter Unicode con el código hex. xxxx (16 bits)
\U*xxxxxxxx*	Carácter Unicode con el código hex. xxxx (32 bits)
\v	Código ASCII tabulador vertical (VT)
ooo	Carácter con el valor octal ooo
\x*hh*	Carácter con el valor hexadecimal hh

Internamente, un *string* es un objeto de Python. Más adelante definiremos lo que es un objeto, pero por ahora nos bastará saber que sobre un objeto podemos realizar operaciones, y en el caso de las cadenas, estas operaciones nos permiten realizar tareas como buscar caracteres dentro de ellas, pasarlas a mayúsculas o minúsculas, eliminar espacios, etc. Para realizar estas operaciones se utiliza el operador punto (.) y la operación (que más adelante llamaremos método) que queremos realizar. Veamos algunos ejemplos que nos ayudarán a entender cómo funciona.

```
>>> texto="Hola, soy un string"

>>> texto.count('o') # cuenta el número de 'o'

2

>>> texto.find('un') # busca la posición de 'un'

10

>>> texto.replace('un', 'el') # cambia 'un' por 'el'
```


© RC Libros

```
'Hola, soy el string'

>>> texto.upper() # conviente a mayusculas

'HOLA, SOY UN STRING'
```

Estos son solo unos ejemplos, el número de operaciones que podemos realizar sobre un string es amplio, y si queremos obtener información podemos pedírsela a Python usando el comando.

```
help (str)
```

Y nos informará de todas las operaciones que podemos realizar con *strings*.

Además de las secuencias de escape, podemos utilizar el operador de formato %. Gracias a este operador es posible tener un mayor control sobre el formato del texto. La tabla siguiente muestra los operadores de conversión disponibles.

Conversión	Significado
d	Entero decimal con signo.
i	Entero decimal con signo.
o	Octal sin signo.
u	Decimal sin signo.
x	Hexadecimal sin signo (minúsculas).
X	Hexadecimal sin signo (mayúsculas).
e	Notación científica exponencial (minúsculas).
E	Notación científica exponencial (mayúsculas).
f	Formato decimal punto flotante.
F	Formato decimal punto flotante.
g	Como ‘e’ si el exponente es mayor que -4 o menor que la precisión. Si no es como ‘f’.
G	Como ‘E’ si el exponente es mayor que -4 o menor que la precisión. Si no es como ‘F’.
c	Un solo carácter o su código ASCII en formato entero.
r	Convierte un objeto a string usando la función repr().
s	Convierte un objeto a string usando la función str().

© RC Libros

Algunos ejemplos de uso de estos operadores son:

```
>>> lenguaje='python'

>>> "yo utilizo %s" % lenguaje

'yo utilizo python'

>>> edad=30

>>> "tengo %d días" % (edad*365)

'tengo 10950 días'

>>> "tengo %f dias" % (edad*365)

'tengo 10950.000000 dias'
```

Junto con los operadores que acabamos de ver podemos utilizar los siguientes modificadores.

Modificador	Significado
#	La conversión de valor usará el método alternativo.
0	Indica cuántas cifras tendrá el valor impreso. Se rellena con 0.
-	Indica cuántas cifras tendrá el campo justificado a la izquierda.
	Indica cuántas cifras tendrá el campo justificado a la derecha.
+	Se añade el signo del valor (+ o -)

Algunos ejemplos:

```
>>> "tengo %03d años" % 30 # tres cifras

'tengo 030 años'

>>> "tengo %-5d años" % 30 #  justifica  5  caracteres  a  la
izquierda

'tengo 30    años'
```


© RC Libros

```
>>> "tengo % 5d años" % 30 # justifica 5 caracteres a la derecha

'tengo    30 años'
```

Finalmente, para terminar el repaso a los *strings*, hay que decir que es posible acceder a los caracteres que lo forman de forma independiente tal y como observamos en el siguiente ejemplo:

```
>>> lenguaje='python'

>>> lenguaje[1] # carácter 1 del string

'y'
```

Hay que tener en cuenta que el primer carácter del *string* es el 0.

Estructuras de control

Las estructuras de control que nos ofrece cualquier lenguaje estructurado permiten controlar el flujo de la ejecución del programa. El primer programa de ejemplo que hemos mostrado es muy simple, y su ejecución se produce de forma secuencial, es decir, una línea tras otra.

Existen dos clases de estructuras de control: estructuras condicionales y estructuras repetitivas. Hay una tercera, que es la de salto, pero que obviaremos ya que su uso no es una buena práctica en la programación con lenguajes estructurados de alto nivel.

La estructura condicional por excelencia es *if*, que es la construcción que permite a los programas tomar decisiones. El siguiente listado muestra un programa que hace uso de este tipo de estructura.

```
# ejemplo if


print "Introduce edad:"
edad = input()
```

```
edad=int(raw_input("Introduce tu edad: "))
if edad>=18:
  print "Ya eres mayor de edad"
else:
  print "Aun no eres mayor de edad"
```

Listado. A-2 if.py

El listado anterior solicita la edad por teclado y responde en función de si la edad introducida es menor a 18 o no. La construcción *if* es la que permite mostrar una frase u otra. La forma general de esta estructura de control es la siguiente.

If <condición>:

acción

[elif <condición>:

acción

…

else:

acción]

Primero se evalúa la primera condición, y si se cumple se ejecuta la primera acción y la ejecución sale de la estructura if (solo se ejecutará la acción que primero cumpla la condición). En caso de no cumplirse la primera condición se continúa con las siguientes (puede usarse *elif* parar poner tantas condiciones como sea necesario). En caso de no cumplirse ninguna, se ejecuta la acción que hay tras *else*, que puede verse como si fuera la acción por defecto.

La porción encerrada entre corchetes (los corchetes no se ponen en el programa) significa que esta parte es opcional, es decir, podemos prescindir de *elif* o *else* si no lo necesitamos (en el ejemplo no hemos usado *elif*).

Es importante no olvidar los dos puntos tras *if*, *elif* y *else* y que las acciones van identadas usando un tabulador. En caso contrario Python dará un error.

© RC Libros

Las condiciones en Python pueden evaluarse usando los operadores de comparación de la siguiente tabla (supondremos dos variables con los valores a = 10 y b = 20). El resultado de una operación de comparación siempre es un valor booleano que puede ser verdadero (*True*) o false (*False*). La acción se llevará a cabo si tras evaluar la condición el resultado es verdadero.

Operador	Descripción	Ejemplo
==	Devuelve el valor verdadero si a es igual a b.	(a == b) es falso.
!=	Devuelve el valor verdadero si a es distinto a b.	(a != b) es verdadero.
<>	Igual que !=.	(a != b) es verdadero.
>	Devuelve el valor verdadero si a es estrictamente mayor a b.	(a > b) es falso.
<	Devuelve el valor verdadero si a es estrictamente menor a b.	(a < b) es verdadero.
>=	Devuelve el valor verdadero si a es mayor o igual a b.	(a >= b) es falso.
<=	Devuelve el valor verdadero si a es menor o igual a b.	(a <= b) es verdadero.

Pueden combinarse comparaciones usando los operadores lógicos AND, OR y NOT.

Operador	Descripción	Ejemplo
and	Devuelve verdadero si ambas cláusulas son verdaderas.	(a==10 and b==20) es verdadero.
or	Devuelve verdadero si al menos una cláusula es verdadera.	(a==20 or b==20) es verdadero
not	Devuelve verdadero si la cláusula que sigue es falsa y falso en caso contrario (cambia el valor).	not (a==10 and b==20) es falso.

© RC Libros

Las dos siguientes estructuras de control que vamos a presentar pertenecen al tipo repetitivo, que permiten realizar bucles en acciones repetitivas dentro del programa. La primera de ellas es *while*.

```
# ejemplo while


print "Introduzca una palabra:"
palabra=raw_input()
i=len(palabra)
while i>0:
  print palabra[i-1],
  i=i-1
```

Listado. A-3 while.py

El programa del listado anterior solicita una palabra por teclado y la muestra invirtiendo el orden de sus caracteres (si introducimos Python se mostrará nohtyp). Observemos que en vez de input() hemos usado la función raw_input(), que permite la entrada de texto. Cualquier cosa que introduzcamos por teclado será almacenado como una cadena de caracteres.

En la siguiente línea usamos la función *len()* que retorna el número de caracteres que componen la cadena (en realidad el número de elementos de cualquier secuencia) que se almacena en la variable i.

Seguidamente se utiliza un bucle while que se repetirá mientras se cumpla la condición que le sigue (i>0). Es decir, mientras que i sea mayor que cero imprimimos el carácter i-ésimo (menos uno ya que recordemos que el primer carácter ocupa el lugar 0) y seguidamente decrementamos i en una unidad (i hace la función de contador inverso).

El formato general de while es el que sigue.

```
while <condición>:
   acción 1
[else:
   acción 2]
```


© RC Libros

De nuevo hay que tener muy presente la indentación y el uso de los dos puntos tras *while* y *else*. El bloque *acción 1* se ejecutará repetitivamente hasta que se cumpla la condición que hay tras *while* (llamada condición de salida). Si se da la condición de salida, se ejecutará el bloque *acción 2* que hay tras *else*. Este bloque es opcional.

Hay dos instrucciones que permiten modificar el comportamiento de *while*. Se tratan de *break* y *continue*. Si dentro del bloque de acciones se encuentra la instrucción *break*, el flujo del programa sale del bucle y sigue por la instrucción siguiente. La instrucción *continue* es parecida, pero en vez de salir del bucle, deja de ejecutar el resto de las instrucciones internas del bucle y continúa desde el principio en la siguiente iteración del bucle (salta al principio del *while*).

La otra estructura repetitiva que nos queda por presentar es el bucle *for*. Si la conoce de otros lenguajes de programación, este bucle es ligeramente diferente en Python. En concreto, su misión es la de recorrer una secuencia de elementos. Aún no conocemos las listas, pero adelantaremos que son una colección de elementos de cualquier tipo, y que se declaran poniéndolas entre corchetes. Por ejemplo:

```
>>> personas=['juan', 'ana', 'antonio']

>>> print personas

['juan', 'ana', 'antonio']
```

Hablaremos de las listas un poco más adelante, pero por ahora vamos a ver cómo recorrer una usando el bucle *for*.

```
# ejemplo for

personas=['juan','ana','antonio']
for p in personas:
  print p
```

Listado. A-4 for1.py

En este listado se define una lista de elementos (*strings*) que se almacenan en la lista *personas*. Seguidamente el bucle *for* recorre la lista almacenando en cada vuelta el siguiente *string* de la lista en la variable p, hasta que se haya completado toda la lista. Por cada elemento de la lista, el programa usa *print* para mostrar cada uno de los nombres.

El formato general del bucle *for* es:

```
for <variable> in <secuancia>:

    acciones
```

Es decir, que *for* recorre una secuencia almacenando en cada iteración el siguiente dato de la secuencia en la variable y ejecuta el código del bloque de acciones. El bucle se repite tantas veces como elementos haya en la secuencia.

Una función muy utilizada junto a *for* es *range()* que permite generar secuencias numéricas y así hacer que *for* se comporte de forma similar a como lo hace en otros lenguajes. Veamos unos ejemplos del uso de *range()* y seguidamente un programa de ejemplo que hace uso de ella.

```
>>> range(9)

[0, 1, 2, 3, 4, 5, 6, 7, 8]

>>> range(2,8)

[2, 3, 4, 5, 6, 7]

>>> range(1,9,2)

[1, 3, 5, 7]

>>> range(10,0,-1)

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

>>> range(10,0,-2)

[10, 8, 6, 4, 2]
```


© RC Libros

Esta función puede recibir uno, dos o tres parámetros. Si solo ponemos un parámetro n, genera una lista de n números empezando por el 0 (de 0 a n-1). Con dos parámetros m y n indicamos el número inicial y el final en la lista (de n a m-1). Finalmente, el tercer parámetro nos permite indicar el incremento entre un número y el siguiente. Por ejemplo range(1,9,2) mostrará los números impares entre 1 y 8. Cualquiera de los tres parámetros puede tener valores negativos.

```
# ejemplo for2
import string

print "Introduzca tabla del 0 a 9:"
n=raw_input()
if len(n)==1 and n in string.digits:
  tabla=int(n)
  for i in range(10):
    print str(i)+" x "+str(n)+" = "+str(i*tabla)
```

Listado. A-5 for2.py

El código anterior hace uso de *for* y *range*, pero introduce algunos conceptos nuevos. Este programa es capaz de generar las tablas de multiplicar del 0 al 9, pero hemos añadido un control extra para evitar que el usuario pueda solicitar una tabla no permitida; es decir, controlamos que el número introducido sea un número válido entre 0 y 9.

En el *if* que controla la validez de la variable n comprobamos que el tamaño de la entrada sea de un solo carácter (una sola cifra) usando la función *len()*. En la segunda parte del *if* comprobamos que es un dígito válido.

Para realizar esta comprobación usamos una constante del módulo *string* llamada *digits* (una constante es un valor fijo que no cambia durante la ejecución del programa). Para poder usar funciones o constantes de un módulo hay que importarlas antes. Esto lo hacemos al principio del programa usando la instrucción *import*. En general podemos importar módulos usando:

```
import modulo 1,[modulo 2, …, modulo n]

from modulo import elemento 1[,…, elemento n]

from modulo import *
```

La primera y la última forma permiten importar uno o más módulos completamente. La segunda ofrece la posibilidad de importar solo los elementos del módulo que interesen. Profundizaremos en los módulos un poco más adelante.

Para ver qué contiene la constante *digits*, tecleamos en la línea de comando:

```
>>> import string

>>> string.digits

'0123456789'
```

Es decir, que la constante es un *string* con todos los dígitos válidos. En la condición hemos usado el operador *in* para comprobar si el dígito que hemos introducido está dentro del *string*. La siguiente tabla describe los operadores de pertenencia.

Operador	Descripción	Ejemplo
in	Toma el valor verdadero si el valor está dentro de la secuencia.	(‘1’ in ‘0123456789’) es verdadero
not in	Toma el valor verdadero si el valor no está dentro de la secuencia.	(‘ana’ not in ['juan', 'ana', 'antonio']) es falso

Obsérvese cómo se realizan las conversiones entre tipos necesarias usando *str()* e *int()*, ya que *raw_input()* toma la entrada desde teclado como un *string* y no como un número.

© RC Libros

Secuencias

Uno de los puntos fuertes de Python que lo dotan de una gran expresividad comparado con otros lenguajes es su capacidad para trabajar con secuencias de datos de una forma natural y eficiente. En otros lenguajes hay que recurrir a complejas APIs para el manejo de estructuras de datos. A continuación, veremos qué son y cómo funcionan las listas, las tuplas, los conjuntos y diccionarios, que son los tipos de secuencia que Python pone a nuestra disposición.

Ya hemos tenido un primer contacto con las listas cuando hemos tratado el bucle *for*. De hecho, los *strings* son una especie de lista de caracteres. Recordemos cómo definíamos una lista:

```
>>> semana=['Lunes', 'Martes', 'Miércoles', 'Jueves', \
'Viernes']
```

La barra invertida \ se utiliza en este caso para indicar a Python que el código continúa en la siguiente línea. Las listas, al igual que los *strings*, son objetos sobre los que podemos realizar operaciones. A continuación, se muestran las posibles operaciones y algunos ejemplos.

Método	Descripción	Ejemplo
append(x)	Añade el elemento x al final de la lista.	semana.append(‘Sábado’)
extend(l)	Añade la lista l al final de la lista.	semana.extend([‘Sábado’,’Domingo’])
insert(i, x)	Inserta el elemento x en la posición i de la lista.	semana.insert(5,'Sábado')
remove(x)	Elimina el elemento x de la lista.	semana.remove('Sábado')
pop([i])	Elimina y retorna con el elemento de la posición i. Si no se indica i la operación se realiza sobre el último elemento de la lista.	semana.pop() devuelve 'Viernes'
index(x)	Devuelve el índice cuyo valor sea x.	semana.index('Jueves') devuelve 3

© RC Libros

count(x)	Devuelve el número de ocurrencias del elemento x en la lista.	semana.count('Lunes') devuelve 1
sort()	Ordena los elementos de una lista.	semana.sort() devuelve ['Jueves', 'Lunes', 'Martes', 'Miércoles', 'Viernes']
reverse()	Invierte el orden de los elementos de una lista.	semana.reverse() devuelve ['Viernes', 'Jueves', 'Miércoles', 'Martes', 'Lunes']

```
>>> numeros=['1','2','3','4']
>>> numeros.append('5')
>>> numeros
['1', '2', '3', '4', '5']
>>> numeros.pop()
'5'
>>> numeros
['1', '2', '3', '4']
>>> numeros.index('3')
2
>>> numeros.reverse()
>>> numeros
['4', '3', '2', '1']
```

Además de los métodos *remove()* y *pop()*, podemos usar la instrucción *del* para borrar elementos de la lista indicando su posición en esta.

© RC Libros

```
del lista[1] # borra el elemento 1 de la lista

del lista[2:4] #borra del elemento 2 al 4 de la lista

del lista[3:] # borra los elementos a partir de la posición 3

del lista[:3] # borra los elementos hasta la posición 3

del lista[:] # borra toda la lista
```

También es posible acceder a elementos o grupos de estos de la lista usando corchetes de la misma forma que en los *strings*. He aquí unos ejemplos:

```
>>> numeros=['1','2','3','4']

>>> numeros[2] # número en posición 2

'3'

>>> numeros[2:4] # números entre posición 3 y 4

['3', '4']

>>> numeros[2:] # números a partir de la posición 2

['3', '4']

>>> numeros[:3] # números hasta la posición 3

['1', '2', '3']
```

De la misma forma que accedemos a los elementos para leerlos, podemos modificarlos:

```
>>> numeros[1]='a'

>>> numeros

['1', 'a', '3', '4']
```

Las listas pueden estar compuestas de cualquier tipo de datos, incluso otras listas. He aquí un ejemplo de lista de listas, que es equivalente a un *array* bidimensional en otros lenguajes de programación.

© RC Libros

```
>>> datos=[[1,2,3],[4,5,6],[7,8,9]]

>>> datos

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> datos[2][2]

9
```

Las tuplas son similares a las listas en el sentido de que pueden almacenar series de elementos. Estas se crean usando paréntesis o simplemente se separan por comas:

```
>>> a=(1,2,3)

>>> b=4,5,6

>>> b

(4, 5, 6)
```

Como puede observarse Python pone automáticamente los paréntesis a los elementos de la tupla *b*.

A diferencia de las listas, las tuplas son inmutables, es decir, una vez creadas no es posible modificarlas. Si intentamos modificarlas este es el resultado que obtenemos:

```
>>> a=(1,2,3)

>>> a.append(4)

Traceback (most recent call last):

  File "<interactive input>", line 1, in <module>

AttributeError: 'tuple' object has no attribute 'append'
```

Esto quiere decir que no podremos utilizar ninguna operación sobre ella que modifique su contenido, pero sí aquellas que no lo hacen, como *count()*.

© RC Libros

Los conjuntos son una serie de datos no ordenados (no guardan un orden concreto) y que además tienen la particularidad de que no tienen elementos repetidos. Se crean con *set*:

```
>>> numeros=[1,2,3,4,5,3,4,5]

>>> conjunto_numeros=set(numeros)

>>> conjunto_numeros

set([1, 2, 3, 4, 5])
```

Son muy útiles a la hora de asegurarnos de que no almacenamos información repetida. Es posible crear conjuntos a partir de cadenas de caracteres:

```
>>> texto="sin repeticiones"

>>> letras=set(texto)

>>> letras

set([' ', 'c', 'e', 'i', 'n', 'o', 'p', 'r', 's', 't'])
```

Además, se pueden realizar las operaciones clásicas de los conjuntos:

```
>>> texto1=set("primer texto")

>>> texto2=set("segundo texto")

>>> texto1-texto2 # letras de texto1 que no están en texto2

set(['i', 'm', 'p', 'r'])

>>> texto1 | texto2 # letras en texto1 o en texto2

set([' ', 'd', 'e', 'g', 'i', 'm', 'n', 'o', 'p', 'r', 's',
't', 'u', 'x'])

>>> texto1 & texto2 # letras en texto1 y texto2 a la vez

set([' ', 'e', 'o', 't', 'x'])
```

© RC Libros

```
>>> texto1 ^ texto2 # letras en texto1 o texto2 pero no en
ambos

set(['d', 'g', 'i', 'm', 'n', 'p', 'r', 's', 'u'])
```

Por último tratamos los diccionarios, que en otros lenguajes son más conocidos como *arrays* asociativos. Permiten almacenar datos pero, a diferencia de las listas o las tuplas, no los referenciamos por su posición dentro de la estructura de datos, sino que le asignamos un nombre que llamamos clave. De esta forma, cuando añadimos un dato a un diccionario, hemos de añadir la clave y su valor correspondiente.

Para crear un diccionario utilizamos las llaves {} e introducimos los pares clave, y valor separados por dos puntos.

```
>>> ventas={'ratones':10, 'teclados':13, 'CDs':200,
'DVDs':150}

>>> ventas

{'CDs': 200, 'DVDs': 150, 'ratones': 10, 'teclados': 13}
```

Obsérvese que al mostrar la información almacenada en el diccionario no tiene por qué mantenerse el orden en el que lo hemos almacenado.

Hay algunas limitaciones en cuanto al uso de claves: no pueden estar repetidas. Otra limitación es que la clave tiene que ser un *string*, un número o una tupla, es decir, valores inmutables. No podemos, por lo tanto utilizar una lista (ni nada que contenga una lista) como clave.

Una vez creado el diccionario podemos añadir más elementos de la siguiente manera:

```
>>> ventas['impresoras']=2

>>> ventas

{'CDs': 200, 'DVDs': 150, 'impresoras': 2, 'ratones': 10,
'teclados': 13}
```

Al igual que con las listas podemos usar la instrucción *del* para borrar un elemento:

© RC Libros

```
>>> del ventas['impresoras']

>>> ventas

{'CDs': 200, 'DVDs': 150, 'ratones': 10, 'teclados': 13}
```

Para acceder a un valor concreto lo hacemos mediante su clave:

```
>>> ventas['ratones']

10
```

Con los diccionarios también pueden usarse los operadores de pertenencia:

```
>>> 'teclados' in ventas

True
```

Hay varias formas de acceder al contenido de un diccionario, ya que es habitual que en algún momento surja la necesidad de tener que recorrer todos sus elementos. Vamos a presentar las dos formas más comunes. En la primera usamos la función keys() como vemos en el siguiente ejemplo.

```
>>> for clave in ventas.keys():

...         print clave+" = "+str(ventas[clave])

...


teclados = 13

DVDs = 150

ratones = 10

CDs = 200
```

Alternativamente podemos usar el método *items()* o *iteritems()*.

```
>>> for clave, valor in ventas.items():
```

© RC Libros

```
...          print clave+" = "+str(valor)

...

teclados = 13

DVDs = 150

ratones = 10

CDs = 200
```

Podemos usar *ítems()* o *iteritems()* indistintamente. La diferencia radica en que *ítems()* devuelve una lista con las tuplas (clave, valor) e *iteritems()* devuelve un iterador. No entraremos en detalles, baste saber que para nuestros propósitos son equivalentes.

Funciones

Cuando se desarrollan programas de cierto tamaño interesa organizar el código para que sea lo más claro posible, no solo para que otros puedan entenderlo, sino para poder entenderlo uno mismo (es sorprendente como al cabo de unos meses podemos no ser capaces de entender un código que hemos escrito si este no está bien organizado y comentado).

Uno de los defectos del software que hay que evitar a toda costa es el uso de código redundante. Si tenemos una porción de código que realiza una operación concreta y esa misma operación ha de realizarse en distintos puntos de la aplicación, conviene separar ese código en una función reutilizable.

Una función se define con la palabra reservada *def*, asignándole un nombre y unos parámetros de entrada. En general una función se define de la siguiente forma:

```
def nombre_función([parámetro1, …, parámetroN]):

    acciones

    [return valor]
```


© RC Libros

Tanto los parámetros como la sentencia *return* son opcionales. Los parámetros son nombres de variables mediante las cuales pasaremos valores desde el programa principal a la función. La sentencia *return* nos permite devolver valores desde la función al programa principal. Obsérvese que el código de la función va indentado de la misma manera que en los bucles.

Analicemos un ejemplo sencillo. Supongamos el siguiente programa que pide tres nombres por teclado y comprueba que ninguno se repita.

```
print "introduzca tres nombres no repetidos"
nombres=[]
n=raw_input('Nombre 1:')
nombres.append(n)

n=raw_input('Nombre 2:')
if n not in nombres:
  nombres.append(n)
else:
  print "repetido"
  exit(1)

n=raw_input('Nombre 3:')
if n not in nombres:
  nombres.append(n)
else:
  print "repetido"
  exit(1)

print nombres
```

Listado. A-6 funciones1.py

La función *exit()* retorna al sistema (sale del programa) con el valor que le pasamos como parámetro. Puede ser un entero o un *string*.

Este programa es un claro ejemplo de cómo no deben hacerse las cosas. Un primer problema que apreciamos es que cuando se pide por teclado un nombre, hay que mirar en la lista a ver si ya está almacenado. Además, si queremos extender el

© RC Libros

programa para que sea capaz de solicitar cien nombres tendremos un serio problema.

En un primer momento vamos a tratar de separar el código del programa que solicita un nombre por teclado y comprueba si ya existe. Esta es una posible solución:

```
def pide_nombre(cont, nombres):
  n=raw_input('Nombre '+str(cont)+':')
  if n not in nombres:
    nombres.append(n)
  else:
    print "repetido"
    exit(1)

nombres=[]
for i in range(3):
  pide_nombre(i, nombres)

print nombres
```

Listado. A-7 funciones2.py

De entrada hemos reducido el número de líneas, pero además, si ahora queremos solicitar cien nombres en vez de tres, solo hay que cambiar el valor 3 del bucle *for* por un 100. Un detalle a tener en cuenta es que el nombre de la variable que pasamos como parámetro desde el programa principal a la función (variable *i*) no tiene por qué coincidir con el nombre que va a tener dentro de la función (variable *cont*). En este caso *cont* dentro de la función tomará el valor de *i*.

No es buena práctica tratar los errores dentro de la propia función, es mejor devolver un valor que indique si todo ha ido bien o no. Para eso vamos a usar *return*, y el control de error lo vamos a hacer ahora fuera de la función.

```
def pide_nombre(cont, nombres):
  n=raw_input('Nombre '+str(cont)+':')
  if n not in nombres:
```


© RC Libros

```
        nombres.append(n)
        return True
      else:
        return False

nombres=[]
for i in range(3):
  if not pide_nombre(i, nombres):
    exit('Repetido')

print nombres
```

Listado. A-8 funciones3.py

Ahora la función devuelve el valor verdadero (*True*) si el nombre no está repetido, y falso (*False*) en caso contrario. El *if* del bucle principal evalúa si la función tiene valor verdadero o falso y actúa en consecuencia.

Es posible asignar valores por defecto a los parámetros de una función en caso de que desde el programa principal no se pase ninguno. Esto se hace asignándole un valor en la propia cabecera de la función como puede observarse en el siguiente ejemplo:

```
def potencia(base, exponente=2):
  return base**exponente

print potencia(2,3)
print potencia(2)
```

Listado. A-9 default.py

La función *potencia()* recibe dos parámetros, una base y un exponente y devuelve la base elevada al exponente que le hemos indicado. Si no pasamos como parámetro el exponente, su valor por defecto será 2, tal y como observaremos al ejecutar el programa.

© RC Libros

Por último, hay que dejar dicho que las variables que son declaradas dentro de una función no son visibles fuera de ella, así como tampoco es posible ver desde dentro aquellas variables que hayan sido declaradas fuera de la función (salvo que sean pasadas como parámetros). Este comportamiento puede variarse usando la palabra reservada *global*, que hace que las variables que indiquemos después sean vistas dentro de la función aunque hayan sido declaradas fuera.

Clases y objetos

Python soporta el paradigma de programación orientada a objetos de una forma natural. Este paradigma busca asimilar la estructura de un programa a cómo son los objetos de la vida real. De esta forma se pretende conseguir una mayor modularidad, reutilización de código y sencillez a la hora de escribir programas.

Cuando un ingeniero diseña un coche, selecciona una serie de elementos que ya existen porque han sido fabricados por otras empresas u otras personas, y los ensambla. Por ejemplo, pude decidir usar un tipo de ruedas concreto, un motor de otro fabricante, etc. No tiene que pararse a diseñar cada pequeño detalle. Lo mismo ocurre con un relojero, que selecciona una serie de engranajes, tuercas y tornillos que utiliza para fabricar el reloj, sin tener la necesidad de tener que tornear y diseñar cada pieza.

La ingeniería del software trata de acercar este concepto de modularidad a la programación, y para ello utiliza el paradigma orientado a objetos. La programación orientada a objetos (POO) se basa en el uso de clases y objetos (también llamados instancias).

Para entender el concepto de clase recurrimos a un ejemplo de la vida real: hemos dicho que el diseñador de coches necesita poner unas ruedas, ya que todos los coches las llevan. El concepto general de rueda, antes de seleccionar una en concreto, es lo que se llama una clase. Decimos que todas las ruedas de coche pertenecen a una clase concreta de objeto, que es la clase rueda, es decir, una clase es una generalización que representa a todos los objetos de esa clase.

Ahora bien, una vez que el diseñador de coches elige una rueda concreta, está usando un objeto que pertenece a la clase rueda. Esa rueda concreta es lo que llamamos objeto en POO o también instancia de la clase rueda.

© RC Libros

En teoría podemos construir clases para cualquier tipo de cosa que pueda hacerse en Python, y reutilizarlas en cualquier programa. Por ejemplo, se podría construir una clase Matriz que represente a una matriz algebraica, y utilizar esa clase en cualquier programa en el que necesitemos operar con matrices. Una instancia de la clase Matriz sería una matriz concreta con los valores asignados a sus filas y columnas. Por ejemplo la matriz unitaria.

Sobre una clase, es posible realizar operaciones. Volviendo al ejemplo anterior, tiene sentido que sobre la clase Matriz se puedan aplicar las operaciones aritméticas como sumas o restas, obtener la matriz traspuesta, etc. Estas operaciones las definimos dentro de la clase y se denominan métodos. Un método define una operación que es posible realizar con los objetos de esa clase y son, de hecho, muy similares a las funciones.

Vamos a comenzar mostrando un ejemplo de una clase a la que vamos a llamar Coordenada, y que nos va a servir para representar coordenadas en un plano.

```
class Coordenada:
  def __init__(self, x, y):
    self.posx=x
    self.posy=y

  def ver(self):
    return self.posx, self.posy

  def mover(self, x, y):
    self.posx=x
    self.posy=y

punto1=Coordenada(10,10)
punto2=Coordenada(20,20)
print punto1.ver()
print punto2.ver()
```

Listado. A-10 clases1.py

Observemos que toda clase comienza con la palabra reservada *class* seguida del nombre de la clase. Por convención, los nombres de las clases comienzan con mayúsculas. Dentro de una clase se definen sus métodos, que como ya hemos comentado, son similares a las funciones. De hecho se declara de la misma forma mediante la palabra reservada *def*. A diferencia de las funciones, los métodos siempre llevan un primer parámetro llamado *self* que es una referencia al propio objeto instanciado.

En la clase de ejemplo del listado anterior hay tres métodos. El método mover, que cambia la posición de la coordenada asignándole nuevos valores; el método ver devuelve una tupla con las coordenadas del objeto, pero ¿de dónde salen esas coordenadas? Para entenderlo hay que fijarse en el primer método llamado __init__. Este método es especial y se llama constructor de la clase. Cada vez que creamos una instancia de la clase punto se invoca este constructor.

Si echamos un vistazo más de cerca al constructor vemos que, además del obligatorio parámetro *self*, recibe otros dos. En este caso x e y, que son las coordenadas del objeto de la clase Coordenada que queremos crear. No es obligatorio que el constructor tenga que recibir parámetros (aparte de *self*, claro). La misión del constructor es almacenar el código de inicialización de los objetos, que en este caso consiste en almacenar las coordenadas en dos variables de instancia. Una variable de instancia es una variable que es accesible por todos los métodos de la clase, y se crean, tal y como vemos en el ejemplo, como variables del objeto *self*. En el método *ver* observamos cómo es necesario poner el prefijo *self* delante de las variables para indicar que queremos referenciar dos variables de instancia.

Para crear objetos de la clase Coordenada solo hay que escribir:

```
punto1=Coordenada(10,10)
```

que crea un objeto de tipo Coordenada que es inicializado con la coordenada 10,10. Para invocar un método del objeto, simplemente lo hacemos utilizando el operador punto de la siguiente manera:

```
punto1.ver()
```

En el código de ejemplo vemos cómo se crean dos instancias de la clase Coordenada totalmente independientes, cada una con sus valores propios.

Una característica muy interesante de la POO es que podemos crear clases a partir de otras ya existentes. Este mecanismo se llama herencia, ya que permite crear

© RC Libros

clases nuevas heredando características de otra anterior. Supongamos que queremos crear una clase que represente un círculo. Podríamos crear una nueva clase cuyo constructor tomaría como parámetros la coordenada del círculo y su radio. Aprovechando que ya tenemos una clase Coordenada, podemos hacer que la clase Circulo herede sus características.

```
class Coordenada:
  def __init__(self, x, y):
    self.posx=x
    self.posy=y

  def ver(self):
    return self.posx, self.posy

  def mover(self, x, y):
    self.posx=x
    self.posy=y


class Circulo(Coordenada):
  def __init__(self, x, y, r):
    Coordenada.__init__(self,x,y)
    self.radio=r

  def ver(self):
    coord=Coordenada.ver(self)
    return coord[0], coord[1], self.radio


circulo=Circulo(10,10,5)
print circulo.ver()
circulo.mover(5,5)
print circulo.ver()
```

Listado. A-11 clases2.py

En el listado anterior hemos creado la clase Círculo que hereda de la clase Coordenadas. Esto se hace indicando la clase de la que hereda (también llamada super clase o clase padre) entre paréntesis. Hay que advertir que las clases padre e hija no tienen por qué estar en el mismo archivo (de hecho es mejor que cada clase se codifique en un archivo aparte).

El constructor de Círculo lo primero que hace es llamar al constructor de la clase padre pasándole las coordenadas, tras lo cual, almacena el radio del círculo en una variable de instancia. También se ha modificado el método ver, que hace uso del mismo método del padre y además añade la información del radio.

A pesar de que no se ha codificado el método *mover*, vemos cómo es posible invocarlo en la instancia de Círculo, ya que ha sido heredado de Coordenada. Es decir, que hay métodos como *ver* que necesitan cambios y otros como *mover* que pueden utilizarse tal cual. Se dice que los métodos que se modifican están sobreescritos en la clase hija.

Finalmente vamos a ver otra potente característica de la POO llamada polimorfismo. De una clase padre pueden descender varias clases hija, en este caso, siempre que necesitemos tratar con alguno de los hijos podremos utilizar variables del tipo de la clase padre, lo que nos puede facilitar el trabajo con grandes cantidades de objetos diferentes. Veámoslo mejor con un ejemplo.

```
class Coordenada:
  def __init__(self, x, y):
    self.posx=x
    self.posy=y

  def ver(self):
    return self.posx, self.posy

  def mover(self, x, y):
    self.posx=x
    self.posy=y



class Circulo(Coordenada):
  def __init__(self, x, y, r):
```


© RC Libros

```
        Coordenada.__init__(self,x,y)
        self.radio=r

    def ver(self):
        coord=Coordenada.ver(self)
        return coord[0], coord[1], self.radio


class Cuadrado(Coordenada):
    def __init__(self, x, y, l):
        Coordenada.__init__(self,x,y)
        self.lado=l

    def ver(self):
        coord=Coordenada.ver(self)
        return coord[0], coord[1], self.lado


figuras=[]
figuras.append(Circulo(10,10,5))
figuras.append(Cuadrado(20,20,10))
una_figura=Coordenada(0,0)
for figura in figuras:
    una_figura=figura
    una_figura.mover(5,5)
    print una_figura.ver()
```

Listado. A-12 clases3.py

En el ejemplo hemos creado una clase Cuadrado que hereda de Coordenada, pero a la que hemos añadido la longitud del lado. Si tenemos una serie de figuras almacenadas en una lista, por ejemplo, y necesitamos moverlas todas, podemos tratar a esas figuras como si se trataran de objetos Coordenada, sin importar si realmente es un círculo o un cuadrado. En el ejemplo se observa como la variable *una_figura*, que es de tipo Coordenada, va almacenando instancias de Circulo y Cuadrado e invocando a su método *mover*.

© RC Libros

Respecto a la POO se queda mucho en el tintero, pero lo visto hasta ahora nos vale como introducción.

Módulos

Los módulos son una forma de organizar datos, funciones y clases en Python, un módulo no es más que un archivo que contiene una serie de definiciones y que pueden ser usadas desde otros archivos de Python.

Para poder usar el contenido de un módulo debemos usar la palabra reservada import, como ya vimos anteriormente. Imaginemos, por ejemplo, que almacenamos el listado A-12 en un archivo llamado figuras.py, y que queremos hacer uso de las clases en él definidas. Si el fichero figuras.py se encuentra en el mismo directorio solo tendremos que usar las instrucciones siguientes para poder usarlo.

```
import figuras

circulo=figuras.Circulo(10,10,5)
```

Hay una variable especial llamada __name__ que nos indica el nombre del módulo que fue lanzado por el intérprete, y que toma el valor __main__ si el módulo que se está ejecutando actualmente es el que fue ejecutado por el usuario (el principal). Puede comprobarse así:

```
if __name__ == '__main__':
```

Esto hace posible que si tenemos una función o clase en un módulo, este pueda utilizarlo normalmente desde otro módulo, pero si lo ejecutamos directamente, podemos ejecutar código para, por ejemplo, probar el correcto funcionamiento de las funciones y clases que lo componen.

Paquetes

Los paquetes son otro método que nos ofrece Python para organizar el código, pero esta vez a nivel de sistemas de archivos. La idea es la siguiente: si tenemos una serie de módulos que hacen diferentes cosas, nos puede interesar organizarlos según su utilidad. Para ello, solo hay que separarlos en directorios. Imaginemos que hemos creado la siguiente estructura de directorios con los siguientes módulos para separarlos de otros archivos que tienen cometidos diferentes.

© RC Libros

figuras\
coordenada.py
circulo.py
cuadrado.py

Esto ayuda a organizar el código agrupando paquetes del mismo tipo. Podemos hacer uso de los paquetes de la forma siguiente.

```
from figuras import circulo
```

© RC Libros

ÍNDICE ALFABÉTICO

A

agentes, 5
Ajedrez, 142
Alan Turing, VII, 2, 6
Algoritmo A, 73*
Algoritmo de Clarke y Wright, 94
Algoritmo de Dijkstra, 88
algoritmo de los ahorros, 94
algoritmo minimax, 143
algoritmo negamax, 155
Algoritmos constructivos voraces, 86
algoritmos evolutivos, 126
Algoritmos genéticos, 126
aprendizaje, V, 3, 4, 193, 223
aprendizaje no supervisado, 223
Árboles, 31

B

back propagation network, 223
backtracking, 69
base de conocimiento, 172
base de hechos, 172
base de reglas, 172
búsqueda, 29
búsqueda con profundidad iterativa, 52
búsqueda con vuelta atrás, 69
búsqueda de coste uniforme, 56
búsqueda en amplitud, 34
búsqueda en profundidad, 45
búsqueda en profundidad limitada, 52
búsqueda informada, 29
Búsqueda local, 83
búsqueda no informada, 29
Búsqueda tabú, 117

C

capcha, 233
clase NP, 19
CLASIFICACIÓN PROBABILÍSTICA, 194
clasificador bayesiano ingenuo, 194
cláusulas, 21
complejidad espacial, 42
complejidad temporal, 42
completable, 69
completitud, 40
Conecta 4, 142
conjunto de conflicto, 174
Conjuntos difusos, 177
conjuntos nítidos, 178
Convergencia de la población, 127
criterio de aspiración, 119
Cromosoma, 127
Cruce, 127

D

Damas, 142
dominio de aplicación, 172

E

Eliminación, 127
encadenamiento hacia adelante, 173
encadenamiento hacia atrás, 173
entrenamiento, 211
escalada de la colina, 102

F

factor de convergencia, 127
factor de ramificación, 32
forma normal conjuntiva, 21
forma normal disyuntiva, 21
Función de adaptación, 127
función de evaluación, 14
Función Heurística, 65
Función semántica, 184

G

Genes, 127
Grafos, 31
greedy, 83

H

hechos, 172
heurística, 66
Hill climbing, 102

I

Individuo, 126
Inferencia difusa, 183

J

John McCarthy, 6
juegos, 141

L

lenguaje natural, 3
linealmente separable, 221
linealmente separables, 221
LISP, 6
lógica difusa, 177

M

máquina de Turing, VII
máximo global, 86
máximo local, 86
memoria asociativa, 233
memoria de trabajo, 172
metaheurísticas, 111
minería de datos, 203
modelo, 14
motor de inferencia, 172
Mutación, 127

N

naive bayes classifier, 194
Nitidificación, 191
no determinista, 138
nodos de probabilidad, 167
nodos frontera, 153

O

objetivo, 14
optimalidad, 40

P

perceptrón, 210
PLE, 23
Población, 126
Poda alfa-beta, 155
probabilidad, 194
probabilidad condicionada, 197
Problema, 9
programación lineal entera, 23
Python, 243

R

Ramón Llull, 5
razonamiento, V, 3, 4, 5, 171, 208
Red de Hopfield, 233
red de retropropagación, 223

© RC Libros

REDES NEURONALES ARTIFICIALES, 208
Redes neuronales multicapa, 220
regla de Laplace, 196
reglas, 172
representación del conocimiento, 4
robótica, 4
rutas de vehículos, 26

S

SAT, 19
satisfacción de restricciones, 85
satisfacibilidad booleana, 19
Savings Algorithm, 94
Selección, 127
Simulated annealing, 111
Sistemas Basados en Conocimiento, 172
sistemas basados en reglas, 172
SISTEMAS DIFUSOS, 176
SISTEMAS EXPERTOS, 172
sistemas multiagente, 5

T

t-conorma, 180
templado simulado, 111
teorema de Bayes, 199
teorema de las probabilidades totales, 198
t-norma, 182
Tres en Raya, 142
TSP, 15

U

Universo de discurso, 184

V

Valores lingüísticos, 184
variable lingüística, 184
viajante de comercio, 15
visión artificial, 4
voraces, 83
VRP, 26, 94

W

Walter Pitts, 6
Warren McCulloch, 6

© RC Libros