



ALGORITMOS DE OPTIMIZACIÓN

Raúl Reyero Díez

MÁSTER EN INTELIGENCIA ARTIFICIAL

Módulo II. Fundamentos matemáticos

viu

**Universidad
Internacional
de Valencia**

Este material es de uso exclusivo para los alumnos de la Universidad Internacional de Valencia. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la Universidad Internacional de Valencia, sin autorización expresa de la misma.

Edita

Universidad Internacional de Valencia

Máster en
Inteligencia Artificial

Algoritmos de optimización
Módulo II. Fundamentos matemáticos
6 ECTS

Raúl Reyero Díez

Leyendas



Enlace de interés



Ejemplo



Importante



abc Los términos resaltados a lo largo del contenido en color **naranja** se recogen en el apartado **GLOSARIO**.

Índice

CAPÍTULO 1. INTRODUCCIÓN A LOS ALGORITMOS	7
1.1. ¿Qué es un algoritmo?	7
1.2. Tipos de algoritmos.....	8
1.3. Desarrollo e implementación de algoritmos.....	9
1.4. Complejidad computacional	11
1.4.1. Análisis de la complejidad de un algoritmo	12
1.4.2. Órdenes de complejidad	14
1.4.3. Tipo de problemas según su complejidad	15
CAPÍTULO 2. ALGORITMOS DE ORDENACIÓN	17
2.1. Ordenación por inserción.....	18
2.2. Ordenación de burbuja	18
2.3. Ordenación por selección	20
2.4. Ordenación por mezcla	20
2.5. Ordenación por montículos	23
2.6. Ordenación rápida	24
2.7. Ordenación por residuos	25
CAPÍTULO 3. TÉCNICAS DE DISEÑO DE ALGORITMOS	26
3.1. Algoritmos voraces	26
3.1.1. Características de los algoritmos voraces	26
3.1.2. Uso de grafos para la modelización de problemas	29
3.2. Vuelta atrás	30
3.2.1. Diseño y resolución según la técnica de vuelta atrás	30
3.3. Divide y vencerás	33
3.4. Programación dinámica	35
CAPÍTULO 4. PROBLEMAS TIPO	37
4.1. Problema del agente viajero	37
4.2. Problema de la mochila	39
4.3. Problema de programación lineal entera.....	39
4.4. Problema de enrutamiento	41
4.5. Otros problemas	42

CAPÍTULO 5. ALGORITMOS DE BÚSQUEDA	44
5.1. Búsqueda en amplitud	45
5.2. Búsqueda en profundidad	45
5.3. Ramificación y poda	47
CAPÍTULO 6. DESCENSO DEL GRADIENTE	48
CAPÍTULO 7. MÉTODOS HEURÍSTICOS Y METAHEURÍSTICOS.....	52
7.1. Búsqueda tabú	54
7.2. Recocido simulado.....	56
7.3. Optimización por enjambre de partículas.....	58
7.4. Procedimientos de búsqueda voraz aleatorios y adaptativos	60
CAPÍTULO 8. ALGORITMOS EVOLUTIVOS Y GENÉTICOS	62
8.1. Algoritmos evolutivos	63
8.2. Algoritmos genéticos.....	63
8.3. Optimización por colonia de hormigas	65
GLOSARIO.....	67
ENLACES DE INTERÉS	69
BIBLIOGRAFÍA.....	71



Capítulo 1

Introducción a los algoritmos

En este primer capítulo introduciremos el concepto de **algoritmia**, que permitirá el desarrollo de herramientas para resolver problemas a través de la computación. Estudiaremos el concepto de complejidad computacional y describiremos algunas técnicas generales en el diseño de algoritmos.

1.1. ¿Qué es un algoritmo?

Cada día utilizamos **algoritmos** casi sin darnos cuenta, por ejemplo, cuando realizamos operaciones matemáticas sencillas (sumas, restas, multiplicaciones y divisiones). En el colegio nos enseñaron a aplicar algoritmos a un dato de entrada para obtener otro dato como resultado. Y no solo lo hacemos para resolver cuestiones relacionadas con las matemáticas (al menos directamente). El ámbito de aplicaciones de algoritmos abarca casi todos los aspectos de la vida en los que es necesario llegar a una situación partiendo de otra situación de partida: poner la mesa para comer, preparar la comida, elegir una película, buscar el asiento asignado, conducir... La lista es interminable. Con mayor o menor grado de optimización, somos capaces de aplicar algoritmos en la vida diaria. Sin embargo, nuestro interés se centrará en los algoritmos relativos a cálculos para resolver problemas matemáticos y más concretamente en los que conduzcan a soluciones óptimas.

La palabra *algoritmo* proviene del nombre **al-Juarísmi**, que fue un matemático persa que vivió entre los años 780 y 850 dC.



Un algoritmo es un conjunto de reglas ordenadas y finitas para realizar una actividad mediante la sucesión de pasos claramente establecidos.

1.2. Tipos de algoritmos

Es necesario comentar esta definición en la parte que se refiere a la expresión “pasos claramente establecidos”, lo cual nos permite introducir una primera clasificación.

Podemos clasificar los algoritmos en **dos tipos principales**:

- **Algoritmos deterministas**, que producen siempre la misma solución para los mismos valores de entrada.
- **Algoritmos probabilistas o no deterministas**, que pueden generar soluciones distintas para los mismos datos de entrada producidas por la incorporación de la aleatoriedad en el proceso.

Otra clasificación general que podemos hacer sobre los algoritmos es atendiendo al **tipo de solución** que proporcionan:

- **Algoritmos exactos**, que proporcionan la solución óptima
- **Algoritmos aproximados**, que proporcionan una buena solución con un grado de aproximación cercano al óptimo
- **Algoritmos heurísticos**, para los que no podemos conocer el grado de aproximación a la solución óptima.

El primer algoritmo conocido, o al menos el más famoso de los antiguos que se citan en bibliografía, se atribuye a Euclides y permite calcular el **máximo común divisor** (MCD) de dos números cualesquiera.

Se atribuye a Euclides porque aparece en *Los elementos*, volumen VI, capítulo 2, por primera vez.

Euclides lo describe desde el punto de vista geométrico: encontrar el segmento de mayor tamaño commensurable con otros dos segmentos dados. El algoritmo podría expresarse de la siguiente manera:

Dados dos números, $N > M$:

1. Restamos M a N tantas veces como sea posible. Si no hay resto, entonces M es el MCD.
2. Se repite el paso 1, pero ahora con M y el resto del paso 1.

El proceso siempre termina en algún momento del punto 1. El **pseudocódigo** asociado a este algoritmo es el siguiente:

```
función mcd(N,M)
    mientras M > 0
        si N > M
            M = N - M
        si no
            N = N - M
    devolver N
```

</>

Otros algoritmos utilizados desde tiempos antiguos son (Peña, 2006):

- El **método Horner** para calcular el valor de un polinomio se conoce en la cultura matemática china desde el siglo I aC.
- El **método de Herón**, del siglo I dC, para obtener la raíz cuadrada de un número.
- La **criba de Eratóstenes**, del siglo II aC, para encontrar números primos.



Enlace de interés

En la web del National Institute of Standards and Technology podemos consultar el *Dictionary of Algorithms and Data Structures*, una lista de todos los algoritmos documentados con una breve descripción:

<https://xlinux.nist.gov/dads/>

1.3. Desarrollo e implementación de algoritmos

El desarrollo de un algoritmo supone varias fases (Joyanes, 2003):

Durante la primera fase debemos realizar un modelo del problema a resolver. **Modelar** significa determinar los supuestos del problema y los datos de inicio. En muchos casos no podremos realizar un modelo que se ajuste exactamente a la realidad, pero debemos determinar si estas diferencias nos pueden llevar a soluciones deseadas.

La segunda fase es el **diseño** del algoritmo utilizando algunas de las diferentes técnicas orientadas a la resolución de problemas. Veremos en los siguientes capítulos algunas de las técnicas más importantes y utilizadas.

Durante la tercera etapa realizaremos el **análisis** del algoritmo. En esta etapa debemos estudiar la cantidad de operaciones que realizará el algoritmo para resolver el problema. El estudio práctico conlleva el análisis del peor caso (cota superior), sobre todo si el algoritmo debe resolver diferentes casos de un mismo problema.

Puesto que el peor caso puede ser excepcional, también se contempla un análisis promedio, pero a veces este cálculo es más difícil de obtener y en algunos casos no es posible. La información obtenida del análisis puede dar lugar a una revisión del diseño porque los resultados obtenidos no sean satisfactorios. Por otro lado, si encontramos un algoritmo para el que podemos demostrar que no existe otro que lo mejore, diremos que el **algoritmo es óptimo**.

En algunos casos nos encontraremos con problemas para los que no existe un algoritmo práctico conocido. Hay que tener en cuenta que no todos los algoritmos conocidos que resuelven el problema resultan prácticos, porque tardarán un tiempo excesivo. En este caso, debemos recurrir a la búsqueda de reglas que, aunque no nos den la solución óptima, sí nos den una buena aproximación. Estos son los **algoritmos heurísticos**, en los que, como vimos anteriormente, no podemos siquiera determinar si son óptimos globales o locales.

Ya hemos adelantado que uno de los propósitos de los algoritmos es encontrar soluciones para problemas que puedan resolverse computacionalmente. En este punto podríamos preguntarnos **si la solución que proporciona el algoritmo es única**. En caso afirmativo, la siguiente cuestión a plantearse es la siguiente: dadas todas las soluciones posibles, ¿podemos encontrar la mejor solución? Quizá para este problema no tenga sentido esta pregunta. En este caso habremos terminado la tarea de resolver el problema.

Sin embargo, existen muchos problemas en los que sí tiene sentido encontrar la **mejor solución**. Es evidente que debemos disponer de algún criterio que nos permita comparar las posibles soluciones para establecer o llegar a la mejor. Si podemos establecer una función que permita esta comparación entre soluciones, tendremos la **función objetivo**, para la que debemos encontrar la solución óptima entre todas las posibles. Por tanto, en la práctica, encontrar la solución óptima de un problema consiste en hallar el máximo o el mínimo de una función dentro de un dominio de soluciones. Esta área de las matemáticas se llama **optimización**.

Una dificultad añadida se produce cuando el problema trata de optimizar más de un valor. En este caso, hablamos de **optimización multiobjetivo**. Aunque se dan en muchos problemas reales, un problema típico de optimización multiobjetivo es el de encontrar la mejor solución para un recorrido que minimice el tiempo empleado y el combustible gastado.

En general, estas situaciones presentan un **conflicto entre los objetivos** perseguidos. En estos casos, a diferencia de la optimización sobre un solo objetivo, no existe una solución única. Solo podemos alcanzar un conjunto de soluciones en que todos los objetos son satisfechos de una manera aceptable. Una posibilidad para estos casos es el llamado **óptimo de Pareto**, en el que se ofrece un subconjunto de soluciones para las que el resto de las soluciones no pueden mejorar ningún objetivo sin perjudicar a los otros.

1.4. Complejidad computacional

Debido a la progresión en la mejora de la velocidad de los cálculos, podemos pensar que un mal **diseño** puede compensarse con una alta **velocidad de proceso**. Pero esto puede no ser cierto, ya que hay casos en los que un mal diseño con una alta velocidad de proceso puede dar peores resultados que un buen diseño con velocidades considerablemente más bajas.

En el Capítulo 2, sobre algoritmos de ordenación, veremos en detalle algunos algoritmos de ordenación que tienen cierto protagonismo en la resolución de problemas, ya que, aunque *a priori* el problema no sea específicamente de ordenación, la solución sí pasa por realizarla.

En la Figura 1 puede observarse el tiempo que tardan dos algoritmos en ordenar una lista de N elementos. Aunque para valores pequeños observamos que la diferencia es pequeña, a medida que el número de elementos crece, la diferencia de tiempos se dispara:

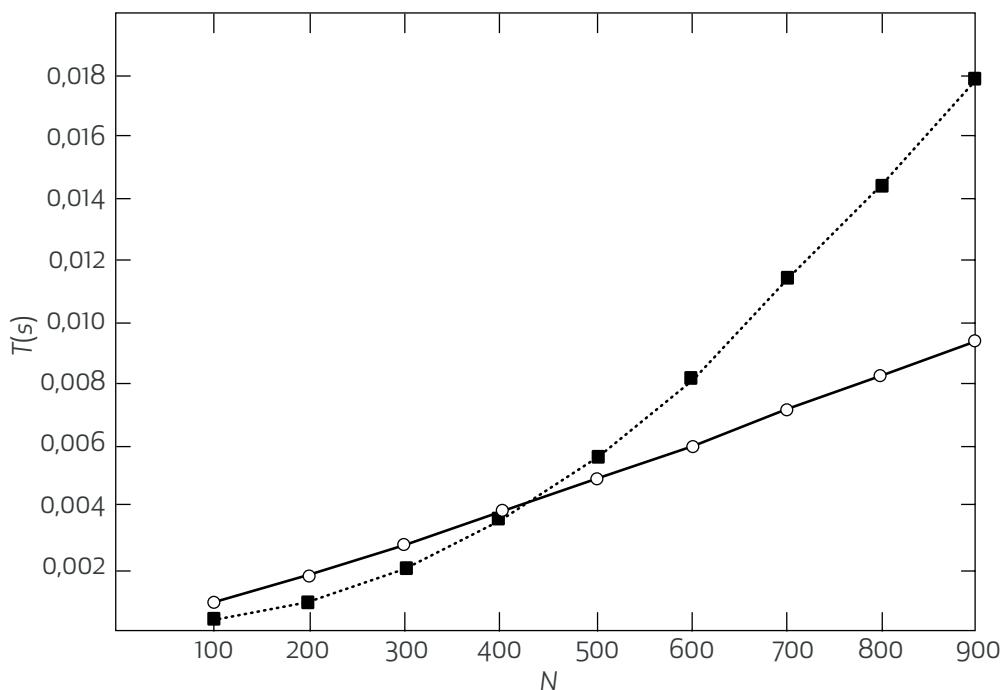


Figura 1. Tiempo de ejecución de dos algoritmos según la cantidad de elementos a ordenar: ordenación por inserción y ordenación rápida o *quick sort*.

Por tanto, es necesario analizar el **rendimiento** del algoritmo. En la eficiencia del rendimiento se suelen tener en cuenta dos elementos: el **tiempo** y el **espacio**. El tiempo se refiere al tiempo que tarda en ejecutarse y el espacio a la memoria que necesita para ser ejecutado por uno o varios procesadores. Estos dos valores representan el coste de encontrar la solución al problema y además los usaremos para comparar la eficiencia de diferentes algoritmos.

En general, centraremos los análisis en el tiempo, que suele ser el elemento más limitante.



El **tiempo de ejecución** depende de los siguientes factores:

- El tamaño de entrada.
- La calidad del código del programa.
- El procesador que ejecute la programación.
- La complejidad propia del algoritmo.

Por **tamaño de entrada**, entendemos el número de componentes sobre los que se va a ejecutar el algoritmo (por ejemplo, el tamaño del vector, la lista de valores numéricos que se ordenan o el orden de las matrices que se multiplican).

1.4.1. Análisis de la complejidad de un algoritmo

En el estudio o análisis del tiempo de ejecución podemos considerar **dos posibilidades** (Brassard y Bratley, 1997):

- A. Realizar una **medida teórica** (antes de la ejecución) para que acote por arriba o por abajo el tiempo de ejecución del algoritmo para unos valores de entrada dados.
- B. Realizar una **medida real** (después de la ejecución) para unos valores dados y un procesador concreto.

El punto B ya se ve poco adecuado debido a la dependencia de un procesador concreto. Esto impide la posibilidad de realizar comparaciones de eficiencia entre algoritmos si la ejecución se efectúa en procesadores diferentes o incluso en el mismo procesador, pero en circunstancias diferentes. Por tanto, la unidad para medir la eficiencia no puede ser el segundo ni ninguna otra unidad temporal.



Usamos una medida teórica, que consiste en **contar el número de instrucciones ejecutadas** para conseguir la independencia del procesador. Esta medida es una función $T(n)$ que depende del tamaño de entrada.

Así pues, decimos que un algoritmo se ejecuta en orden de $T(n)$ si existe una constante C y una implementación / del algoritmo para la que la ejecución tarda menos de $C \cdot T(n)$ para cualquier tamaño n de los datos de entrada.

Partiendo de esta consideración, debemos tener en cuenta dos elementos:

- Las **constantes multiplicativas**. Aunque, por ejemplo, un orden cuadrático $T(n) = n^2$ es mejor y más deseable que un orden cúbico, $T(n) = n^3$, puede ocurrir que para valores pequeños deseemos un algoritmo de orden cúbico frente a uno cuadrático si el número de instrucciones son $5n^3$ y $10^7 n^2$. En este caso particular, los datos de entrada no son de un tamaño superior a 142.000.

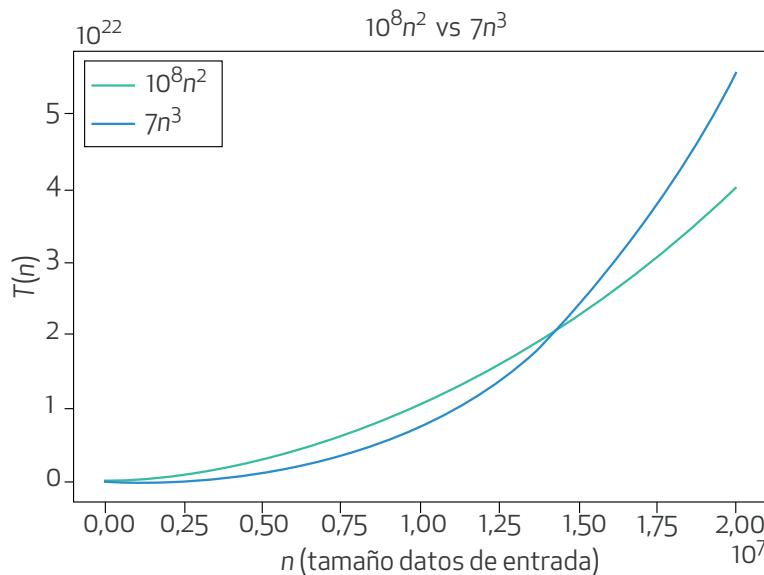


Figura 2. Comparación de órdenes cuadrático y cúbico con diferentes constantes multiplicativas.

- El **comportamiento de un algoritmo**, que puede variar para diferentes entradas de datos, aunque estas sean del mismo tamaño. Un ejemplo sencillo de entender de algoritmo de ordenación consiste en proporcionar una lista ya ordenada o casi ordenada frente a una lista completamente desordenada.

Así pues, estudiamos un algoritmo desde el punto de vista del comportamiento atendiendo a tres casos: el caso más desfavorable, el caso más favorable y el caso promedio. El caso promedio se calcula como la esperanza matemática según la variable aleatoria que determina todas las posibles opciones para los valores de entrada teniendo en cuenta las probabilidades de que estas opciones ocurran. Este cálculo suele ser más complejo que el de los casos más y menos favorables. Una consideración importante y que no debe llevar a confusión es tener en cuenta el caso favorable como el caso para un tamaño de entrada para $n = 1$, por ejemplo. Siempre debemos considerar los tres casos (más favorable, menos favorable y medio) para un tamaño de datos dado.



Mediremos, por tanto, el tiempo de ejecución de un algoritmo en **operaciones elementales** (OE). Estas son operaciones que realiza el procesador y que podemos asegurar que siempre, en cualquier circunstancia, están acotadas por una constante. Se consideran operaciones elementales las operaciones aritméticas básicas (suma, resta, multiplicación y división), las asignaciones de valores a variables, los saltos o llamadas a funciones externas, los retornos de estas funciones, las comparaciones y los accesos a estructuras de datos a través de índices. Para cada una de estas operaciones diremos que estamos consumiendo una operación elemental (1 OE).

Siendo estrictos y desde el punto de vista computacional, se podría considerar que estas operaciones no tienen la misma complejidad. Por ejemplo, realizar una suma siempre es menos complejo que realizar una división, pero asumiremos que todas ellas son operaciones elementales desde el punto de vista del cálculo de complejidad de un algoritmo.

1.4.2. Órdenes de complejidad

Decimos que un algoritmo tiene un **orden de complejidad** $O(f)$ si su tiempo de ejecución (medido en operaciones elementales) está acotado por f multiplicado por una constante para todos los n (tamaño de la muestra) a partir de un n_0 .

Según esta clasificación, los órdenes de complejidad dependiendo de n (tamaño de los datos de entrada) más habituales en los que podemos clasificar los algoritmos son los siguientes:

- Orden constante, $O(1)$.
- Orden logarítmico, $O(\log n)$.
- Orden lineal, $O(n)$.
- Orden casi lineal, $n \cdot \log n$.
- Orden cuadrático, $O(n^2)$.
- Orden polinomial ($a > 2$), $O(n^a)$.
- Orden exponencial ($a > 1$), $O(a^n)$.
- Orden factorial, $O(n!)$.

La lista está ordenada de **menor a mayor complejidad**. El caso más sencillo, complejidad constante, nos indica que el algoritmo siempre se ejecutará con un número de operaciones que no depende de n , esto es, del tamaño de los datos de entrada. A partir del segundo orden de la lista, todos dependen de los datos de entrada, pero a medida que avanzamos en la lista la complejidad aumenta. En el peor caso encontramos el orden factorial que para valores no muy grandes de n ya se hace excesivamente complejo y por tanto poco práctico.

En la siguiente tabla se comparan los tiempos para algunos órdenes de complejidad. Observamos como los tiempos para los órdenes logarítmico ($\log n$) y polinomial (n^2) pueden ser razonables aun para tamaños de n grandes. Sin embargo, los algoritmos a partir de órdenes exponenciales (2^n) empiezan a ser de escasa utilidad excepto para valores pequeños de n .

Tabla 1

Comparación de los órdenes de complejidad según el tamaño del problema

n	$\log n$	n^2	$2n$	$n!$
2	1	4	4	2
8	3	64	256	40.320
32	5	1.024	4.3×10^9	2.6×10^{35}
100	6	104	1.2×10^{27}	9.3×10^{177}

La siguiente imagen nos permite comparar de manera visual estas magnitudes:

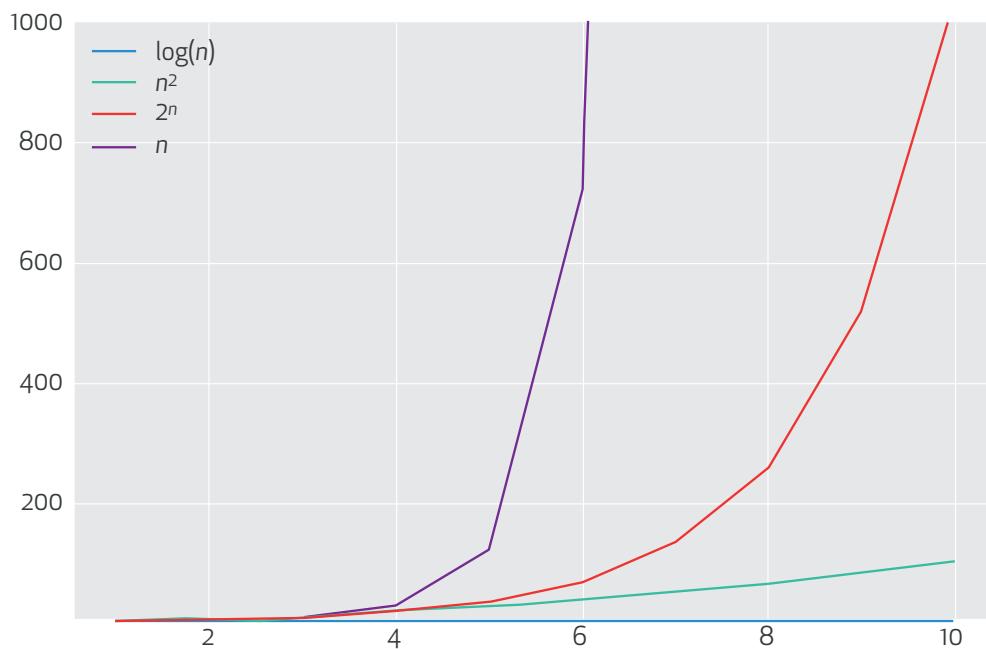


Figura 3. Gráfico que muestra la cantidad de operaciones asociadas a las diferentes complejidades.

Determinar el orden de complejidad es fundamental porque, aunque conoczamos algún algoritmo para resolver algunos problemas tipo, el **tamaño de los datos de entrada** puede hacer inútil el algoritmo si este es de un orden exponencial o superior. En estos casos, debemos considerar algunas técnicas que nos puedan proporcionar buenas soluciones, aunque no nos permitan alcanzar el óptimo global.

1.4.3. Tipo de problemas según su complejidad

Hasta ahora, hemos estudiado la complejidad computacional desde el punto de vista de los algoritmos. A continuación, introduciremos aspectos de la complejidad desde el punto de vista de los **problemas** en sí mismos. Esto nos ayudará a tener una visión más clara a la hora de establecer una estrategia abordable para resolverlos.

La clasificación de los problemas atendiendo a su complejidad se fundamenta en la existencia o no del mejor algoritmo que lo resuelva. Así pues, si para un problema determinado conocemos la existencia de un algoritmo que lo resuelva en un tiempo polinomial, diremos que el **problema es de clase P**. Estos son los más sencillos de la escala que vamos a considerar.

En el siguiente nivel se encuentran los **problemas de la clase NP**, para los que no se conoce un algoritmo que los resuelva en tiempo polinómico, pero para los que sí podemos verificar una solución en tiempo polinómico. El hecho de que no dispongamos de un algoritmo para resolverlo no quiere decir que no exista. Podría ser descubierto en el futuro y convertirse en un problema de la clase P. Es evidente que los problemas P están dentro de NP, pero la cuestión de si todos los problemas NP están en P no es tan evidente y es una de las cuestiones fundamentales de las matemáticas y la computación a día de hoy. Tanto es así que es uno de los siete problemas fundamentales de las matemáticas por los que el Clay Mathematics Institute ofrece un millón de dólares de recompensa para quien lo resuelva.



Enlace de interés

El Clay Mathematics Institute es una fundación privada sin ánimo de lucro dedicada a aumentar y difundir el conocimiento matemático. La página de los problemas del milenio es la siguiente:

<http://www.claymath.org/millennium-problems>

La mayoría de los matemáticos apuntan que **P es diferente de NP**, pero no está probado. Lo contrario, que $P = NP$, supondría la seguridad de encontrar algoritmos polinomiales para resolver problemas que a día de hoy consideramos difíciles. Existen muchos problemas considerados NP en la actualidad. La mayoría son muy conocidos, como el problema del viajero, la coloración mínima de un mapa, el problema de la mochila, un sudoku o el problema de las n reinas, entre otros muchos.

Un subconjunto de la clase NP es la clase **NP-completo**. En este caso, decimos que un problema pertenece a la clase NP-completo (o es NP-completo) si, además de ser de la clase NP, se verifica que todos los problemas de la clase NP pueden ser reducidos a dicho problema.

Por deducción, decimos que un problema se puede convertir en otro con transformaciones de orden polinómico. El primer problema que se demostró que era NP-completo es el de satisfacibilidad booleana (o SAT), que veremos en el Capítulo 2, sobre problemas tipo. La importancia de encontrar un algoritmo polinomial para resolver cualquier problema NP-completo supondría probar que $P = NP$, con las consecuencias que hemos visto anteriormente.

Finalmente existe una clasificación de problemas llamados **NP-difíciles** (*NP-hard* en inglés). Diremos que un problema es NP-difícil si cualquier problema de la clase NP se puede transformar polinomialmente en dicho problema. Esta notación es algo confusa, pues, como se ve en la Figura 4, hay problemas NP-difíciles que no son NP. Sin embargo, ya es histórica y es vigente a día de hoy.

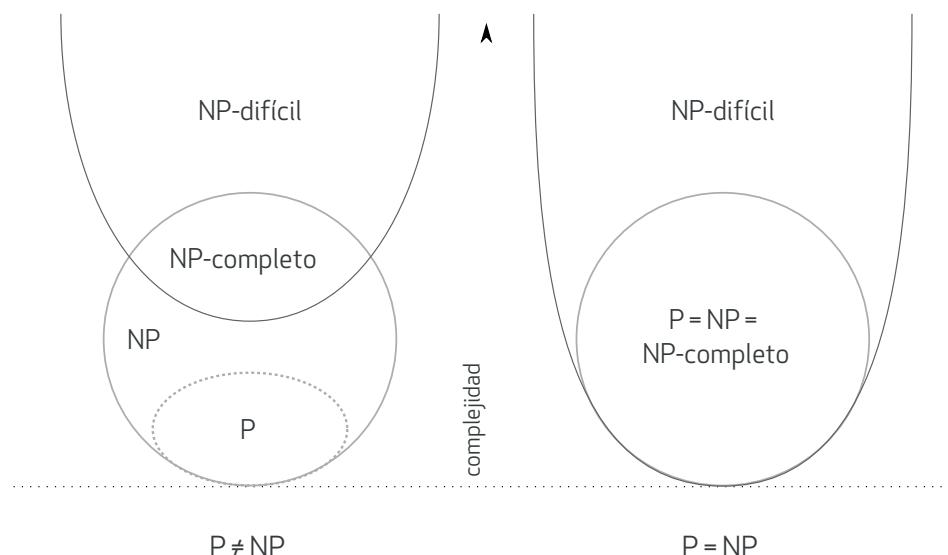


Figura 4. Representación de la estructura de las clases de problemas según si las clases P y NP son diferentes o no. Por Behnam Esfahbod bajo licencia CC BY-SA 3.0, 2.5, 2.0, 1.0. Recuperado de https://commons.wikimedia.org/wiki/File:P_np_np-complete_np-hard.svg



Capítulo 2

Algoritmos de ordenación

Los **problemas de ordenación** consisten en ordenar un vector o lista de valores numéricos partiendo de una relación de orden sobre los elementos. Existen algoritmos con diferentes funciones de eficiencia.



Entre los más utilizados y conocidos con su orden de complejidad tenemos los siguientes (Guerequeta y Vallello, 1998):

- Complejidad $O(n^2)$: por inserción, por selección y burbuja.
- Complejidad $n \log(n)$: mezcla, montículos y ordenación rápida.

En algunos de estos algoritmos nos vemos en la necesidad de utilizar algunas funciones básicas que se repiten. Estas funciones son: obtener el máximo o el mínimo valor de una lista que tiene una complejidad lineal ($O(n)$) e intercambiar dos valores de una lista que tiene una complejidad constante (concretamente, 7 operaciones elementales).

Revisaremos estos algoritmos y sus ventajas e inconvenientes para decidir en cada caso concreto cuál puede ser el más conveniente.

2.1. Ordenación por inserción

La ordenación por inserción, *insertion sort* en inglés, es el algoritmo más natural e intuitivo para el ser humano, pues es el que utilizamos habitualmente para realizar **ordenaciones de objetos**. Se basa en realizar ordenaciones parciales de la lista original. En cada iteración consideramos el elemento i -ésimo para situarlo ordenadamente en la lista ya ordenada de los anteriores elementos ya considerados.

En el mejor de los casos, tiene una **complejidad lineal**, $O(n)$, pero **cuadrática**, $O(n^2)$, para los casos desfavorables o incluso para el caso promedio. Por este motivo, es menos eficiente que otros algoritmos menos complejos que veremos en las siguientes secciones. Sin embargo, se sigue considerando por su sencillez. Es posible implementarlo en escasas líneas de código para muchos lenguajes de programación.

```
1 def insertionSort(alist):
2     for index in range(1,len(alist)):
3
4         currentValue = alist[index]
5         position = index
6
7         while position>0 and alist[position-1]>currentValue:
8             alist[position]=alist[position-1]
9             position = position-1
10
11         alist[position]=currentValue
12
13 alist = [54,26,93,17,77,31,44,55,20]
14 insertionSort(alist)
15 print(alist)
16
```

Figura 5. Código en Python para ilustrar la sencillez de la implementación del algoritmo de ordenación por inserción.

Este algoritmo es aconsejable, por tanto, si queremos realizar ordenaciones de listas pequeñas y sobre todo para listas en las que sabemos que ya hay un cierto orden.

2.2. Ordenación de burbuja

La **ordenación de burbuja** (*bubble sort* en inglés) también es un algoritmo muy intuitivo. Se llama así porque en cada iteración se persigue encontrar el valor máximo (o mínimo según queramos orden ascendente o descendente) y situarlo en la posición correspondiente. En cada iteración, por tanto, reducimos el tamaño de la lista en un elemento. Para hacerlo, el algoritmo recorre las listas parciales comparando dos elementos adyacentes e intercambiándolos si corresponde.

En la Figura 6 se muestra que, en la primera iteración, el algoritmo localiza el mayor valor de la lista situándolo en la última posición y que ha ido comparando los valores adyacentes de dos en dos.

54	26	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
26	54	17	93	77	31	44	55	20
26	54	17	77	93	31	44	55	20
26	54	17	77	31	93	44	55	20
26	54	17	77	31	44	93	55	20
26	54	17	77	31	44	55	93	20
26	54	17	77	31	44	55	20	93

Figura 6. Visualización de la primera iteración del algoritmo de ordenación de burbuja.

En este caso, la implementación también es **sencilla**. Sin embargo, a diferencia del algoritmo por inserción, para este algoritmo siempre tendremos una complejidad $O(n^2)$, independientemente de si estamos en el caso más favorable o no. Siempre realizaremos la misma cantidad de operaciones. Es posible mejorarlo introduciendo un control en el que, si en una iteración no se ha realizado ninguna modificación en la lista, podemos considerar que esta ya está ordenada. Con esta mejora podemos acercarnos a una complejidad $O(n)$ para los casos más favorables.

Aunque no es el mejor algoritmo en cuanto a rendimiento, sigue estando vigente por su sencillez en la implementación.

La implementación en Python puede realizarse con el siguiente **código**:

```
</>  
def bubbleSort(alist):  
    for passnum in range(len(alist)-1,0,-1):  
        for i in range(passnum):  
            if alist[i]>alist[i+1]:  
                temp = alist[i]  
                alist[i] = alist[i+1]  
                alist[i+1] = temp  
  
    alist = [54,26,93,17,77,31,44,55,20]  
    bubbleSort(alist)  
    print(alist)
```

2.3. Ordenación por selección

El funcionamiento del algoritmo de **ordenación por selección** es sencillo e intuitivo. Si hay que ordenar de menor a mayor el algoritmo, busca el menor número de todos dentro de la lista. Una vez localizado, lo pone en primer lugar intercambiándolo con el que allí esté. Procede a continuación con la misma operación, pero ahora considerando la sublistas a partir del segundo valor y así sucesivamente hasta completar todos los elementos de la lista.

La implementación en Python puede realizarse con el siguiente **código**:

```
</>  
def selectionSort(alist):  
    for fillslot in range(len(alist)-1,0,-1):  
        positionOfMax=0  
        for location in range(1,fillslot+1):  
            if alist[location]>alist[positionOfMax]:  
                positionOfMax = location  
  
            temp = alist[fillslot]  
            alist[fillslot] = alist[positionOfMax]  
            alist[positionOfMax] = temp  
  
    alist = [54,26,93,17,77,31,44,55,20]  
    selectionSort(alist)  
    print(alist)
```

El algoritmo tiene una complejidad cuadrática $O(n) = n^2$ en todos los casos, pues siempre va a realizar las mismas operaciones.



Enlace de interés

En el siguiente enlace puede observarse la resolución visual de un ejemplo de ordenación:

<http://interactivepython.org/runestone/static/NPHSCSA/SortSearch/TheSelectionSort.html>

2.4. Ordenación por mezcla

El algoritmo de **ordenación por mezcla** (*merge sort* en inglés) utiliza la técnica de resolución conocida como "Divide y vencerás". Veremos esta técnica en el Capítulo 3, sobre técnicas de diseño de algoritmos, ya que merece un tratamiento más detallado debido a su versatilidad para aplicarlo a diferentes tipos de problemas.

La estrategia del algoritmo consiste en dividir la lista en dos mitades, realizar la ordenación mediante llamadas recursivas y combinar las dos listas ya ordenadas.

En la Figura 7 vemos cómo se van dividiendo las listas (señaladas en gris y blanco) en cada iteración:

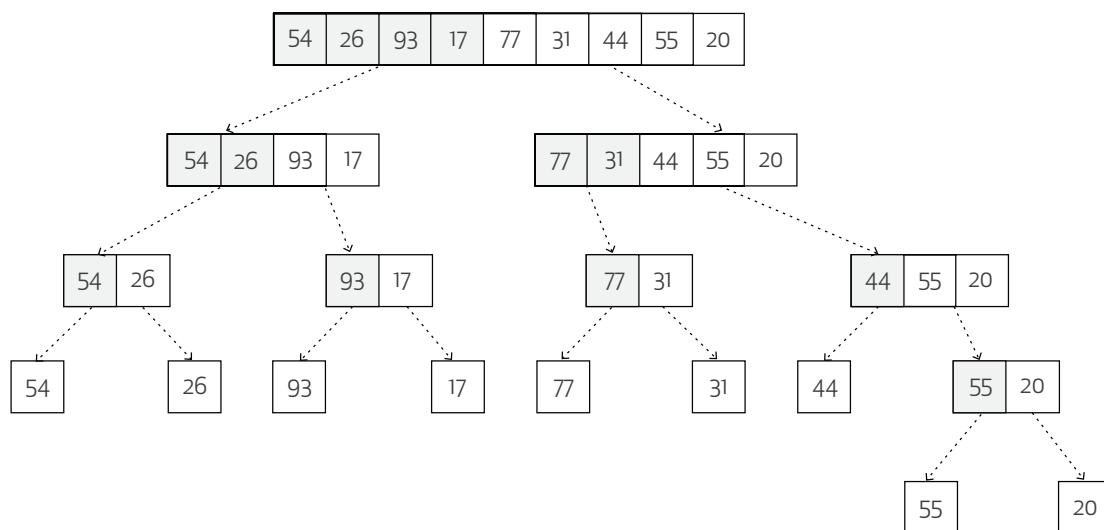


Figura 7. División de la lista original en la ordenación por mezcla.

El segundo paso será combinarlas para formar la lista ordenada definitiva:

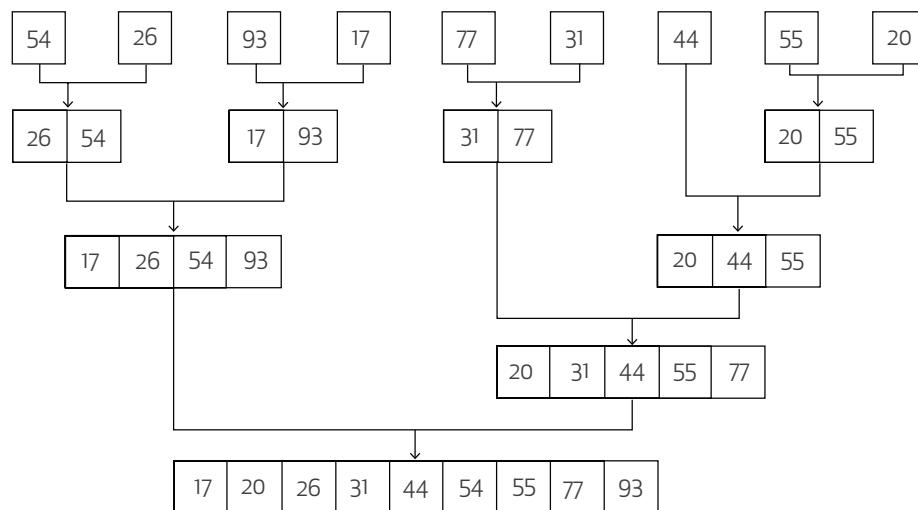


Figura 8. Combinación de las listas parciales generadas en el algoritmo ordenación por mezcla.

La implementación en Python puede realizarse con el siguiente **código**:

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
```



```
mergeSort(lefthalf)
mergeSort(righthalf)

i=0
j=0
k=0
while i < len(lefthalf) and j < len(righthalf):
    if lefthalf[i] < righthalf[j]:
        alist[k]=lefthalf[i]
        i=i+1
    else:
        alist[k]=righthalf[j]
        j=j+1
    k=k+1

while i < len(lefthalf):
    alist[k]=lefthalf[i]
    i=i+1
    k=k+1

while j < len(righthalf):
    alist[k]=righthalf[j]
    j=j+1
    k=k+1
print("Merging ",alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

En cuanto a la **complejidad**, el algoritmo se clasifica dentro de $O(n) = n \cdot \log(n)$ en todos los casos; mejor, peor y promedio. Este algoritmo, por tanto, mejora en complejidad en relación con los anteriores, ya que es de orden inferior al cuadrático.

El estudio de la complejidad se realiza a través de la resolución de **ecuaciones en recurrencia** como ocurre en general en la técnica de "Divide y vencerás".

El algoritmo por mezcla ha sido muy utilizado, dado que se adapta muy bien al **acceso secuencial de los datos**.

Actualmente, con las nuevas tecnologías, en general ya no es necesario almacenar los datos en sistemas de acceso secuencial, por lo que esta ventaja queda ensombrecida y su uso ha disminuido.

2.5. Ordenación por montículos

El algoritmo de **ordenación por montículos** (*heap sort* en inglés) aprovecha las propiedades de la estructura de datos de un montículo (árbol binario completo).

Sin pérdida de generalidad, podemos suponer que la ordenación es de menor a mayor. Para ello, construimos el árbol de tal manera que cada nodo tenga menor valor que el de sus hijos.

La **implementación** del algoritmo depende de la implementación que realicemos para almacenar el árbol binario. Una elección sencilla es utilizar un vector. De esta manera el padre del i -ésimo elemento está en la posición $i/2$ y sus hijos (si los tiene) en las posiciones $2i$ y $2i + 1$.

Una vez elegida la implementación del árbol, en cada iteración se sitúa cada valor en la posición que le corresponde, recolocando los elementos de la rama donde ha sido situado.

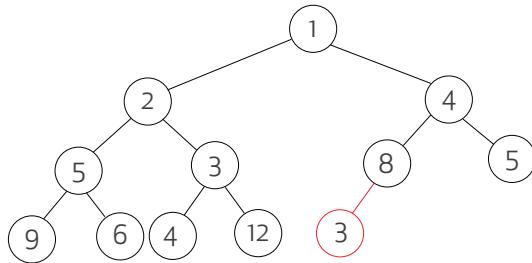


Figura 9. Estructura inicial de un árbol para el algoritmo de ordenación por montículos.

En el ejemplo de la Figura 9, debemos insertar el valor 3. El primer espacio libre para que el árbol siga siendo completo es el hijo izquierdo del nodo 8. Sin embargo, esta inserción contradice la definición de montículo (los padres deben tener mayor valor que los hijos). Para resolverlo, el algoritmo intercambia estos dos valores (el 3 por el 8). Aun así, vuelve a producirse una contradicción, este caso con el 4. El algoritmo procede de la misma manera (método recursivo) para intercambiar el 3 por el 4. El montículo final después de la inserción del 3 queda tal como se ve en la Figura 10:

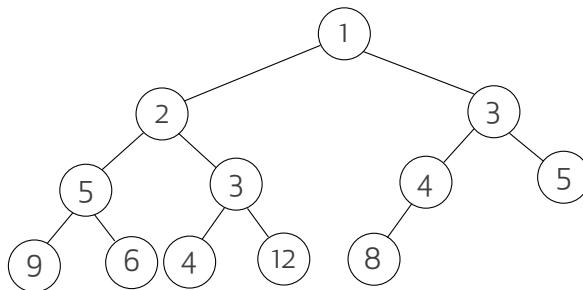


Figura 10. Estructura final, tras la reordenación para equilibrar el árbol en el algoritmo de ordenación por montículos.

La complejidad de este algoritmo es **casi lineal**, $O(n) = n \cdot \log(n)$ en el peor caso, si bien es posible que sea menor para casos favorables en los que no haya que profundizar muchos niveles para recolocar los elementos. Si el elemento es mayor que todos, en cada iteración no será necesario realizar

recolocaciones. Y, al contrario, si el elemento a considerar es el menor, será necesario realizar la reco-locación en todos los niveles para cada inserción.

Este algoritmo es una buena opción para implementar en la práctica. Se comporta bien para datos muy desordenados y muy bien cuando los datos tienen cierto orden que nos beneficie. Además, no necesita almacenamiento temporal extra, lo cual sí sucede en otros algoritmos. Quizá el único inconveniente es que es algo más complejo de implementar que otros algoritmos, como el de ordenación rápida.

2.6. Ordenación rápida

El algoritmo de **ordenación rápida** (*quick sort* en inglés) es el más utilizado para ordenaciones, ya que reúne la facilidad de la implementación para diferentes situaciones (de la más favorable a la menos favorable) y necesita menos recursos de almacenamiento que otros.

Se basa en la técnica de "Divide y vencerás", que veremos en el Capítulo 3, sobre técnicas de diseño de algoritmos. Está dividido en **dos fases**:

1. Se divide la lista que hay que ordenar en dos listas más pequeñas, de tal manera que todos los elementos de la primera lista son menores que los de la segunda lista. El elemento clave es la elección del **pivote** o elemento que va a servir para realizar esta partición.
2. A las dos listas generadas se les aplica recursivamente el punto anterior. Como las dos listas se ordenan de forma separada sobre ellas mismas no es necesaria ninguna operación para combinarlas.

En cuanto a la **eficiencia**, este algoritmo puede ser de $O(n) = n^2$ en el peor caso o $O(n) = n \log(n)$ en el mejor caso y también en el caso promedio. Sin embargo, a diferencia de los anteriores, en este algoritmo el peor caso no depende de la mejor o peor ordenación de la lista inicial, sino de la elección del **pivote** para dividir en cada iteración las listas. Una elección del pivote que nos dé una partición equilibrada (cerca de la mediana) asegura que nos encontramos en el mejor caso. Por tanto, la eficiencia de este algoritmo depende de su diseño y no de los valores de entrada, lo que en realidad podríamos clasificar como de complejidad $O(n) = n \log(n)$, **casi lineal**.

Existen varias estrategias que pueden **mejorar la elección del pivote** en cada iteración. Una de ellas, si no disponemos de ninguna información previa sobre la distribución de los valores, consiste en elegir una muestra al azar y tomar el valor medio como pivote.

Un punto importante respecto a la implementación de este algoritmo es el momento en el que, después de algunas iteraciones, las listas que hay que ordenar son de **pocos elementos**. En este momento, suele ser más eficiente realizar las ordenaciones con otro algoritmo, como el de selección o inserción, que, aunque son de orden cuadrático $O(n) = n^2$, se comportan mejor en tamaños pequeños debido a las constantes multiplicativas de ordenación rápida.



Este procedimiento de **mezclar diferentes técnicas o algoritmos** es habitual en problemas de gran tamaño. El descubrimiento de algoritmos completamente originales es, a medida que avanza el tiempo, cada vez menos probable. De ahí que el conocimiento de diferentes técnicas y su comportamiento son la base actual para diseñar nuevos algoritmos por combinación que mejoren la eficiencia de los que ya conocemos.

Los algoritmos revisados hasta ahora son los más utilizados para ordenar listas para las que no se ha impuesto ninguna restricción o cuando se desconoce si disponen de alguna distribución que se pueda aprovechar en la ordenación. Existen algoritmos, que no estudiaremos, más específicos los cuales, bajo ciertas circunstancias, pueden aplicarse con un orden de efectividad lineal o casi lineal que mejora el de los vistos anteriormente.

2.7. Ordenación por residuos

El algoritmo de **ordenación por residuos** (*radix sort* en inglés) se utiliza para valores compuestos por letras o cifras en los que es posible definir un orden lexicográfico. El proceso consiste en clasificar en colas según la última posición de cada valor (última letra de cada palabra). El número de colas depende del número diferente de valores que puedan tomar los elementos en cada posición (27 letras en el caso de las palabras). La nueva lista vuelve a clasificarse, ahora por la penúltima letra y así sucesivamente hasta llegar a la primera posición.

La **ventaja** de este algoritmo es que solamente necesitamos tantas iteraciones como el tamaño máximo de las palabras. Este comportamiento promete buenos resultados en cuanto a efectividad, pero la mala noticia es que resulta **extremadamente caro** en cuanto a necesidad de recursos de memoria y/o almacenamiento.

Ordenación por 1. ^a cifra	Ordenación por 2. ^a cifra	Ordenación por 3. ^a cifra	FINAL
9 5 4	4 1 1	0 0 9	0 0 9
3 5 4	9 5 4	4 1 1	3 5 4
0 0 9	3 5 4	9 5 4	4 1 1
4 1 1	0 0 9	3 5 4	9 5 4

Figura 11. Ordenación según posiciones para el algoritmo por residuos.

Finalmente, para concluir el repaso de los algoritmos de ordenación, se muestra una tabla a modo de resumen con las distintas complejidades:

Tabla 2

Comparación de los órdenes de complejidad computacional y de espacio para diferentes algoritmos de ordenación

Algoritmo	Mejor	Peor	Medio	Espacio
Ordenación mezcla	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Ordenación por inserción	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Ordenación por burbuja	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Ordenación rápida	$O(n \log(n))$	$O(n^2)$	$O(n \log(n))$	$O(1)^a$
Ordenación por montículos	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$

^a En el mejor caso incluso es $\log(n)$.



Capítulo 3

Técnicas de diseño de algoritmos

3.1. Algoritmos voraces

En esta sección describimos una serie de **algoritmos clasificados como voraces**, que se utilizan especialmente en algunos problemas de optimización cuando se dan ciertas condiciones y que suelen proporcionar buenos resultados (Guerequeta y Vallecillo, 1998).

3.1.1. Características de los algoritmos voraces

La estrategia en este tipo de algoritmos consiste en **trabajar por etapas**, eligiendo en cada etapa una decisión para construir la solución que resulte más adecuada en ese momento sin considerar las consecuencias futuras. Las posibles soluciones no consideradas en la decisión serán descartadas para siempre.

Podemos enumerar una serie de **elementos que permiten identificar** un problema susceptible de ser resuelto con algún algoritmo voraz:

- Un problema con entradas para las que existe un conjunto de candidatos como solución.
- Una función de selección que en cada paso determina el mejor candidato para formar parte de la solución de entre los que no han sido descartados.

- Una función que comprueba si un determinado subconjunto de candidatos es prometedor para encontrar la solución. Es prometedor si es posible seguir adelante añadiendo candidatos en las siguientes etapas sin violar ninguna restricción.
- Una función objetivo que en cada paso puede ofrecer un valor de la solución propuesta. Es la función que queremos maximizar o minimizar.
- Una función que comprueba si un subconjunto de entradas es una posible solución al problema.

Aunque parezca una lista extensa de condiciones, la implementación **sencilla**. Esto se debe a que el proceso es bastante intuitivo. Lo podemos resumir en elegir en cada etapa la **decisión óptima**.

Esta característica tan “agresiva” de resolver problemas resulta la **más eficiente** para muchos de ellos en los que es evidente tomar la mejor decisión en cada etapa para alcanzar la mejor solución final. Sin embargo, no en todos los problemas se cumple este comportamiento. De hecho, el conjunto de estos problemas es pequeño, pero el consejo es que debemos aplicar esta técnica cuando estemos ante uno de ellos.

La naturaleza agresiva de estos algoritmos los convierte en muy eficaces. Esta eficacia se aprovecha incluso cuando sabemos que **la solución final puede no ser el óptimo** que buscamos. A veces, debemos elegir entre encontrar una solución cualquiera a un problema (aunque no sea la óptima) o tardar mucho tiempo en lograr la mejor. En estos casos los algoritmos voraces también son apropiados.



En general, **la mejor solución en cada etapa no lleva a la mejor solución final**. Esto es una dificultad añadida, ya que debemos probar de manera formal que el problema alcanza la mejor solución final si tomamos las mejores soluciones parciales. Otra dificultad añadida que se presentan al utilizar un algoritmo voraz consiste en **determinar una función de selección**. Como hemos visto, esta función es clave ya que nos permite ir encontrando las mejores opciones en cada etapa.

Estudiemos ahora un problema sencillo que nos servirá para entender el funcionamiento de este algoritmo voraz:



Ejemplo

El problema es el cálculo de **devolver el cambio** de una transacción económica. Normalmente, podemos formularlo de la siguiente forma: dado un sistema monetario de valores $(1, v_1, v_2, \dots)$, debemos descomponer cualquier cantidad C de manera que usemos el menor número posible de monedas. Debemos hacer la suposición de que disponemos de una cantidad ilimitada de cada una de las monedas.

>>>

>>>

Revisemos todos los **elementos** para construir el algoritmo:

- El **conjunto de candidatos** es el conjunto de todas las posibilidades de obtener la cantidad C con sumas de los elementos $(1, v_1, v_2, \dots)$.
- La **función de selección** en cada etapa i -ésima toma tantas monedas como sea posible sin pasarse teniendo en cuenta la cantidad acumulada en las etapas anteriores.
- La función que comprueba si el subconjunto de candidatos es **prometedor** nos la proporcionan las sumas parciales que llevamos acumuladas en todas las etapas. La suma total parcial no debe superar nunca la cantidad final C .
- La **función objetivo** es la que nos proporciona la cantidad de monedas elegidas hasta el momento de cada uno de los valores. Esta función es la que queremos minimizar.
- La función que comprueba que una elección es una **posible solución** es similar, en este caso, a la función que comprueba si un subconjunto es prometedor. Basta con sumar la cantidad acumulada, que debe coincidir con C .

A veces, no es necesario utilizar las funciones de comprobación de candidatos o de la solución porque el diseño del algoritmo ya las asegura. Por tanto, el **algoritmo** que solemos utilizar para obtener esta solución es el siguiente:

- Ordenamos los valores $(1, v_1, v_2, \dots)$, de mayor a menor. Estas serán las etapas y, una vez considerada una etapa, ya no volveremos a ella.
- En cada etapa, tomamos tantas monedas como podamos (técnica voraz) para acercarnos lo más posible a la cantidad C teniendo en cuenta la cantidad acumulada en las etapas anteriores. Se guarda en un vector V el número de monedas tomadas en cada etapa en la posición i .

La **solución** al problema es el vector V , que devuelve el número de monedas de cada valor.

El algoritmo se comporta de manera eficiente bajo algunas **consideraciones iniciales**. En primer lugar, debemos tener una moneda de **valor unitario**. En caso contrario, no es posible encontrar soluciones para algunos casos.

Otra consideración importante es que **no todos los sistemas monetarios**, $(1, v_1, v_2, \dots)$, permiten obtener soluciones óptimas por la técnica voraz. El siguiente ejemplo ilustra este inconveniente:



Ejemplo

Si el sistema de monedas es $(1, 5, 1)$, por ejemplo, el algoritmo encontrará la solución $V = (1, 0, 4)$ para alcanzar el valor $C = 15$. Sin embargo, el óptimo se consigue con la solución $V = (0, 3, 0)$.

Como se puede comprobar, el **defecto lo causa la voracidad** del algoritmo al elegir en primer lugar la moneda de 11 unidades, de modo que este ejemplo ilustra la necesidad de probar formalmente en cada caso la condición de que la optimización por etapas equivale a la optimización final.

A pesar de esta contrariedad para algunos problemas, la técnica voraz puede resultar útil en la **primera etapa** para algunos problemas complejos en los que se necesita encontrar una solución factible inicial (aunque no sea óptima) para después aplicar otras técnicas y mejorar esta solución inicial o llegar al óptimo. La técnica voraz suele ser muy útil en **combinación con otras técnicas**.

La **modelización de problemas** utilizando grafos suele resultar de especial interés, dado que permite representar eficientemente la información del tipo discreto asociada a las relaciones entre los elementos del problema. Además, es probable que ya exista un algoritmo que resuelva el problema de acuerdo a la modelización que hayamos realizado. Para uno de estos modelos revisaremos dos algoritmos voraces que nos ayuden a resolver varios tipos de problemas que aparentemente no tienen relación.

3.1.2. Uso de grafos para la modelización de problemas

Definimos un **grafo** como una pareja de conjuntos V (no vacía) de **vértices** y A de **aristas** donde cada arista se compone de un par de vértices. Gráficamente, los vértices se representan como puntos y las aristas como líneas que los unen. Si el par de vértices (u, v) o arista es diferente del par (v, u) , se dice que el grafo es dirigido y se representan las aristas con flechas. Si no se hacen distinciones entre las dos aristas que unen cada par de vértices, diremos que el grafo es no dirigido y representamos las aristas con líneas (Jordan y Torregrosa, 1996).



Dado un **grafo conexo, ponderado y no dirigido**, $g = (V, A)$, en el que deseamos encontrar un **árbol de recubrimiento mínimo**, disponemos de dos algoritmos que nos ayudan a resolver el problema. Estos dos algoritmos son el algoritmo de Prim y el algoritmo de Kruskal.

El **algoritmo de Prim** comienza con un vértice y , en cada etapa, escoge el arco de menor coste que verifica que une el subconjunto de vértices ya seleccionados con el subconjunto de vértices no seleccionados. Al incluir el nuevo arco, se añade en la lista de vértices seleccionados en nuevo vértice y se procede a la siguiente etapa. En la p. 82 de *Introducción al diseño y análisis de algoritmos* (Lee, 2005) encontramos una demostración formal por reducción al absurdo de que el árbol obtenido con este algoritmo es de coste mínimo.

En el caso del **algoritmo de Kruskal**, se ordenan en primer lugar los arcos por orden creciente de peso. Se recorren los arcos atendiendo a este orden y se van seleccionando arcos de manera que no se formen ciclos. Los arcos que pueden formar un ciclo en una etapa se descartan y no se tienen en cuenta en las siguientes etapas.

Un análisis de la eficiencia de ambos algoritmos determina que la **complejidad** de algoritmo de Prim es siempre $O(n) = n^2$, donde n es el número de vértices. La complejidad del algoritmo de Kruskal es $O(n, a) = a \cdot \log(n)$, donde a es el número de aristas.

La elección entre uno u otro depende finalmente del número de aristas, ya que, para los grafos muy densos (aquellos que conectan todos los vértices entre sí), el número de aristas se aproxima a $n(n - 1)$, por lo que el algoritmo de Kruskal se puede acercar a una complejidad $O(n) = n^2 \log(n)$ peor que la complejidad de Prim. Sin embargo, para los grafos poco densos (pocas conexiones entre nodos), el número de aristas es próximo al de los vértices y, por tanto, la complejidad de Kruskal se aproxima a $O(n) = n \log(n)$ (Lee, 2005).

3.2. Vuelta atrás

Otra de las técnicas en las que se basan un gran número de algoritmos es la **vuelta atrás** (*backtracking* en inglés), una técnica muy utilizada, ya que permite resolver algunos problemas tipo y, además, está orientada a obtener soluciones óptimas.

3.2.1. Diseño y resolución según la técnica de vuelta atrás

La base de la resolución mediante esta técnica es el **estudio detallado de las soluciones**, entre las que trataremos de encontrar la solución óptima. Para ello, el proceso construye de manera sistemática y por etapas todas las soluciones posibles.

Podemos describir los problemas que se pueden resolver con esta técnica como aquellos en los que podemos expresar la solución como una **n-tupla** (x_1, x_2, \dots, x_n) , donde cada una de las componentes x_i puede ser elegida en cada etapa.

Para realizar la implementación, se construye un **árbol de expansión**. En cada nivel del árbol situamos todos los vértices que proporcionan una parte de la solución en la etapa identificada con el nivel. Los hijos de cada nodo de un cierto nivel constituyen nuevas partes de la solución para la siguiente etapa.

A medida que avanzamos por el árbol desde la raíz, vamos construyendo las posibles soluciones. Puede ocurrir que, llegados a un nodo, se verifique que dicha rama nunca proporcionará una solución. En ese caso, lo descartaremos y volveremos hacia atrás (acción que da nombre a la técnica) en el árbol hasta el nodo en el que aún podamos encontrar posibles soluciones. En otro caso, podemos llegar a una hoja (nodo final) que podamos verificar que es una solución al problema. La registraremos con su coste (el valor de aplicar la función de coste) y continuaremos la exploración de nuevo con vuelta atrás.

La **modelización** del problema para poder aplicar esta técnica pasa por identificar el problema en etapas y en cada etapa establecer los posibles valores que van componiendo las soluciones. Una vez identificados estos elementos, la construcción del árbol de expansión es relativamente sencilla.

En términos de **eficiencia** observamos que esta técnica se asemeja a un proceso de prueba y error en el que la estructura en árbol nos facilita la tarea. Es evidente que, en algunos problemas, puede llegar a ser exponencial y, por tanto, mejorable.

Las **mejoras** que se pueden introducir dependen de cada problema concreto. Estas mejoras vienen dadas por la capacidad de descartar nodos en los que, aunque no haya que explorar en profundidad, podamos asegurar que no van a formar parte de una solución (o de una solución óptima, si es lo que buscamos). Esta función de descarte debe ser tan sencilla en términos computacionales como la propia exploración de rama, pues en caso contrario estaríamos empeorando el rendimiento.



Ejemplo

Un problema que nos permite ilustrar esta técnica es el de las **ocho reinas**. Se trata de situar 8 reinas sobre un tablero de ajedrez sin que se amenacen entre sí. Para simplificar el problema y representar gráficamente el árbol de expansión, reducimos el problema a 4 reinas en un tablero de 4×4 , tal como vemos en la Figura 12:

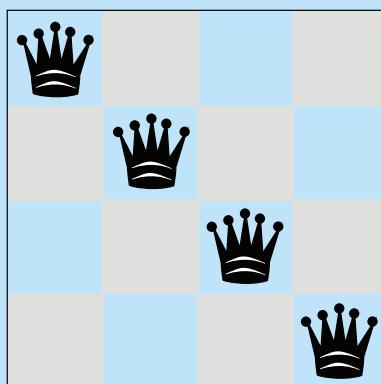


Figura 12. Cuatro reinas en un tablero de 4×4 correspondiente a la representación del vector $(1, 2, 3, 4)$.

La solución estará representada por **vectores de dimensión 4**, donde cada una de las posiciones indica la columna, y el valor (de 1 a 4) indica la fila correspondiente donde se sitúa cada reina. De esta manera, el vector $(1, 2, 3, 4)$ es el que representa la colocación de las cuatro reinas en la diagonal principal del tablero. Obviamente, esta situación no es solución al problema pues no se cumple que haya reinas no amenazadas.

La implementación del algoritmo para obtener una solución recorre un **árbol de expansión** en el que los niveles del árbol representan las posibles situaciones de las reinas en cada columna del tablero. De esta manera, es sencillo determinar en cada nodo si puede ser o no solución. Diremos que el nodo representa una solución k -prometedora en el nivel k si no vulnera la restricción de amenaza entre reinas.

A medida que construimos la solución, podemos descartar nodos mediante la aplicación de algunas **reglas sencillas para no evaluar el árbol completo**. En primer lugar, no puede haber dos reinas en la misma columna. Esta restricción garantiza la estructura de datos que hemos elegido para representar las soluciones. En cada coordenada del vector solo puede haber un valor. En cada fila tampoco puede haber más de una reina. Esta restricción la podemos verificar con una función que descarte una rama en la que se ha repetido un valor. Por ejemplo, el nodo asociado al vector $(1, 1, *, *)$ representa la situación de dos reinas situadas en la primera fila. Por último, debemos descartar también las amenazas a través de diagonales. Si analizamos las coordenadas que están determinadas por las filas y las columnas, veremos que dos reinas están en la misma diagonal si, siendo sus posiciones (x_1, y_1) y (x_2, y_2) , se cumple que $|x_1 - y_1| = |x_2 - y_2|$ (el valor absoluto de la diferencia de sus coordenadas coincide). Basta con incorporar esta comprobación para descartar o no un nodo.

>>>

>>>

En el diagrama de la Figura 13 vemos como la rama asociada a situar la primera reina en la primera columna lleva siempre a situaciones que no permiten alcanzar ninguna solución:

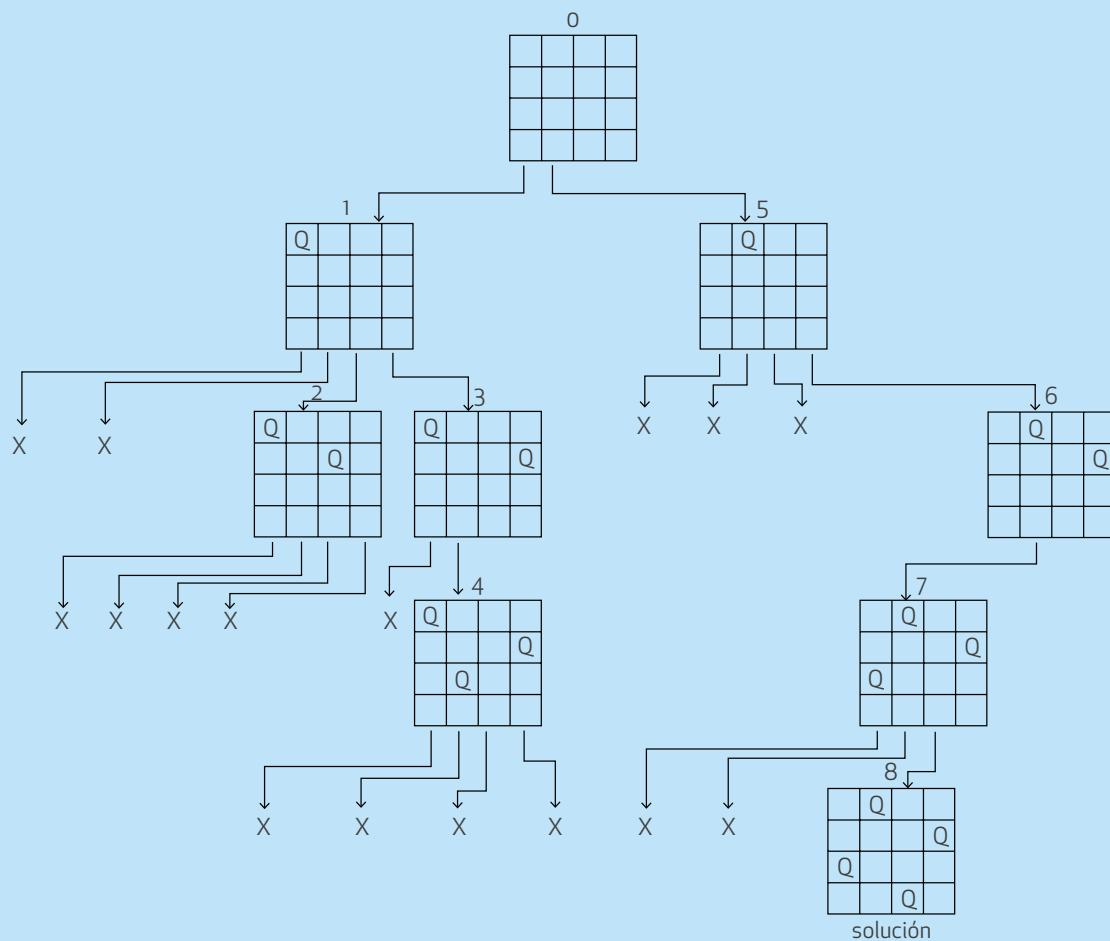


Figura 13. Diagrama que muestra algunas composiciones de soluciones en el algoritmo de vuelta atrás para el problema de las cuatro reinas.

Finalmente, para concluir con la descripción de la técnica de vuelta atrás, es interesante considerar la importancia de otra técnica de ámbito más general a la computación, la **recursividad**. Aunque la analizaremos en la sección “3.3. Divide y vencerás”, es preciso tenerla en cuenta en este tipo de algoritmos, dado que, si nos fijamos, en cada etapa realizamos casi siempre las mismas acciones con la única excepción de que vamos añadiendo elementos en una estructura de datos (vector solución en el ejemplo) prefijada de antemano.

El ejemplo de las cuatro reinas ilustra cómo podemos obtener todas las soluciones de un problema. En este caso, puesto que no teníamos una función de evaluación, no tiene sentido hablar de óptimos.

Sin embargo, en otro problema en el que sí dispusiéramos de una función de evaluación, podríamos ir guardando la mejor solución en cada momento para conseguir obtener el **óptimo** al final del proceso.

3.3. Divide y vencerás

Otra técnica que debemos tener siempre presente para resolver problemas es la de “**Divide y vencerás**”. En este caso, el concepto va más allá del ámbito computacional, ya que también se emplea en la resolución de problemas de otras áreas. Seguro que la hemos empleado en la vida cotidiana sin ser conscientes de ello.

En el contexto computacional, la técnica consiste en resolver el problema principal a partir de las **soluciones que proporcionan los subproblemas** del mismo tipo pero de menor tamaño. En algún momento, mediante el proceso de hacer más pequeños los subproblemas, llegamos a una solución sencilla o trivial. En este tipo de algoritmos se hace evidente el uso de la recursividad, que vimos en el capítulo anterior.

Para reconocer que estamos ante un problema resoluble con la técnica de “Divide y vencerás” debemos identificar los siguientes **requisitos**:

1. El problema puede ser descompuesto en subproblemas del mismo tipo que el original pero de menor tamaño en cada descomposición.
2. Debemos poder resolver los problemas o bien de manera directa (si son sencillos) o bien de manera recursiva.
3. Debemos ser capaces de combinar las soluciones de los subproblemas para lograr la composición de la solución al problema original.

Siempre que usemos la **recursividad**, debemos asegurar que la implementación finalizará en algún momento, pues en caso contrario nos podemos encontrar con la sorpresa de que el proceso no finaliza nunca o finaliza con algún error asociado a alguna limitación de memoria. Para los algoritmos de “Divide y vencerás”, la finalización está asegurada si aseguramos que, en cada etapa, el subproblema es más pequeño, siempre que nos movamos en términos discretos.

La recursividad nos proporciona **implementaciones sencillas y buena legibilidad** de cara a posibles futuras depuraciones. En su contra, debemos **evaluar la complejidad** en cuanto a uso de memoria o espacio en disco que suponga su implementación. En cada iteración estamos multiplicando los recursos espaciales.

En cuanto a la eficiencia de estos algoritmos, podemos obtener **complejidades excelentes** (lineales o casi lineales) si se dan algunas condiciones. Para obtener el orden de complejidad de un algoritmo del tipo “Divide y vencerás”, debemos estar familiarizados con las ecuaciones en recurrencia.

La forma general de una **ecuación en recurrencia** de estos algoritmos es la siguiente:

$$T_n = \begin{cases} c \cdot n^k, & 1 \leq n < b \\ a \cdot T\left(\frac{n}{b}\right) + c \cdot n^k, & n \geq b \end{cases}$$

En la ecuación anterior:

- a es el número de subprogramas que obtenemos.
- n/b es el tamaño del nuevo subprograma en cada iteración.
- cn^k es el coste que puede suponer realizar las descomposiciones y la posterior combinación.

Queda fuera de este alcance la resolución de ecuaciones en recursividad. Podemos mejorar nuestros conocimientos sobre este tema consultando Guerequeta y Vallecello (2000, cap. 1).

La ecuación anterior tiene las siguientes **soluciones** expresadas en términos de complejidad según cómo están relacionados los valores a y b :

- $T(n) = O(n^k)$ si $a < b^k$
- $T(n) = O(n^k \cdot \log(n))$ si $a = b^k$
- $T(n) = O(n^{\log_b(a)})$ si $a > b^k$

Como vemos, la complejidad nunca es mayor que la polinomial, por lo que siempre se obtienen buenos comportamientos. Las diferencias dependen de los valores de a y b , que a su vez vienen determinados por el número de subproblemas y el tamaño de estos, respectivamente.

Si recordamos el algoritmo de ordenación rápida, estudiado en el Capítulo 2, podemos observar las características de “Divide y vencerás”: una **división equilibrada** de los subconjuntos que hay que ordenar **supone un rendimiento más eficiente**. Esto es así en general en la teoría de “Divide y vencerás”. Un diseño del algoritmo que realice divisiones equilibradas siempre proporciona mejor eficiencia.

Otra consideración que no debemos obviar es la complejidad y, por tanto, el coste que puede suponer realizar las subdivisiones y las combinaciones posteriores de los subproblemas. Un representante de los problemas con solución basada en “Divide y vencerás” es el conocido juego de las **torres de Hanói** (Gómez y Cervantes, 2014):



Figura 14. Juego de las torres de Hanói.

El juego consiste en trasladar una torre con varios niveles de una posición a otra. Existe una tercera posición a modo de ayuda o pivote para construir las torres parciales. En cada torre solo podemos extraer el nivel superior y debemos situarlo en cualquiera de las otras dos posiciones. Además, no podemos situar un nivel inferior por encima de uno superior. La elección de las estructuras de datos para las torres y la posición facilita el diseño del algoritmo que resuelve el juego.

La estrategia para conseguir el diseño correcto está en darse cuenta de que podemos dividir el problema en etapas. En la primera etapa, para un problema de N torres, podemos diseñar un **pseudocódigo** similar al siguiente:

Resolver (n, A, B, C):

Resolver ($n - 1, A, C, B$)

Mover (A, C)

Resolver ($n - 1, B, A, C$)

La función recibe **cuatro parámetros**:

1. El número que debemos resolver.
2. Posición donde se encuentra la torre que hay que resolver.
3. Posición pivot.
4. Posición destino.

En realidad, el cuarto parámetro no sería necesario porque viene determinado por el conocimiento del segundo y el tercero.

3.4. Programación dinámica

Algunas técnicas revisadas, como "Divide y vencerás", hacen uso del **modelo de recursividad** para resolver problemas que pueden descomponerse en otros más pequeños. En ocasiones, se produce una repetición de los cálculos producido por el solapamiento de los problemas en los que se descompone el problema principal, lo que produce una ineficiencia en el algoritmo.

Un ejemplo sencillo de entender y que ilustra esta situación es el que se produce al calcular los términos de la **sucesión de Fibonacci** a través del algoritmo de "Divide y vencerás". Puesto que cada iteración en el cálculo de cada término depende de los dos anteriores, nos vemos obligados a repetir cálculos para todos los términos:

$$T(n) = T(n-1) + T(n-2)$$

Por ejemplo, para el quinto término, $T(5)$, debemos calcular los términos $T(4)$ y $T(3)$. Pero para el término $T(6)$, el proceso repite el cálculo para $T(4)$. La recursividad agrava esta situación para convertir el proceso en **exponencial en términos de complejidad**, pues traslada la repetición a todos los niveles inferiores. Esta recursividad, que nos favorece en los casos en los que la descomposición genera problemas independientes, se vuelve en nuestra contra si hay solapamiento.

Para evitar estas situaciones, acudimos a la **programación dinámica**. A costa de un pequeño suplemento en el coste de almacenamiento, evitamos la repetición de cálculos, ya que el algoritmo va guardando en una tabla todos los cálculos intermedios.

La base de esta técnica se fundamenta en el **principio de optimalidad** de Bellman, enunciado en 1957, que dice que en una secuencia de decisiones toda subsecuencia también es óptima.

Es evidente que este principio no se da en todos los problemas, por lo que nos vemos obligados a garantizarlo para aplicar esta técnica. Un contrajeemplo evidente es el problema del viajante que veremos en el Capítulo 4, sobre problemas tipo. En este caso, dado el camino más corto entre dos puntos, no podemos asegurar que sean óptimos los caminos parciales entre otros dos puntos intermedios.



En vista de lo expuesto, las dos **condiciones generales** que deben cumplir los problemas para ser tratados por programación dinámica son los siguientes:

- Se puede alcanzar la solución a través de una secuencia de decisiones.
- La secuencia cumple el principio de optimalidad.

A diferencia de otras técnicas o algoritmos en los que la **complejidad** no resulta tan importante, en los algoritmos basados en programación dinámica, sí debemos analizarla en cuanto al uso de memoria o espacio, debido a la necesidad de guardar información nueva en cada una de las iteraciones. Para muestra de diseño e implementación de esta técnica, analizaremos el problema del viaje por el río más barato:



Ejemplo

En un río hay n embarcaderos y debemos desplazarnos río abajo (no hay posibilidad de remontar) desde un embarcadero a otro. Cada embarcadero tiene precios diferentes para ir de un embarcadero a otro situado más abajo. Para ir del embarcadero i al j , puede ocurrir que sea más barato hacer un trasbordo por un embarcadero intermedio k . El problema consiste en determinar la **combinación más barata**.

Consideramos una tabla $T(i,j)$ para almacenar todos los precios que nos ofrece el embarcadero i hasta cualquier otro embarcadero. Si no es posible desplazarse desde un embarcadero a otro, entonces configuramos un valor grande en T para asegurarnos de que este trayecto nunca se elegirá.

Como en todos los problemas de programación dinámica, necesitaremos construir una tabla para almacenar los valores intermedios. La naturaleza del problema nos permite definir una función de recursividad de la siguiente manera para i diferente de j :

$$C(i,j) = \min \{C(i,k) + C(k,j), C(i,j)\} \text{ para todo } k \text{ entre } i \text{ y } j$$

Aunque la expresión pueda ser algo compleja, la explicación es muy intuitiva. Significa que el coste mínimo para ir de i a j es el mínimo entre de todas las posibilidades de ir de i a j por algún embarcadero intermedio, incluido el propio trayecto directo de i a j .



Capítulo 4

Problemas tipo

En este capítulo describiremos algunos de los problemas clásicos de optimización. Existen muchos más que pueden encontrarse en la red.



Enlace de interés

La web del Kungliga Tekniska Högskolan (Real Instituto de Tecnología de Estocolmo) contiene una buena lista de problemas de optimización:

<https://www.nada.kth.se/~viggo/problemstlist/compendium.html>

4.1. Problema del agente viajero

El **problema del agente viajero** o TSP (*travelling salesman problem* en inglés) es uno de los problemas con más estudios entre los problemas de optimización. Se puede enunciar como la búsqueda óptima del camino que debe realizar un viajero para recorrer una serie de ciudades de tal manera que la distancia (u otra variable asociada a un coste) sea mínima. Además, solo debe pasar una vez por cada ciudad y regresar a la ciudad de partida.

Es un **problema clásico** que se plantea en la vida real para muchos problemas de reparto de mercancías, aunque la realidad siempre complica el modelo introduciendo otros elementos: varios viajeros con diferentes capacidades, horarios, demoras y otros.



El problema se modela como un grafo en el que **los nodos son las ciudades y las aristas son las posibles rutas**, que llevan asociado el coste o la distancia para desplazarse entre los nodos que conectan. Podemos considerar las posibles soluciones como un vector $V = (v(1), v(2)\dots)$, donde cada coordenada indica el nodo o ciudad visitada en orden.

A pesar de la sencillez del enunciado y de compresión, resulta ser uno de los problemas más difíciles de resolver, **NP-difícil**, para casos de gran tamaño. Para un grafo de n vértices, el tamaño del espacio de búsqueda se puede ir al orden de $(n - 1)!/2$.

El problema del agente viajero puede resolverse de diferentes maneras:

- **Fuerza bruta** o revisión exhaustiva de todas las soluciones factibles. Se analizan todas las posibles soluciones al problema, se calculan los costes asociados y se identifica, por comparación, cuál es la solución con el coste más conveniente.
- **Métodos exactos.** También llamados *algoritmos óptimos*, intentan descartar ramas del grafo asociado como posibles soluciones, lo cual hace más eficiente la búsqueda y la obtención de la solución óptima. El algoritmo más usado para resolver el TSP por este método es la ramificación y poda, que veremos en el Capítulo 5, sobre algoritmos de búsqueda.
- **Métodos heurísticos.** Son métodos que obtienen buenas soluciones en tiempos de cómputo muy cortos, pero no garantizan la solución óptima. Veremos en detalle algunos de ellos en el Capítulo 7, sobre métodos heurísticos y metaheurísticos. Así como los dos anteriores están al alcance computacional para resolver problemas pequeños, los métodos heurísticos son imprescindibles para problemas de gran tamaño.

Existen multitud de **ámbitos en los que es aplicable** el problema del agente viajero:

- **Circuitos electrónicos**, para minimizar la cantidad de soldadura utilizada, minimizar el espacio entre los puntos de soldadura de los circuitos.
- **Previsión del tránsito terrestre**, para minimizar el tiempo de traslado según horas y tipos de vehículos.
- **Entrega de productos**, para minimizar recorridos o tiempos de traslado.
- **Estaciones de trabajo**, para minimizar el tiempo en la secuencia de actividades.
- **Edificación**, para minimizar la distancia o combustible usado para transportar materias primas entre las construcciones y los puntos de extracción.
- **Otros**: genética, ingeniería, procesos de producción, computación...

Existen además infinidad de **variantes** o modificaciones al problema original, que dan nombre a nuevos problemas tipo: TSP simétrico, TSP con cuello de botella, TSP con múltiples visitas, etc.

4.2. Problema de la mochila

El **problema de la mochila** o KP (*knapsack problem* en inglés) es otro de los problemas clásicos de optimización combinatoria. Se puede enunciar como la forma de encontrar la mejor elección entre una serie de objetos que tienen asociados un valor y un peso cada uno, de tal manera que el valor total de los objetos sea máximo y el peso no supere un cierto límite.

A pesar de la sencillez del enunciado y de comprensión, el problema es **NP-completo** y, por tanto, su resolución es compleja para tamaños grandes. Ha sido objeto de estudio desde mediados del siglo pasado. Existen dos aspectos que lo convierten en un problema especial destacado en la investigación de operaciones. El primero es que, dada su naturaleza, es un problema para el que no es posible disponer de un algoritmo general que pueda abordarlo, por lo que es necesario tratarlo de manera particular. El segundo aspecto que lo hace tan interesante es que existen muchas situaciones en la vida real que se pueden modelizar como un problema de la mochila.

El problema de la mochila puede resolverse de diferentes maneras:

- Modelizado como problema de **programación lineal entera** (PLE), que veremos en la sección siguiente, dado que consiste en maximizar una función cuyas variables están sujetas a restricciones lineales.
- **Técnicas voraces**, que vimos en el capítulo anterior.
- **Algoritmos genéticos** cuando el tamaño del problema es grande.

Existen **variantes** al problema para adaptarlo a situaciones reales. Una de ellas es el **problema de múltiple elección**, en el que los objetos están agrupados y hay que elegir un representante de cada grupo para incluirlo en la mochila.

4.3. Problema de programación lineal entera

Los **problemas de programación lineal entera** o **PLE** son unos problemas particulares de la programación lineal en los que todas o parte de sus variables deben tener valores enteros.



Ejemplo

En la **planificación de turnos de un centro de trabajo** se desea minimizar el número de empleados en cada turno, donde se deben cumplir algunas condiciones. Estas condiciones se traducen en restricciones de igualdad o desigualdad que deben cumplir las soluciones válidas. Cada empleado debe tener, entre otros:

- Pausas de 30 minutos cada 3 horas.
- Descansos de 2 días seguidos.
- Descansos obligatorios cada 7 días seguidos de trabajo.

La solución aceptable debe ser el número de trabajadores asignados a cada turno, por lo que no tiene sentido considerar valores no enteros.

Repasaremos los **problemas de programación lineal** antes de entrar en las peculiaridades de los que se restringen por incorporar variables enteras. Un problema de programación lineal es aquel que trata de maximizar (o minimizar) una función objetivo lineal sujeta a unas restricciones de igualdad o desigualdad también lineales. Existen muchos problemas que se enmarcan en este modelo, para el que disponemos de un método efectivo de resolución: el **método Simplex**. Aunque no es de complejidad polinomial, se comporta bien para la mayoría de los casos.

Históricamente, la preocupación por estos problemas se remonta a la época de la Segunda Guerra Mundial, para abordar **cuestiones logísticas y de abastecimiento**. El área de investigación de problemas de programación lineal es importante, dado que existen muchos problemas que pueden reducirse a problemas de programación lineal. En 1984, Narendra Karmarkar, introdujo un nuevo método, el **método del punto interior**, que asegura la resolución en tiempo polinomial y es, por tanto, muy adecuado para problemas grandes. Sin embargo, para problemas pequeños el método Simplex sigue siendo vigente porque mejora el del punto interior en estos casos. La estrategia de este método varía respecto a la del Simplex en el sentido de que, en lugar de recorrer los vértices del espacio geométrico de las soluciones, parte de un punto interior de dicho espacio (no frontera) y, con el vector gradiente, se mueve a través de puntos intermedios hasta encontrar el óptimo. Veremos en el Capítulo 6, sobre el descenso de gradiente, que esta técnica también se aplica a otros problemas de optimización no lineal.

Volviendo a los **problemas de programación lineal entera**, existe una primera **clasificación** en lo que se refiere a las variables del problema:



Si todas las variables del problema deben ser enteras, el problema se considera **puro** y, en caso contrario, **mixto (PEM)**. Una particularización de los problemas puros son aquellos en los que las variables solo toman valores binarios. Se trata de los **problemas de programación binaria (PEB)**. Estos problemas vienen dados por enunciados en los que es necesario tomar decisiones de tipo sí o no.

Hay varios problemas que se enmarcan en este modelo: el de la mochila, en el que las variables binarias indican la decisión de incluir o no un objeto en la mochila; el problema de la cartera de inversiones, para determinar si hay que tener en cuenta o no una determinada inversión; un problema de asignación de tareas a recursos, etc.

En cuanto a la **estrategia** para resolver problemas de programación entera, podría suponerse que deben ser tan sencillos de resolver como el equivalente dentro de la programación lineal (sin las restricciones de que las variables deben tener valores enteros), pero esto no es así. Además, podríamos caer en la tentación de resolver el problema no entero y redondear la solución para que se adapte a las restricciones de valores enteros. Sin embargo, este camino nos puede conducir a errores, puesto que la solución puede no ser la óptima y, además, ser muy lejana, o directamente puede que el redondeo conduzca a una solución no factible. Un ejemplo sencillo para el primer caso sería el problema:

$$\max 4x + 3y$$

El problema está sujeto a las siguientes restricciones:

$$2x + y \leq 2$$

$$3x + 4y \leq 6$$

$$x, y \in [0,1]$$

El problema de programación proporciona la solución $x_1 = 0,4$ y $x_2 = 1,2$ que, al redondearla, nos daría la solución $x_1 = 0$ y $x_2 = 1$. Sin embargo, la solución óptima en realidad es otra: $x_1 = 1$ y $x_2 = 0$, como se ve en la Figura 15:

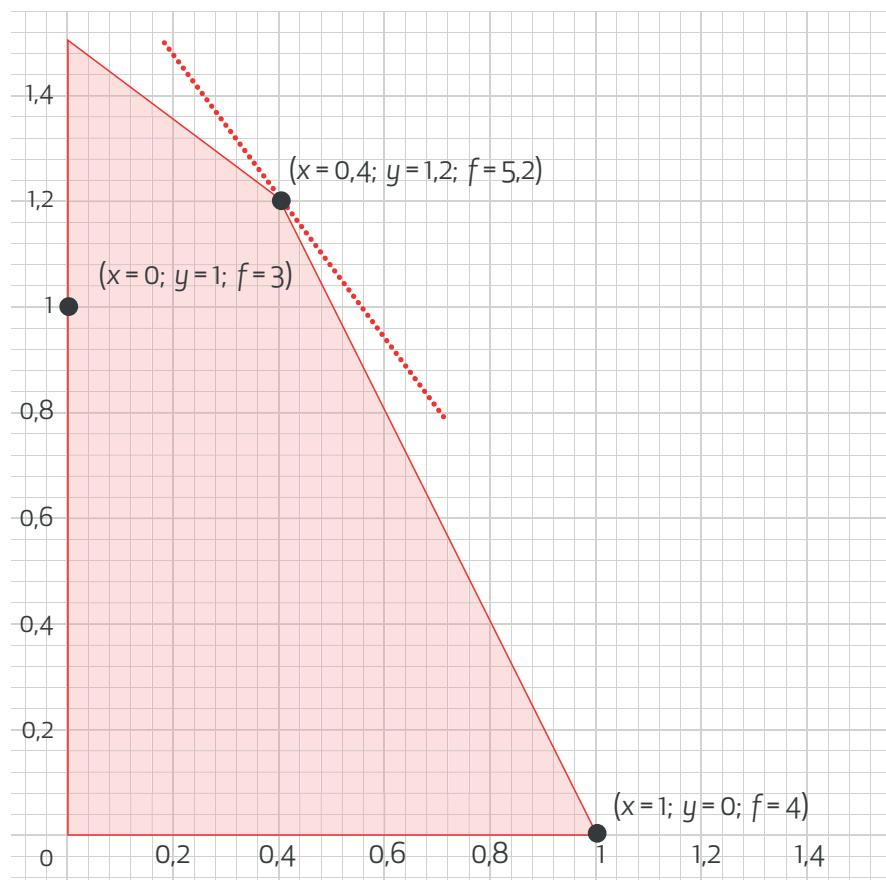


Figura 15. Representación de un problema de optimización de programación entera para ilustrar que el redondeo no conduce a la solución óptima.

4.4. Problema de enrutamiento

El **problema de enrutamiento** o **VRP** (*vehicle routing problem* en inglés) deriva del problema del viajante, pero en este caso hay que encontrar no una sino varias rutas óptimas para una cantidad determinada de vehículos en la flota que deben entregar determinadas mercancías. Por tanto, se trata también de un problema particular tanto de optimización combinatoria como de programación lineal entera.

Como en el caso del problema del viajante, puede **modelizarse con un grafo**. Pueden incorporarse algunas variantes para adaptarlo a las necesidades de los problemas reales. Por ejemplo, podemos determinar que el grafo sea dirigido para establecer costes diferentes dependiendo del sentido del trayecto.

Una de las variantes más analizadas consiste en incluir **ventanas de tiempo** que se traducen en restricciones de obligatoriedad (o penalización) para realizar una determinada entrega en un determinado rango horario. Podríamos tener en cuenta otras variantes considerando el origen y/o los destinos obligados de cada uno de los vehículos de la flota.

Existen **algoritmos exactos** que resuelven el problema para tamaños no muy grandes del problema. Es posible utilizar técnicas de programación dinámica o búsqueda de ramificación y poda, que veremos en el Capítulo 5, sobre algoritmos de búsqueda, para resolverlos de manera exacta. En Kallehauge (2008) se hace una revisión de los algoritmos exactos para resolver el problema.

Dado que el problema es **NP-completo**, cuando el tamaño del problema es grande, debemos explorar los métodos aproximados, puesto que con los algoritmos exactos el problema se hace inabordable. En el Capítulo 7, sobre métodos heurísticos y metahuerísticos, veremos técnicas metaheurísticas, como la optimización por colonia de hormigas, la optimización por enjambre de partículas o algoritmos genéticos que dan buenos resultados para resolverlo.

El problema del enrutamiento es de gran importancia en la **industria logística** debido a los ahorros que puede producir mejorar las decisiones acerca de las rutas de las flotas. Esto hace que se convierta en un área de estudio constante, puesto que no es posible resolverlo de manera exacta. Cualquier mejora, por pequeña que sea, suele traducirse en grandes cantidades de dinero ahorrado para las grandes compañías de la industria. Esto hace que exista un buen catálogo de programas informáticos tanto comerciales como libres para ayudar a resolver el problema.

4.5. Otros problemas

El primer problema que se demostró que pertenecía a la clase NP-completo es el **problema de satisfacibilidad booleana** o **SAT**, enunciado por Cook (1971) de la siguiente manera:

Dadas una lista de variables booleanas (con valores de tipo verdadero o falso) $\{x_1, x_2, \dots, x_n\}$ y una combinación de estas variables con operaciones lógicas (*and*, *or* y *not*), el problema es determinar si existe alguna combinación que haga que la expresión tenga un resultado verdadero. Una de las alternativas para resolver este tipo de problemas son los algoritmos genéticos, que veremos en el Capítulo 8, sobre algoritmos evolutivos y genéticos.

Otro problema tipo es el **problema del transporte**, en el que debemos minimizar el coste total de transportar una determinada mercancía desde unos puntos de abastecimiento hasta otros puntos de destino teniendo en cuenta los costes individuales de cada origen a cada destino. Se plantea como un problema de programación lineal y, en general, es posible resolverlo a través del método del Simplex, pero en casos con muchos datos es posible abordarlo con técnicas heurísticas. Una variante es el **problema del transbordo**, en el que se incorporan nodos intermedios al origen y destino que hacen las veces de almacenes intermedios.

Muy relacionado con los anteriores se encuentra el **problema de la asignación**. En este caso, es necesario realizar n tareas por n agentes con un coste asociado a cada asignación, y se debe minimizar el coste total de las asignaciones.

El **problema del recubrimiento** o **SCP** (*set covering problem* en inglés) se enuncia de la siguiente manera. Dados un conjunto de elementos llamado *universo*, por ejemplo, $\{1,2,3,4,5\}$, y un conjunto de subconjuntos de estos, por ejemplo, $\{\{1,2,3\}, \{3,5\}, \{1,4,5\}, \{1,5\}, \{4,5\}\}$, debemos encontrar una cantidad mínima de estos subconjuntos que cubran todo el universo. En el ejemplo, el óptimo sería $\{1,2,3\}$ y $\{4,5\}$.



Enlace de interés

Se pueden consultar otros problemas asociados a la optimización combinatoria en la web del Kungliga Tekniska Högskolan (Real Instituto de Tecnología de Estocolmo):

<https://www.nada.kth.se/~viggo/problemslist/compendium.html>



Capítulo 5

Algoritmos de búsqueda

Los algoritmos de búsqueda merecen un lugar destacado, dado que, en un problema para el que podemos conocer todas las soluciones posibles y representarlas en una estructura de árbol, es posible establecer un **método de búsqueda** para obtener una solución óptima. El otro requisito necesario es disponer de una función objetivo, que es la que se desea optimizar.



Desde el punto de vista de la optimización, no podemos olvidar si el método de búsqueda que empleamos es **global o local**:

En el caso del **método global**, el algoritmo asegura que la solución encontrada es la mejor posible. Evidentemente, es preferible usar un método global si es posible, pero no siempre lo es: o bien porque no existe o porque, aunque exista, es intratable por las características del problema.

En ese caso debemos usar algún algoritmo con método basado en **búsquedas locales**. La solución encontrada posiblemente no será la mejor, pero sí será una buena solución. Estos métodos de búsqueda local suelen partir de una solución cualquiera y van mejorando iterativamente hasta llegar a un punto en el que no es posible mejorar. La bondad de la solución encontrada suele depender de la calidad de la solución inicial propuesta. Es habitual, si disponemos de esta posibilidad, ejecutar el algoritmo para diferentes soluciones de partida y elegir la mejor.

En este capítulo se analizan algunas estrategias para realizar búsquedas para los problemas en los que podemos modelizar con una **estructura de árbol**. Estas estrategias van encaminadas, en general, a realizar podas en el árbol.

5.1. Búsqueda en amplitud

La **búsqueda en amplitud o en anchura** o **BFS** (*breadth first search* en inglés) se basa en analizar todos los nodos de un mismo nivel antes de seguir con los del siguiente. Para implementar esta técnica, se necesita una estructura de cola que permita guardar los nodos expandidos.

El **esquema general** que debe seguir el algoritmo es el siguiente:

1. Formar una cola con el elemento raíz del árbol.
2. Probar si el primer elemento de la cola es solución final. En caso afirmativo, terminamos. En caso contrario, seguimos con el paso 3.
3. Quitar el nodo explorado en el paso 2 y añadir al **final** de la cola todos sus nodos hijos.
4. Si en la cola no hay nodos, entonces no podemos encontrar solución. En caso contrario, repetimos el paso 2.

Un caso particular de este tipo es el **algoritmo A***. En este caso introducimos un componente heurístico en la evaluación de cada uno de los nodos, de tal manera que la función de evaluación se compone como suma de otras dos funciones:

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

En la fórmula anterior:

- $f(n)$ es la función que evalúa cada nodo.
- $g(n)$ es el coste real del camino recorrido desde la raíz hasta el nodo n .
- $h(n)$ es una función que representa un valor heurístico del camino desde n hasta el nodo final.

5.2. Búsqueda en profundidad

A diferencia de la anterior, la técnica de **búsqueda en profundidad** o **DFS** (*depth first search* en inglés) analiza primero los nodos más profundos. También hace falta una pila para ir almacenando los nodos que se van tratando.

El **esquema general** del algoritmo es el siguiente:

1. Formar una cola con el elemento raíz del árbol.
2. Probar si el primer elemento de la cola es solución final. En caso afirmativo, terminamos. En caso contrario, seguimos con el paso 3.

3. Quitar el nodo explorado en el paso 2 y añadir al **principio** de la cola todos sus nodos hijos.
4. Si en la cola no hay nodos, entonces no podemos encontrar solución. En caso contrario, repetimos el paso 2.

Es posible **mejorar esta técnica** desde el punto de vista de la eficiencia si somos capaces de ordenar la entrada en la pila de los nodos hijos de acuerdo con algún criterio que asegure que algunos de estos nodos hijos son mejores que otros. La mejora pasa por encontrar alguna función de evaluación que permita compararlos.

En realidad, esta manera de mejorar la veremos en el Capítulo 7, sobre metaheurísticas, pero la introducimos en combinación con la búsqueda en profundidad por ser muy usadas conjuntamente en la práctica. La incorporación de una función de evaluación que nos permita una comparación entre los nodos intermedios (o soluciones parciales) es conocida como **heurística**.

La combinación del método de búsqueda en profundidad junto con una función de evaluación entre nodos candidatos es conocida como **ascenso de colina** (*hill climbing* en inglés).

En algunos problemas, puede tener sentido combinar incluso los dos métodos, la búsqueda en profundidad y en amplitud junto con una función de evaluación, tal como en la variante del ascenso de colina. En este caso, evaluamos en cada momento todos los nodos de la pila, tanto los incluidos por amplitud como por profundidad. Esta estrategia recibe el nombre de **primero el mejor** (*best first search* en inglés).

Un caso particular de algoritmo de búsqueda en profundidad es el **MinMax**, que se emplea en juegos con adversarios. Proviene de la teoría de juegos enunciada por John von Neumann en la década de los años veinte.

Von Neumann demostró que, en los **juegos de suma cero** donde cada jugador conoce de antemano las todas las posibilidades del otro, existe una estrategia para minimizar la pérdida esperada. En la práctica se traduce en considerar el árbol con todas las posibilidades y tomar elecciones que minimicen la pérdida máxima. Si ambos jugadores actúan con esta estrategia, las jugadas coinciden y se dice que el juego está resuelto y determinado.

Aplicado a la búsqueda de soluciones en un árbol, el proceso supone generar todos los posibles nodos, calcular los valores finales y determinar el camino que ha llevado desde la posición raíz hasta el nodo final de mayor valor.

En general, el algoritmo no es práctico para la mayoría de los problemas de optimización para los que se generan grafos muy extensos. Sin embargo, debemos conocerlo para los problemas que son abordables a través de esta técnica porque nos garantiza obtener la mejor solución. Dos ejemplos para ilustrar estas dos situaciones son el ajedrez y el tres en raya. En el caso del ajedrez, sería excesivamente ingenuo intentar construir un algoritmo mediante esta técnica. El número de nodos se dispara y lo hace inviable. Sin embargo, en el otro extremo se encuentra el juego de tres en raya. En este caso, usar esta técnica es una buena opción, ya que el número de nodos es limitado y es posible evaluar todas las posibilidades.

5.3. Ramificación y poda

En las técnicas de amplitud y profundidad, se revisaron estrategias para recorrer un árbol de diferentes maneras para encontrar una solución. La técnica de **ramificación y poda** (*branch and bound* en inglés) va un poco más allá, ya que permite obtener soluciones óptimas para problemas complejos.

Durante la exploración exhaustiva del árbol, se trata de **eliminar** definitivamente de la exploración aquellas **ramas que sabemos que no van a mejorar las** soluciones que ya hemos encontrado. A diferencia de las técnicas de amplitud y profundidad, en las que se realiza una expansión del árbol de soluciones de manera sistemática, para el caso de ramificación y poda establecemos en cada nodo una cota del posible valor de todas las soluciones alcanzables desde dicho nodo. Si dicha cota resulta ser peor que la de la mejor solución alcanzada hasta el momento, entonces se decide no explorar dicha rama y, por tanto, podarla (Guerequeta y Vallecillo, 2000).

La forma de **implementar** la ramificación y poda consiste en establecer una pila para almacenar los nodos que aún no han sido explorados y determinar una función de coste para estimar en cada nodo una cota asociada. Esta función de coste determina en cada momento cuál es el nodo más prometedor y permite descartar otros.

El proceso se realiza en **tres etapas**:

- En la primera, la de **selección**, se selecciona uno de los nodos que quedan por explorar.
- En la segunda etapa, la **ramificación**, se expanden los posibles nodos hijos.
- En la tercera y última etapa, la **poda**, se eliminan los nodos hijos que no mejoran la cota obtenida hasta el momento.

Obviamente, al comienzo del proceso, debemos llegar a una **primera solución**, que es la que marcará la primera referencia para las siguientes cotas.

Como puede anticiparse, la dificultad de esta técnica consiste en encontrar una buena **función de coste** que garantice la correcta estimación del coste de las soluciones para cada nodo y que sea sencilla en términos de complejidad computacional.

Debido a su funcionamiento, esta técnica **permite ahorrar muchos cálculos** durante la exploración exhaustiva de árbol, lo que la convierte en una buena elección para problemas de gran tamaño. Otra ventaja adicional de esta técnica es que permite el **tratamiento en paralelo**.



Capítulo 6

Descenso del gradiente

El método del **descenso del gradiente** se considera como uno de los más populares dentro de los algoritmos de optimización por su extensivo uso en el campo de la inteligencia artificial. Existen varias denominaciones, pero la palabra *gradiente* siempre está presente.



El concepto fundamental de gradiente no es más que la **generalización vectorial de la derivada de una función**. El gradiente es un vector de tantas dimensiones como variables tenga la función objetivo y, en cada dimensión, contiene la derivada parcial sobre cada una de las variables. Tal como ocurre con las funciones de una variable, el gradiente nos proporciona información de cuánto crece la función objetivo en un punto específico para cada una de las variables de manera independiente.

La analogía virtual para entender la estrategia del descenso de gradiente es el símil del **descenso de una montaña** si lo que queremos es alcanzar el punto más bajo (optimización para minimizar).

Una vez que conocemos la dirección que debemos tomar para mejorar la solución actual que nos proporciona el gradiente, la siguiente cuestión a resolver es el **tamaño del paso** para descender. Un primer criterio lógico consiste en elegir dar un paso grande si el descenso es muy pronunciado y un paso más corto si el descenso es pequeño. Este parámetro es conocido como tasa de aprendizaje (*learning rate* en inglés).

Finalmente, como en la mayoría de los algoritmos iterativos (este lo es), debemos establecer un **criterio de parada**. En este caso, una posibilidad es medir en cuánto se mejora de una solución a la siguiente. Si esta mejora no supera un cierto umbral que habremos establecido previamente, el proceso se detiene. Como siempre, también podemos establecer un criterio de parada una vez que hemos alcanzado un número de iteraciones.

Es necesario conocer algunas cuestiones importantes para aplicar esta técnica. En primer lugar, debemos conocer la **convexidad de la función objetivo**. Como ocurre en otros algoritmos, este comportamiento enlaza con la capacidad de asegurar o no que estamos ante un mínimo global o simplemente conformarse con un mínimo local.

En este punto, debemos repasar el concepto de **convexidad** y su relación con la optimización de funciones. De manera simplificada, se dice que una función es convexa si entre dos puntos cualesquiera de la función el segmento que los une está siempre dentro de la gráfica de la función.

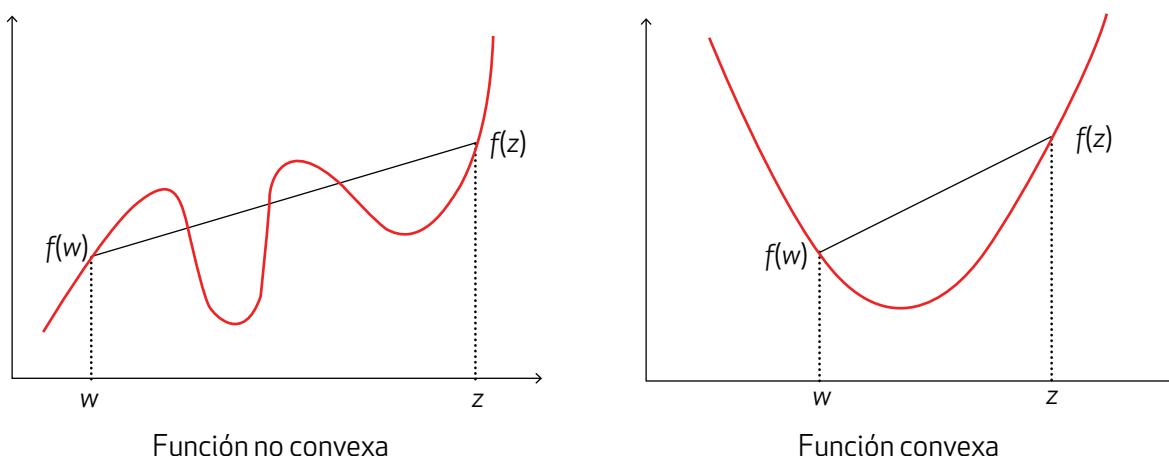


Figura 16. Representación de una función no convexa y una función convexa.

Aunque la expresión “**estar siempre dentro de la gráfica de la función**” no es muy rigurosa matemáticamente, sirve para una asimilación intuitiva. La consecuencia es que, si encontramos un mínimo en una función convexa, podemos asegurar que se trata de un mínimo global. Por el contrario, si la función no es convexa, no hay manera de confirmar que dicho mínimo sea el global que se busca. Esta situación, que es recurrente, como veremos, en las técnicas de optimización, se conoce como la capacidad para escapar de los mínimos locales. En general, un método intuitivo para lograr esta escapada es ejecutar el algoritmo sobre diferentes soluciones iniciales.

Un último apunte respecto al ámbito de la convexidad es el que se refiere a las **funciones cóncavas**. Dada una función cóncava (x), podemos considerar la función $-f(x)$ (con el signo menos delante) y la nueva función se convertirá en convexa. Por esta razón, incluiremos también a las funciones cóncavas en el mismo tratamiento. Así pues, diremos que son no convexas las funciones que no son convexas ni cóncavas.

Otra de las consideraciones importantes para utilizar el descenso del gradiente es, lógicamente, la condición de que la función objetivo sea **derivable**. A pesar de este inconveniente, la técnica del descenso del gradiente es de las más utilizadas en redes neuronales, aunque muchos de los problemas son discretos y, por tanto, ni continuos ni derivables.

Por último, es importante el **valor de la tasa de aprendizaje**. Aunque no siempre es posible, conocer la función objetivo ayuda a decidir la estrategia sobre qué valor asignar a este parámetro. Parece evidente que un valor constante para cada iteración no es la mejor decisión. Un valor muy pequeño puede alargar excesivamente el número de iteraciones y un valor muy alto puede provocar que no demos con la solución.

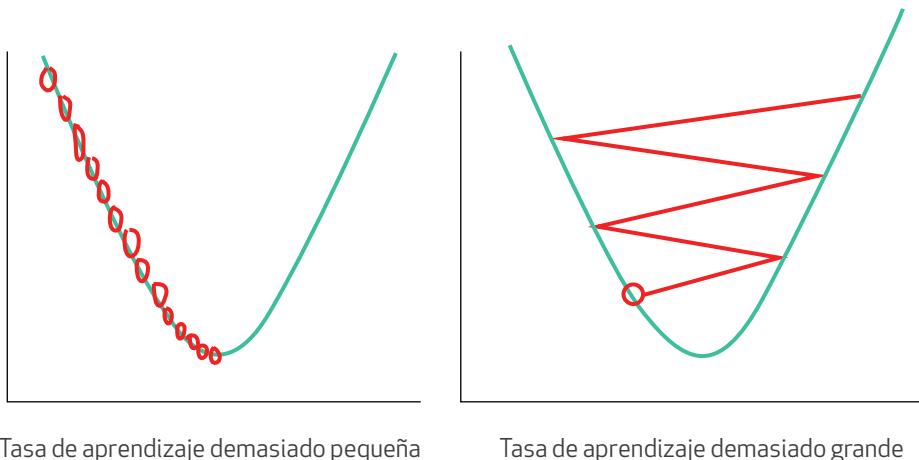


Figura 17. Inconvenientes de seleccionar una tasa de aprendizaje excesivamente pequeña y excesivamente grande en el descenso del gradiente.

Tampoco es una buena opción, para elegir la tasa de aprendizaje, tener en cuenta la **magnitud del descenso**, pues podemos encontrarnos cerca del punto de silla, en el que dicha magnitud es muy pequeña, lo cual estanca el proceso en soluciones que no interesan.

En general, la elección depende de cada problema y, por tanto, de cada función que se quiere optimizar. De todos modos, las estrategias más utilizadas consisten en **hacer más pequeño este parámetro de forma dinámica**, dependiendo tanto de la magnitud del gradiente como del número de iteraciones.

¿Pero qué implicación tiene el descenso del gradiente en la **inteligencia artificial** y, más en concreto, en el **aprendizaje automático** (*machine learning* en inglés)? La relación directa se da en los procesos de clasificación del ámbito del aprendizaje supervisado. En estos problemas, el objetivo es clasificar en diferentes categorías una serie de puntos en el espacio N -dimensional de tal manera que podamos encontrar una función que minimice el error de la clasificación.



La función que minimiza los errores del conjunto de datos etiquetados se llama **función de coste**. La función de coste habitualmente usada es la **función de coste cuadrático**. Esta función tiene su analogía en el método que utilizamos para el cálculo de la regresión lineal a través del error cuadrático medio. Si recordamos, este método trata de buscar la mejor aproximación lineal reduciendo el error al cuadrado medio de los valores conocidos. La función de coste ahora ya es una función continua y derivable. En concreto, es una función cuadrática en cada una de las variables del problema y, por tanto, es posible obtener el gradiente.

Existen otras posibilidades para la elección de la función de coste según como se quiera interpretar el concepto de **distanzia** dentro del problema. En el caso del error cuadrático medio, por ejemplo, minimizamos el error asociado a la **distanzia euclídea** de los objetos.

Sin embargo, podemos utilizar otros conceptos de distancia para obtener diferentes funciones de coste. Por ejemplo, en el caso de usar una **función de entropía cruzada** (*cross-entropy cost* en inglés), se consideran las soluciones como composiciones binarias. En este caso, la distancia entre dos soluciones es la cantidad de bits en los que se diferencian las dos soluciones que se comparan. Esta medida es una de las más aconsejadas cuando se trata de un problema de clasificación, pues en realidad mide cómo de diferentes son dos objetos. Es habitual usarla en varios ámbitos, como la clasificación de imágenes en algoritmos de aprendizaje automático.

Por otro lado, en **problemas médicos** es importante considerar de manera diferente y, por tanto, con distancias diferentes, los diagnósticos de falsos positivos y los de falsos negativos. En tal caso, debemos cuidar la elección para forzar este desequilibrio.

En definitiva, cada problema necesitará una **función de coste apropiada**. Además, debe cumplirse que sea una función continua y derivable para poder aplicar el gradiente. Es también deseable que la función de coste **no tenga "valles"**: en estas zonas el proceso es poco efectivo, dado que en esas zonas el gradiente varía, pero muy poco a poco.

Finalmente, concluimos el estudio de la técnica del descenso del gradiente con una **clasificación dependiendo del volumen de datos** que debemos tratar. En general, ante un volumen alto de datos, debemos balancear entre la precisión en los cálculos (y tener en cuenta cuantos más datos mejor) y la velocidad de ejecución del algoritmo.

En primer lugar, tenemos la opción de considerar todo el conjunto de datos, **descenso del gradiente por lotes** (*batch gradient descent*). En este caso estamos en el extremo de máxima precisión, pero de mayor tiempo de ejecución. Adicionalmente, tenemos la desventaja de la lentitud si queremos modificar el modelo con nuevos valores.

En cambio, con un **descenso del gradiente estocástico** (*stochastic gradient descent* en inglés), no consideramos el conjunto completo de datos, sino que vamos iterando sobre cada dato para realizar los cálculos. Existe una ventaja en cuanto al tiempo de procesamiento, pues los cálculos sobre elementos individuales son más rápidos. En contra, vemos que la incorporación de datos muy distintos de los que ya han sido procesados puede provocar movimientos muy sinuosos durante las primeras etapas. Para evitar estas situaciones, es aconsejable realizar un tratamiento secuencial lo más aleatorio posible de los datos.

El algoritmo con descenso del gradiente por lotes reducido (*mini-batch gradient descent* en inglés) toma las ventajas de ambas estrategias para realizar tratamientos en bloques de los datos.



Capítulo 7

Métodos heurísticos y metaheurísticos

Durante los capítulos anteriores hemos analizado algunas técnicas y algoritmos que nos permiten resolver problemas de optimización. También hemos visto algunos casos en los **no es posible encontrar un algoritmo realmente eficiente** que nos lleve a la solución óptima. Esto puede deberse al hecho de que no existe un algoritmo conocido que se ajuste al problema o, simplemente, que los algoritmos o técnicas que conocemos son ineficientes para nuestro propósito (Hillier y Lieberman, 2015).

En estos casos, debemos rebajar nuestras pretensiones y conformarnos con encontrar **buenas soluciones**, aunque no podamos asegurar que quizás hayamos encontrado el óptimo.



Los métodos heurísticos pretenden encontrar **buenas soluciones**. No podemos conocer la calidad que proporcionan, pero la solución puede estar cerca de la óptima si realizamos un buen diseño (Duarte, 2008). Una de las características de estos problemas es que son grandes en términos de datos. Si no fuera así, muy probablemente podríamos resolverlos con algunas de las técnicas de búsqueda y exploración de árboles que ya hemos estudiado.

El **diseño** de estos algoritmos se basa en realizar iteraciones cada una de las cuales supone la búsqueda de una solución que mejore la que hemos obtenido en iteraciones anteriores.

Otra de las características importantes de estos algoritmos es la forma en que buscamos nuevas soluciones que mejoren la que ya tenemos. Suelen ser **ideas simples y de sentido común**, aplicables desde el punto de vista computacional, las que permitan diseñar el proceso de búsqueda.

La propia naturaleza tan específica de estos problemas provoca que los algoritmos que los resuelven sean también **muy particulares** y no puedan trasladarse a otros problemas. Esto conlleva, en general, mayor esfuerzo de cara a su diseño, implementación y aplicación.

A pesar de esta limitación, existen métodos generales que proporcionan tanto una estructura general como criterios estratégicos para encontrar un método heurístico específico para el problema. Se trata de los **métodos metaheurísticos**, que se han convertido en los últimos tiempos en un ámbito de estudio destacado y orientado a los problemas asociados a la inteligencia artificial.



El funcionamiento general de las técnicas metaheurísticas consiste en mantener almacenado durante el proceso una **solución o estado de referencia**, que es el mejor encontrado hasta el momento a través de las iteraciones. Se puede proporcionar el punto inicial como punto de entrada (en general, es una solución factible que ya conocemos) o puede ser el propio proceso el que la genere. Esta solución de referencia se modifica ligeramente para compararla con la anterior mediante la función de evaluación (que suele ser la función que deseamos optimizar). En caso de mejorarla, se convierte en la nueva solución de referencia.

La manera en la que realicemos las **ligejas modificaciones** a nuestro estado de referencia da nombre a los métodos metaheurísticos. Estas modificaciones y como se realicen llevarán en gran medida al éxito o no del proceso para obtener una buena solución. Algunos métodos no almacenan un solo estado, sino que mantienen vigentes varios estados, eliminando en cada iteración los peores y conservando los mejores. La generación de nuevos estados se basa en algún tipo de combinación o cruce entre los vigentes, como veremos en los algoritmos genéticos.

La mayoría de estos métodos **se inspiran en procesos de la naturaleza**, que ha generado mecanismos de optimización para determinadas situaciones concretas.

Dado que la utilización de estos métodos está orientada a problemas con un número elevado de soluciones o estados posibles, suelen incorporarse **mecanismos forzados de interrupción del proceso** para evitar tiempos excesivos de ejecución o bucles no esperados en el diseño. De esta manera, dado un cierto número de iteraciones, el proceso devuelve la mejor solución obtenida hasta ese momento.

Una de las cuestiones que más deben preocupar al considerar estos métodos se refiere a lo que se denomina “**escapar de un óptimo local**”. Podemos disponer de un algoritmo eficiente que consiga una solución rápida al problema. Sin embargo, en la mayoría de los casos, con la utilización de los métodos heurísticos no sabremos si estamos ante un óptimo local o el óptimo global. Para resolver esta situación, se consideran algunas alternativas. Una de ellas consiste en incorporar un mecanismo en el proceso que permita eventualmente escapar de la inevitable convergencia a un mínimo local. Otra alternativa, combinada con la anterior, y no siempre disponible, puede ser la ejecución del algoritmo con soluciones o estados iniciales diferentes con el propósito de disponer de varias alternativas para la solución.

A continuación, veremos algunos métodos metaheurísticos.

7.1. Búsqueda tabú

El término **tabú** proviene del ámbito sociocultural, en el que se establecen una serie de prohibiciones sobre algunos asuntos, las cuales suelen superarse y desaparecer con el tiempo.

La **búsqueda tabú** (*tabu search* en inglés) se basa en ideas sencillas y de sentido común para que el proceso de búsqueda de la solución óptima pueda escapar de óptimos locales (Duarte, 2008). Para ello, se parte de una solución factible y se establece un procedimiento de **búsqueda local en la vecindad** de dicha solución para determinar nuevas soluciones factibles. Debemos establecer la definición de vecindad a partir de una solución dada a través de un procedimiento o función que proporcione otras soluciones efectuando movimientos sobre los elementos de la solución dada.

Las soluciones locales (o determinados elementos que las componen o determinados movimientos entre soluciones) se guardan en una lista tabú para evitarlas en las siguientes iteraciones. El propósito de la **lista tabú** es evitar volver a soluciones encontradas recientemente para evitar los ciclos en el proceso. La mejor solución hasta el momento se guarda para ser considerada la solución final. Se establece de antemano un tamaño en la lista tabú y se van eliminando las soluciones más antiguas (cola FIFO) con el propósito de dar más flexibilidad al proceso, suponiendo que las nuevas soluciones están ya suficientemente lejos en términos de vecindad como para que sean consideradas nuevamente.

El **tamaño de la lista tabú** es el que controla la memoria del proceso, de tal manera que, para tamaños pequeños, el proceso se concentra en espacios de soluciones pequeños. Sin embargo, para tamaños grandes de la lista, el proceso explora espacios más grandes de soluciones.

Una **mejora avanzada** introduce un tamaño variable de la lista a lo largo del proceso dependiendo de determinadas circunstancias del problema. Esta mejora es conocida como **búsqueda tabú reactiva** y permite al algoritmo ser más robusto.

De igual modo, también para fortalecer el algoritmo, el estatus tabú puede ser **relajado o no considerado bajo ciertas circunstancias** para evitar, por ejemplo, despreciar una solución vecina clasificada como tabú que sea la mejor encontrada hasta el momento. Esta estrategia generaría una ejecución sin fin del proceso. Por lo tanto, es necesario introducir un **criterio de detención** que lo evite. Dependiendo del problema y de las circunstancias para resolver la detención, estos criterios pueden estar relacionados con el de un número fijo de iteraciones realizadas, una cantidad de tiempo o, el más habitual, un número de iteraciones en las que no se ha podido obtener una solución mejor a todas las encontradas.



Así pues, los principios esenciales de esta técnica son:

- **Memoria adaptativa** del proceso, que se actualiza en función del paso de las iteraciones y del análisis de la vecindad.
- **Búsqueda sensible**, para determinar movimientos acerca de las soluciones de acuerdo con lo ocurrido con anterioridad.

Puede ser habitual considerar algunas incorporaciones para mejorar la eficiencia del proceso. Una de ellas es la **intensificación**, que supone la exploración de algunas soluciones con mayor intensidad que el resto con la esperanza de que sean más prometedoras. Esta decisión se toma como consecuencia de algún análisis previo que permita inducir que estamos en lo cierto. Otra incorporación a considerar es la **diversificación**, que implica forzar la exploración de soluciones locales en áreas de soluciones que no han sido exploradas.

Estos dos conceptos, la intensificación y la diversificación, se usan de manera general en el ámbito del diseño de algoritmos. Se abordará su aplicación en el Capítulo 8, sobre algoritmos genéticos y evolutivos.

Como ocurre con el resto de las técnicas metaheurísticas, la búsqueda tabú proporciona una **estrategia muy general** para abordar los problemas. En cada caso concreto, debemos realizar un esfuerzo para identificar, modelar y diseñar los elementos que van a componer el algoritmo según la metaheurística escogida.

Para **diseñar un algoritmo de tipo búsqueda tabú**, hay que tener en cuenta las siguientes consideraciones:

1. Establecer el proceso de búsqueda local.
2. Establecer la estructura de vecindad o la forma de encontrar soluciones vecinas a una dada.
3. Establecer la estructura tabú o la forma como se representan los elementos de la lista tabú.
4. Establecer el proceso para elegir y añadir un elemento a la lista tabú.
5. Establecer el tamaño de la lista tabú.
6. Establecer la regla de detención.

El **problema del viajante para N ciudades** permite ilustrar un ejemplo de diseño de un algoritmo usando la técnica de búsqueda tabú. Partimos de una solución cualquiera elegida al azar. La estructura de soluciones la encontramos de manera sencilla solo con definir las permutaciones de $\{1, 2, \dots, N\}$.

Definimos la **vecindad** de tal manera que dos soluciones son vecinas si han intercambiado dos ciudades. Por ejemplo, las soluciones $\{1, 2, 3, 4\}$ y $\{2, 1, 3, 4\}$ son vecinas. En cuanto a la **lista tabú**, está compuesta por los movimientos que realicemos de intercambio. Por tanto, debemos definir una estructura que guarde las ciudades que han sido intercambiadas y las posiciones iniciales y finales que han ocupado: $[i, j, Pos(i), Pos(j)]$. En el caso anterior, guardaremos en la lista tabú $[1, 2, 2, 1]$. Debemos considerar que $[j, i, Pos(j), Pos(i)]$ es el mismo movimiento que $[i, j, Pos(i), Pos(j)]$ y, por tanto, también considerarlo tabú si llega el caso.

Para el **proceso de búsqueda local** podemos establecer, por ejemplo, la realización de 20 movimientos para determinar, entre todas las soluciones que los proporcionan, la mejor teniendo en cuenta las restricciones tabú. Esa solución se considera la mejor solución local y se incluye en la lista tabú comparada con la mejor solución encontrada hasta el momento para considerarla o no candidata a solución final.

Este diseño es una de las múltiples posibilidades que se pueden plantear. Podríamos definir **criterios diferentes de vecindad** más o menos restrictivos y disminuir o ampliar el tamaño de la lista tabú o introducir criterios de intensificación o diversificación.

7.2. Recocido simulado

El **recocido simulado**, **enfriamiento simulado** o **SA** (*simulated annealing* en inglés) describe la metaheurística basada en el proceso termodinámico conocido como recocido. En este proceso de tratamiento de metales se eleva la temperatura hasta cierto valor. El proceso posterior de enfriamiento se realiza lentamente y sujeto a controles para que se ajuste a una determinada función con el objetivo de conseguir una estructura óptima para la cristalización del metal de acuerdo con criterios de dureza, durabilidad, pureza, etc. (Duarte, 2008; Brownlee, 2011).

Estos procesos dinámicos se basan en una **distribución de probabilidad** (Maxwell- Boltzmann) que se asocia a los niveles energéticos de las partículas dependiendo de la temperatura. A mayor temperatura, mayor probabilidad de encontrarse en estados de energía mayores y, por tanto, mayor inestabilidad respecto a la temperatura inicial. Esta inestabilidad puede permitir modificar la estructura de la materia durante el enfriamiento posterior.



La adaptación de este proceso al ámbito de los algoritmos (Metropolis, Rosenbluth, Rosenbluth, Teller y Teller, 1953) se fundamenta en la **búsqueda local de soluciones** para las que es posible escapar de óptimos locales permitiendo que, bajo ciertas circunstancias, se consideren movimientos que empeoren la solución encontrada hasta el momento.

Aplicando el principio físico de las partículas, un algoritmo, según esta técnica, comienza con la búsqueda de la solución más estable de acuerdo con una temperatura elevada, que, en términos computacionales, se traduce por la elección de **peores soluciones con una probabilidad alta**. Durante las primeras interacciones, no es relevante que las soluciones no sean buenas, dado que se supone que no estamos cerca de una solución óptima. A medida que el proceso avanza en el tiempo (la temperatura disminuye), la probabilidad de elegir peores soluciones disminuye. A diferencia de la búsqueda tabú, esta técnica introduce un componente aleatorio con la esperanza de encontrar mejores soluciones en otras regiones y proporcionar más flexibilidad.

La implementación de un algoritmo se basa en la búsqueda de soluciones por vecindad, como vimos en el algoritmo de búsqueda tabú, pero en este caso se va ajustando a lo largo de las iteraciones según la llamada **variable temperatura**. Esta variable se inicializa con un valor alto y se va reduciendo en cada iteración hasta llegar a un valor final para el que el proceso finaliza.

La variable temperatura determina en qué medida (según una probabilidad) se aceptan soluciones peores a la actual. En caso de encontrar una solución mejor, siempre se acepta. Pero, si es peor, cabe la posibilidad de que sea aceptada según el criterio de la función de probabilidad y el valor de la temperatura. Esta estrategia permite incorporar buenas soluciones a medida que se encuentran y, a la vez, dar alguna posibilidad al proceso de escapar de óptimos locales.

La probabilidad para determinar la selección de una solución viene dada por una función:

$$F(\delta, T) = \exp(-\delta/T)$$

En la ecuación anterior:

- δ representa la diferencia entre los valores aplicados a la solución actual y la solución vecina a considerar.
- T representa la variable temperatura, que disminuye con el tiempo.

De lo anterior se deduce que, a mayor temperatura, mayor probabilidad de elegir una solución peor para permitir al principio escapar de soluciones óptimas locales; a menor diferencia del valor de las soluciones, mayor probabilidad de aceptar soluciones peores.

La Figura 18 ilustra cómo, en determinados casos en los que existen mínimos locales, el proceso puede conducirnos hacia el mínimo local si el punto de partida se encuentra más cerca de un mínimo local que del mínimo global:

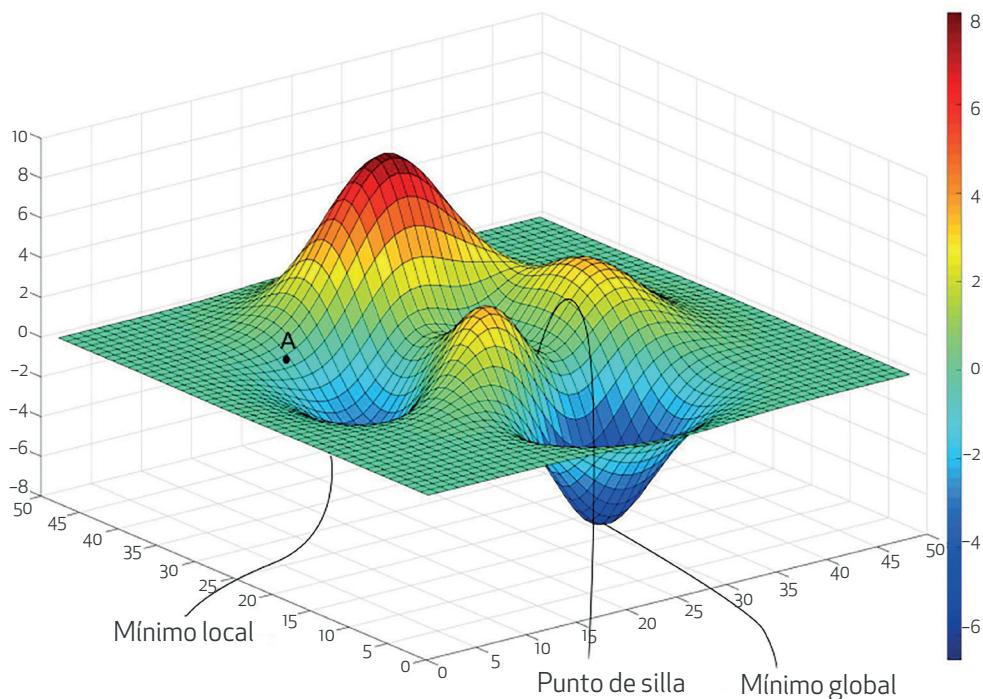


Figura 18. Representación en 3D de una función con un mínimo local, un mínimo global y punto de silla.

Un aspecto importante, como ocurre en el ámbito físico, es establecer la función que determina el **descenso de la variable temperatura**. Existen varias formas de descenso:

- Constante.
- Fijado de inicio con tramos diferentes dependiendo de la iteración.
- Exponencial.
- Inverso al número de iteraciones (criterio de Boltzmann): $T_k = T_0/(1 + k)$.
- logaritmo inverso al número de iteraciones (criterio de Cauchy): $T_k = T_0/(1 + \log(k))$.



Ejemplo

La **implementación** a alto nivel para el problema del viajero consta de las siguientes fases:

1. Seleccionar un recorrido inicial aleatorio y establecer la función que controla la temperatura del proceso.
2. Seleccionar una solución vecina. Una posibilidad es intercambiar el orden de dos de las ciudades o nodos.
3. Comparar la nueva solución con la mejor obtenida hasta entonces. Si es mejor, marcarla como la mejor encontrada hasta el momento. Si no es la mejor, es posible aceptarla o no de acuerdo a la probabilidad determinada por la función que controla la temperatura del proceso.
4. Volver al paso 2 hasta que la temperatura alcance un valor preestablecido.

Como en el caso de la búsqueda tabú, la combinación de diferentes valores para los parámetros iniciales, la elección de las distribuciones de probabilidad, los criterios de definición de vecindad y la combinación con otras técnicas determinan algoritmos completamente diferentes, que hay que ajustar para conseguir los mejores resultados posibles.

7.3. Optimización por enjambre de partículas

En torno al concepto de **inteligencia de enjambre** (*swarm intelligence* en inglés), se engloban un conjunto de técnicas inspiradas en la forma que tienen de actuar un grupo de individuos o partículas con conductas poco sofisticadas, pero con un comportamiento colectivo (movimientos sincronizados o estrategias de defensa ante depredadores) que se asemeja al de un individuo complejo. Sobre todo en la naturaleza, se observan sociedades como bancos de peces, bandadas de pájaros o conjuntos de bacterias u hormigas que se relacionan para obtener un bien común sin la existencia de un control superior centralizado (Duarte, 2008).

Desde el punto de vista computacional, usamos también el concepto de enjambre para referirnos a un **conjunto de agentes que usan reglas sencillas relativamente independientes al resto** (solo se relacionan localmente) para conseguir un objetivo determinado. La implementación consiste en determinar un conjunto de soluciones (puede ser de modo aleatorio) que adoptan el rol de partículas y que adquieren movimiento según unas reglas que tienen en cuenta tanto el conocimiento individual como el conocimiento global de todo el conjunto de partículas.

Basados en este concepto de inteligencia de partículas, existen los algoritmos orientados a **optimización por enjambre de partículas** o **PSO** (*particle swarm optimization* en inglés). Estos algoritmos se caracterizan por que cada partícula tiene asociados un **vector posición** $p[]$ y un **vector velocidad** $v[]$. Como partículas en movimiento, adquieren una **inercia** y una **aceleración** que vienen determinadas por dos características para cada partícula:

- Es atraída por la mejor localización que ha podido encontrar **individualmente** (particular). La llamaremos $pbest[]$.
- Es atraída por la mejor localización obtenida por el conjunto de **todas las partículas** (global). La llamaremos $gbest[]$.

La fuerza (y, por tanto, la velocidad y la aceleración) a la que se somete a estas partículas depende de ciertos parámetros tanto particulares como globales, de manera que, cuanto más se alejen de las mejores localizaciones, mayor es la fuerza de atracción hacia ellas.

En cada iteración y para cada partícula, se obtiene el nuevo **vector velocidad** con la siguiente fórmula:

$$v_{nueva}[] = v[] + c_1 \times \text{rand}() \times (pbest[] - p[]) + c_2 \times \text{rand}() \times (gbest[] - p[])$$

En la fórmula anterior:

- c_1 y c_2 son constantes asociadas a factores de aprendizaje que hay que establecer al inicio.
- $gbest$ es la mejor posición global.
- $pbest$ es la mejor posición particular de cada partícula.
- $p[]$ es la posición actual de cada partícula.
- $\text{rand}()$ es un numero aleatorio entre 0 y 1.
- $v[]$ es la velocidad

Por tanto, la **nueva posición de cada partícula** viene determinada por esta sencilla fórmula:

$$p_{nueva}[] = p[] + v_{nueva}[]$$

Algunas variantes sugieren limitar la velocidad máxima a la que puede desplazarse una partícula para evitar inconsistencias.

En cuanto los **valores c_1 y c_2** , llamados factores de aprendizaje, pueden variar en el rango [0, 4], siendo 2 el valor más habitual. Permiten controlar o ponderar el comportamiento particular y social respectivamente de las partículas.

Como en otros algoritmos, hay que incorporar un **criterio de parada**. En general, se establece según en el número de iteraciones si no se dispone de otro, como una medida de error respecto a la distancia del óptimo. También podemos establecer parada si, después de un número determinado de iteraciones, no se mejora la solución global.

Como sucede en todas las técnicas metaheurísticas, las propuestas iniciales se amplían con variantes para obtener mejores resultados atendiendo al problema particular. Una de ellas consiste en incorporar el concepto de **masa para las partículas** con el fin de influir de manera especial sobre las velocidades de unas partículas sobre otras (Shi y Eberhart, 1998.). Otra variante sencilla de implementar es la de introducir una **velocidad inicial a las partículas** para que no puedan acercarse a una solución local demasiado deprisa.



Enlace de interés

En el vídeo del siguiente enlace puede verse el comportamiento que van teniendo las soluciones para acercarse a la solución óptima:

<https://www.youtube.com/watch?v=OUHAyPWN1Ron>

Las ventajas de este modelo es que solo es necesario realizar **operaciones vectoriales básicas** en cada iteración, que se verán multiplicadas por la cantidad de partículas que queramos introducir en el proceso. En contra, tenemos la necesidad de evaluar en cada iteración la **función de evaluación** para cada uno de los movimientos que puede no ser trivial dependiendo del problema. Además, como ocurre general en las técnicas heurísticas, **no asegura obtener el óptimo global**, aunque con la experiencia en la configuración del número de partículas y el resto de los parámetros podremos encontrarlo en muchas situaciones.

En cuanto a los ámbitos de aplicación, los problemas con un **espacio de soluciones grande y variado** son susceptibles de ser tratados con técnicas basadas en inteligencia de enjambre. Otra área interesante de aplicación son los **problemas multiobjetivo** en los que se persigue que las partículas finalicen su trayectoria en el subconjunto de Pareto (subconjunto de soluciones que no son dominadas por otras para ninguno de los criterios a optimizar).

Dado que los cálculos que se realizan están asociados a valores continuos, por la inspiración cinética de este tipo de algoritmos, podría pensarse que solo se destinan a la resolución de problemas de optimización con **funciones objetivo continuas**. Aunque es cierto que son muy apropiados para este tipo de problemas, podemos resolver también **problemas discretos** mapeando el espacio discreto de soluciones a un espacio continuo y, una vez obtenida la solución, realizando el proceso inverso. Aunque será posible para algunos casos, en otros esta transformación puede resultar difícil o imposible (Clerc, 2004, pp. 219-239).

7.4. Procedimientos de búsqueda voraz aleatorios y adaptativos

Los **procedimientos de búsqueda voraz aleatorios y adaptativos** o **GRASP** (*greedy randomized adaptative search procedures* en inglés) fueron desarrollados por Feo y Resende para resolver problemas de recubrimiento de conjuntos (Feo y Resende, 1995).

El principio de este tipo de metaheurísticas es realizar **iteraciones** (procedimiento multiarranque) de **construcción y mejora** para encontrar una buena solución. Durante la fase de construcción, se va construyendo por etapas una solución factible añadiendo elementos a través de una función voraz que analiza cada uno de los elementos para elegir aleatoriamente uno de entre los mejores (lista restringida de candidatos o RCL, *restricted candidate list* en inglés). La naturaleza voraz de esta estrategia ya la analizamos anteriormente y observamos que la mejor elección en un paso intermedio podría no ser la mejor de en el futuro. Esto hace que algunos autores hablen de estrategia **ciega** en lugar de voraz.

El **componente aleatorio** de la elección permite diversificar las soluciones y no realizar construcciones repetidas. Puesto que en la fase inicial no se ha tomado realmente la mejor opción, sino que se ha elegido aleatoriamente entre un grupo de las mejores, se incorpora una segunda fase de mejora local para, ahora sí, obtener la mejor solución posible entre las más cercanas (mejora local).

El proceso vuelve a reiniciarse para obtener una nueva solución. En caso de que mejore a las anteriores, se anota para devolverla al final tras cumplirse el criterio de parada.

El **tamaño de la lista de candidatos** resulta fundamental en cada problema, pues los tamaños grandes pueden permitir introducir elementos no muy buenos *a priori* pero que pueden formar parte o no de mejores soluciones al final del proceso. Y, por el contrario, las listas pequeñas de candidatos reducen la aleatoriedad y hacen perder oportunidades. Por tanto, un análisis para la configuración de este tamaño puede ser determinante.

Podemos incorporar diferentes **variantes** que nos permitan adaptarnos a cada problema concreto. Una de ellas es el **GRASP reactivo**, que supone utilizar diferentes tamaños para, a medida que generamos soluciones, quedarnos con los que mejores resultados dan. Otra variante puede ser no considerar igual de probables todos los elementos de la lista de candidatos, sino incorporar una **función de distribución** sobre ellos de acuerdo a la calidad que aporten en la solución.



Capítulo 8

Algoritmos evolutivos y genéticos

Dentro de las técnicas metaheurísticas analizadas en el apartado anterior, un grupo de ellas se caracterizan por emplear un **conjunto de soluciones**, habitualmente llamado población.

Estas técnicas proporcionan mecanismos de exploración del espacio de soluciones y su efectividad depende de las reglas de manipulación que incorporemos a dicha población.

En general, las **técnicas genéticas y evolutivas** son utilizadas cuando el espacio de soluciones es grande y tanto este como la función objetivo no presentan un comportamiento lineal.

Históricamente, los primeros estudios sobre algoritmos evolutivos se remontan a los años cincuenta, cuando se propusieron las primeras técnicas para resolver problemas combinatorios sencillos.

La idea fundamental se basa en la **teoría darwiniana sobre las especies**, en la que se establece que los individuos mejor adaptados al medio tienen mayor probabilidad de vivir más tiempo y, por tanto, mayor probabilidad de generar descendencia que conserve sus características.

Al contrario, los individuos peor adaptados, con menor probabilidad de sobrevivir, tienen menos opciones de generar descendencia y probablemente se extinguirán.



Desde el punto de vista computacional, la **población** es un conjunto de soluciones candidatas para resolver el problema. La **selección** es el mecanismo por el que el proceso selecciona con mayor probabilidad las mejores soluciones para trasmitir su contenido genético. Los nuevos **individuos** o soluciones se generan en cada iteración mediante modificación aleatoria. Los diferentes tipos de algoritmos genéticos se producen por la alteración individual (mutación), por combinaciones entre individuos (cruce) o, como ocurre en realidad en la naturaleza, por combinación de ambas formas a la vez.

La implementación de estas reglas de la naturaleza da lugar a diferentes tipos de algoritmos.

8.1. Algoritmos evolutivos

Habitualmente, en los **algoritmos evolutivos** no se contempla el cruce entre soluciones. Se usan en problemas en los que se puede predecir de cierta manera el comportamiento de la mejora de las soluciones. La implementación se basa en generar una población inicial de soluciones y realizar una mutación sobre ellas. Sobre los individuos iniciales y los mutados, se aplica el proceso de selección, para continuar en la siguiente iteración con el mismo esquema e iterar hasta que se cumpla el criterio de parada (Duarte, 2008).

Es importante tener en cuenta que el **proceso de selección** no tiene por qué estar asociado directamente a la función objetivo de problema. Es posible que, como ocurre en la naturaleza, un individuo que no sea el mejor adaptado transmita un componente genético mejorable en generaciones sucesivas. En otras ocasiones, incluso es costoso aplicar la función objetivo o, incluso, puede no tomar valores numéricos. Por ello debemos introducir una **función de evaluación** (*fitness* en inglés), que determina la calidad de los individuos para compararlos.

Detrás de esta consideración se encuentra la posibilidad de **diversificar el ámbito de búsqueda** de soluciones que hemos visto en otras técnicas. Podemos incorporar en cada generación o iteración la conservación del mejor individuo (o los mejores individuos). Hay que tener en cuenta que, a medida que avanza el proceso, estas serán las soluciones más próximas a la que buscamos. Esta estrategia se llama **elitismo**. El equilibrio entre el elitismo y el hecho de permitir la selección de individuos que no son los mejores permite establecer la relación entre la intensificación o diversificación en la búsqueda de soluciones que hemos visto en otras técnicas.

8.2. Algoritmos genéticos

Debemos la incorporación de los **algoritmos genéticos** a John Henry Holland, en la década de los setenta. A diferencia de los anteriores, en los algoritmos genéticos se da importancia a la combinación de soluciones (cruce) y, además, se incorpora la mutación y la selección.

En este caso, existe una dificultad adicional que hay que resolver. Se trata de encontrar la **modelización adecuada** para representar correctamente el problema en términos biológicos (Duarte, 2008). Un pequeño repaso de biología nos ayudará a comprender esta dificultad.

El genotipo es el conjunto de genes de un individuo y el fenotipo está formado por los rasgos observables que vienen determinados por el genotipo. En nuestro ámbito de computación, debemos encontrar la representación a través de las variables del algoritmo, el **genotipo**, de las soluciones del problema, el **fenotipo**. Esta tarea se llama **representación** y permite definir en el siguiente paso los operadores genéticos que darán lugar a las mutaciones y los cruces entre soluciones. La representación debe tener en cuenta que los genotipos están compuestos por genes, que serán los que finalmente intervendrán en los operadores genéticos. En general, la representación utilizada es a través de cadenas binarias, por ser la más intuitiva, aunque también pueden utilizarse valores numéricos enteros y reales.

En cuanto a la **función de selección** para conservar o no determinados individuos de una generación a otra, existe una variedad importante de técnicas que nos permitirán definirla según la naturaleza del problema.

En general, la dificultad para aplicar los algoritmos genéticos la encontramos a la hora de establecer la **representación genómica de las soluciones** y la posterior **definición de las funciones de cruce y mutación**.



Ejemplo

A modo de ejemplo y sin llegar a diseñar completamente el algoritmo, trataremos de indicar algunas pautas para resolver el **problema del viajante a través de un algoritmo genético**. En este caso simplificado, el problema se compone de 7 ciudades. Para mayor claridad, identificaremos las ciudades con letras de la A a la G y supondremos que todas las ciudades están conectadas entre sí.

Así pues, la combinación $s_1 = (A, B, C, D, E, F, G)$ y $s_2 = (A, C, E, G, B, D, F)$ son soluciones al problema que establecen el orden entre ciudades en cada caso. Podríamos establecer una función de cruce de la siguiente manera: elegimos las 3 primeras ciudades de alguno de los padres y las 4 restantes del otro padre manteniendo el orden. De esta manera, un posible hijo de s_1 y s_2 puede ser $s_3 = (A, B, C, E, G, D, F)$. Se han tomado las 3 primeras ciudades de s_1 , {A, B, C} y las otras 4 de s_2 , [E, G, D, F], que son las que faltan en s_2 y por ese orden.

Lógicamente, elegimos este procedimiento porque de otra **manera repetiríamos ciudades y dejaríamos otras fuera** en caso de una mezcla incontrolada. En todos los problemas nos encontraremos con situaciones similares a esta en el diseño. Para el caso de las mutaciones dentro de una solución, podemos establecer la siguiente regla. Una mutación, en general, consistirá en una permutación de uno o varios elementos de la solución con otros. Por ejemplo, una mutación para el individuo s_1 podría consistir en intercambiar la segunda posición por la quinta, de modo que el nuevo individuo tendría el siguiente genotipo: (A, **E**, C, D, **B**, F, G).

Como se puede anticipar, las posibilidades para establecer las reglas genéticas son bastante grandes, por lo que la efectividad de este algoritmo está ligada a nuestra capacidad para entender el problema y encontrar las reglas más adecuadas en cada caso.

Esta variedad de posibilidades, junto con la capacidad para resolver problemas de gran tamaño, hacen que los algoritmos genéticos y evolutivos sean actualmente una de las técnicas más utilizadas y una de las áreas de mayor interés para la comunidad científica.

La mayor parte de los algoritmos y técnicas que hemos revisado están orientados a resolver problemas estacionarios. En estos problemas, las circunstancias del problema no cambian con el tiempo. Sin embargo, en la práctica, existen **problemas dinámicos**, que sí cambian con el tiempo. Para abordarlos, es preciso utilizar técnicas que permitan **adaptar las soluciones a las nuevas circunstancias**. Una de las ventajas de los algoritmos evolutivos y genéticos es que podemos incorporarles algunas mejoras para que tengan en cuenta estos cambios.

La estrategia más adecuada depende de un análisis previo acerca de **cómo se producen los cambios** en cada caso. De esta manera, si observamos que los cambios se producen de manera periódica y son completamente aleatorios, podemos hacer que el proceso incremente la diversidad de las soluciones solo en los momentos posteriores al cambio. Según cómo de grandes sean estos cambios, podemos incorporar la nueva población con un índice de diversidad mayor o menor. A mayores cambios, es posible que necesitemos mayor diversidad. Es posible que los cambios puedan afectar incluso al genotipo de las soluciones, para lo cual debemos tener en cuenta reconsiderar la tarea de representación que vimos anteriormente.

8.3. Optimización por colonia de hormigas

Otra de las técnicas basadas en observaciones de la naturaleza es la **optimización por colonia de hormigas** o **ACO** (*ant colony optimization* en inglés), que se basa en el comportamiento de los individuos que persiguen un fin colectivo. Las colonias de hormigas, por ejemplo, deben encontrar comida en un entorno y llevarla a su hormiguero. Este entorno a veces cambia y, a pesar de ello, las hormigas se adaptan y vuelven a encontrar el mejor camino para volver a llevar la comida al hormiguero. Este comportamiento es, por tanto, dinámico y permite a las hormigas adaptarse a los cambios a lo largo del tiempo (Duarte, 2008).

El mecanismo que rige el comportamiento de las hormigas está basado en la **permanencia en el ambiente de las feromonas** que van emitiendo. Las feromonas permiten a las hormigas comunicarse para indicar a otras que se ha encontrado alimento y cómo llegar a él. Un rastro de feromonas permanece en el entorno un tiempo determinado y es más fuerte mientras se vea reforzado por otras hormigas y no pase mucho tiempo. El comportamiento aleatorio de cada hormiga se ve, por tanto, influenciado por la fuerza de este rastro, de tal manera que, a mayor fuerza, más probable es que la hormiga siga el rastro. Y, al contrario, a medida que el rastro desaparece con el tiempo, menor es la probabilidad de seguir el antiguo rastro y, por tanto, la hormiga explora otros espacios aleatoriamente.

Este **componente aleatorio** es el que permite a las hormigas en conjunto encontrar el mejor camino, pues, aunque en un primer momento el camino encontrado no sea el mejor, es posible que alguna mejor ruta posterior sea encontrada. En ese caso, puesto que el camino de vuelta es menor, el nuevo rastro va tomando mayor fuerza que el inicial y de esta manera pasa a convertirse en la ruta principal (Luke, 2005).

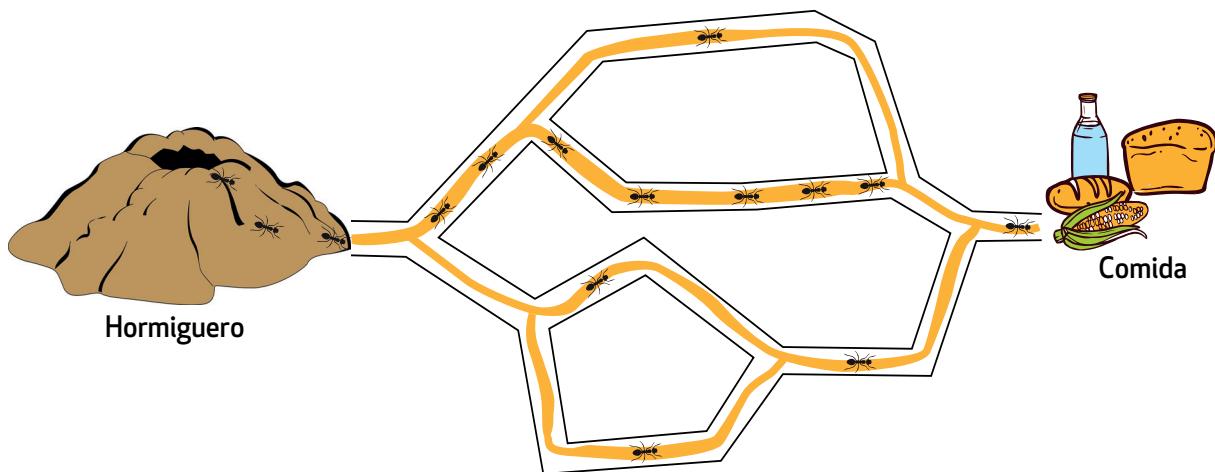


Figura 19. Esquema que ilustra que el camino más corto es elegido con mayor probabilidad debido a la fuerza de las feromonas.

Este esquema inspiró a Marco Dorigo en la década de los noventa a solucionar el problema de encontrar el camino mínimo de un grafo, aunque con el tiempo se ha ido adaptando para resolver otro tipo de problemas.

La adaptación de este comportamiento para diseñar algoritmos se basa en considerar cada solución factible como un rastro de feromonas depositadas por algún individuo. Cada individuo añade su parte de feromona a cada una de las partes de la solución y contribuye de esta manera a aumentar la fuerza del rastro. En cada iteración, el rastro se reduce, tal como actuaría el paso del tiempo sobre las feromonas en la naturaleza. A medida que transcurren las iteraciones, se observan rastros más fuertes, que son los que compondrán las mejores soluciones.

Glosario



Algoritmia

Es la disciplina que estudia el cálculo aritmético, algebraico y los algoritmos.

Árbol de expansión

Dado un grafo conexo, un árbol de expansión es un subconjunto de nodos y aristas el cual contiene todos los nodos y en el cual las aristas no forman ciclos.

Árbol de recubrimiento mínimo

Un árbol de recubrimiento mínimo es un árbol de expansión cuyas aristas consideradas suman un valor mínimo para sus costes asociados.

Distancia euclídea

Es la distancia que usamos habitualmente en la vida cotidiana. En un sistema de coordenadas cartesianas, se basa en el teorema de Pitágoras para calcular la distancia entre dos puntos del espacio n -dimensional, que reconocemos como el tamaño del segmento que los une. Existen otras distancias basadas en otros criterios, como la hiperbólica o la elíptica, entre otras.

Ecuación en recurrencia

Se trata de una ecuación asociada a una secuencia o sucesión en la que cualquier término depende de los anteriores. El caso habitual que se pone como ejemplo es la sucesión de Fibonacci, en la que cada término es la suma de los dos anteriores.

Función objetivo

Es una función matemática sobre una o diversas variables que devuelve un valor real que representa un valor asociado al evento identificado por las variables de entrada. También se llama *función de costes*, *función de pérdida* o *función de utilidad*.

Grafo conexo, ponderado y no dirigido

Decimos que un grafo es conexo si, para cada par de vértices, podemos encontrar un camino o recorrido que los comunique. Un grafo es ponderado si todas las aristas tienen asociado un valor numérico. Un grafo no dirigido es aquel en el que todas las aristas son bidireccionales. Al contrario, un grafo dirigido es aquel en el que existe al menos una arista que va del nodo A al B, pero que no existe la arista que va de B a A.

Juegos de suma cero

Se trata de un juego no cooperativo en el que la ganancia de cada jugador se equilibra con las pérdidas y ganancias de los otros jugadores. En juegos de dos jugadores, equivale a decir que lo que gana un jugador es lo mismo que lo que pierde el otro.

Montículo

Es una estructura de datos de tipo árbol binario y completo con información asociada a una ordenación. Cada nodo padre siempre tiene un valor mayor al de sus hijos. Todos los niveles del árbol están completos, salvo en algunos nodos de la última capa (se suelen llenar de izquierda a derecha). Un árbol binario es un árbol en el que cada nodo tiene como máximo dos hijos.

Pseudocódigo

Es una descripción de alto nivel de un algoritmo que utiliza las convenciones estructurales de los lenguajes de computación, pero que está orientado a ser fácilmente legible y comprensible para las personas.

Sucesión de Fibonacci

Es una sucesión de números naturales en la que cada término se obtiene como suma de los dos anteriores. Se da en algunos procesos naturales.

Enlaces de interés



Instituto de Optimización Aplicada de Hefei

Pertenece a la Facultad de Ciencias de la Computación y Tecnología de la Universidad de Hefei. Fundado en diciembre de 2016, es un grupo de investigación en el campo de la optimización matemática y la combinatoria aplicada, la investigación de operaciones, el aprendizaje automático de inteligencia computacional, las metaheurísticas y la computación evolutiva. Es un excelente recurso para profundizar en la optimización.

<http://iao.hfuu.edu.cn/blogs/science-blog/25-what-is-optimization>

Problemas de optimización con Python

Esta entrada del blog personal de Raúl E. López Briega trata sobre problemas de inteligencia artificial y, en particular, problemas de optimización aplicados en Python.

<https://relopezbriga.github.io/blog/2017/01/18/problemas-de-optimizacion-con-python/>

Graph Online

Graph Online es una herramienta que permite representar estructuras de grafos y aplicar algunos de los algoritmos de búsqueda y los caminos mínimos sobre ellos.

<http://graphonline.ru/en/>

r/ComplIntellCourses

Es una comunidad en el portal reddit.com sobre recursos formativos en inteligencia artificial.

<https://www.reddit.com/r/ComplIntellCourses/>

Virtual Library of Simulation Experiments

La Simon Fraser University ofrece en esta biblioteca decenas de funciones excepcionales que nos permiten poner a prueba los algoritmos que diseñemos para buscar óptimos globales.

<https://www.sfu.ca/~ssurjano/optimization.html>

Wolfram Challenges

La web de Stephen Wolfram propone unos cuantos retos relacionados con el diseño de algoritmos para resolver problemas.

<https://challenges.wolfram.com/tracks/algorithms>

A compendium of NP optimization problems

La web del Kungliga Tekniska Högskolan (Real Instituto de Tecnología de Estocolmo) ofrece un catálogo de problemas NP de optimización.

<https://www.nada.kth.se/~viggo/problemlist/compendium.html>

15 Sorting Algorithms in 6 Minutes

En este vídeo de Timo Bingmann puede verse una curiosa representación audiovisual de los diferentes algoritmos de ordenación. Permite comparar el funcionamiento y el rendimiento de cada uno de ellos.

<https://www.youtube.com/watch?v=kPRAoW1kECg>

Algorithm Animations and Visualizations

Es una colección de animaciones y visualizaciones de algoritmos informáticos. Nos ayudará a entender el funcionamiento de los algoritmos de búsqueda y ordenación más utilizados.

<http://algoanim.ide.sk/>

Deep Learning Cars

Este vídeo de Samuel Artz muestra una simulación en 2D en la que los coches aprenden a maniobrar por una pista utilizando una red neuronal y algoritmos evolutivos.

<https://www.youtube.com/watch?v=Aut32pR5PQA>

Genetic Algorithms - Evolution of a 2D car in Unity

En este vídeo de Tomek puede verse una simulación en 2D de la evolución de un coche que debe avanzar superando obstáculos utilizando un algoritmo genético.

<https://www.youtube.com/watch?v=FKbarpAlBkw>

Evolution of Neural Networks using Genetic Algorithm for a 3D car made in Unity

En este vídeo de Tomek puede verse una simulación en 3D de la evolución de un coche que debe maniobrar por una pista utilizando una red neuronal y un algoritmo genético.

<https://www.youtube.com/watch?v=8V2sX9BhAW8>

Jugando con Redes Neuronales

La página de TensorFlow propone jugar con las redes neuronales con un enfoque visual para ponernos a prueba a la hora de encontrar la configuración de capas intermedias y neuronas que resuelva los problemas de clasificación de dos variables. Es un interesante recurso para iniciarse en las redes neuronales.

<https://playground.tensorflow.org>

Bibliografía



Brassard, G., y Bratley, P. (1997). *Fundamentos de algoritmia*. Madrid: Prentice Hall.

Brownlee, J. (2011). *Clever algorithms: nature-inspired programming recipes*. Recuperado de <http://www.cleveralgorithms.com/nature-inspired/index.html>

Clerc, M. (2004). Discrete Particle Swarm Optimization, illustrated by the Traveling Salesman Problem. En G. C. Onwubolu, B. V. Babu (Eds.), *New Optimization Techniques in Engineering. Studies in Fuzziness and Soft Computing* (pp. 219-239). Berlín: Springer.

Cook, S. (mayo, 1971). The complexity of theorem-proving procedures. En M. A. Harrison, R. B. Banerji y J. D. Ullman (Presidencia), *STOC 1971 Proceedings of the third annual ACM symposium on Theory of computing* (pp. 151-158). Conferencia llevada a cabo en el simposio de la Association for Computing Machinery, Shaker Heights, OH. Recuperado de <https://dl.acm.org/citation.cfm?id=805047>

Duarte, A. (2008). *Metaheurísticas*. Madrid: Dykinson.

Euclides (1994). *Los elementos* (Vol. 6, cap. 2). Madrid: Gredos.

Feo, T. A., y Resende, G. C. (1995). Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6(2), 109-133. Recuperado de <https://doi.org/10.1007/BF01096763>

Gómez M. C., y Cervantes, J. (2014). *Introducción al análisis y al diseño de algoritmos*. Ciudad de México: Universidad Autónoma Metropolitana.

Guerequeta, R., y Vallecillo, A. (2000). *Técnicas de diseño de algoritmos*. Málaga: Servicio de Publicaciones de la Universidad de Málaga. Recuperado de <http://www.lcc.uma.es/~av/Libro/indice.html>

Hillier, F. S., y Lieberman, G. J. (2015). Capítulo 14. Metaheurísticas. En *Investigación de Operaciones*. Ciudad de México: McGraw-Hill Education.

Joyanes, L. (2003). *Fundamentos de programación*. Madrid: McGraw-Hill: Madrid.

Jordan, C., y Torregrosa, J. R. (1996). *Introducción a la teoría de grafos y sus algoritmos*. Valencia: Universitat Politècnica de València.

Kallehauge, B. (2008). Formulations and exact algorithms for the vehicle routing problem with time windows. *Computers and Operations Research*, 35(7), 2307-2330. Recuperado de <https://dl.acm.org/citation.cfm?id=1323908>

Lee, R. C. T., Tseng, S. S., Chang, R. C., y Tsai, Y. T. (2005). *Introducción al diseño y análisis de algoritmos*. Ciudad de México: McGraw-Hill.

Luke S. (2005). *Essentials of Metaheuristics*. Recuperado de <https://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf>

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A.H., y Teller, E. (1953). Equations of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21, 1087-1097.

Peña, R. (2006). *De Euclides a Java: Historia de los algoritmos y de los lenguajes de programación*. Madrid: Nivola.

Shi, Y., y Eberhart, R. C. (mayo, 1998). A modified particle swarm optimizer. En P. K. Simpson (Presidencia), *Proceedings of the IEEE International Conference on Evolutionary Computation* (pp. 69-73). Simposio llevado a cabo en la conferencia del Institute of Electrical and Electronics Engineers, Anchorage, AK. Recuperado de <https://ieeexplore.ieee.org/document/699146>



Autor
Raúl Reyero Díez