ANÁLISIS Y DISEÑOS DE ALGORITMOS

Resumen

Melanie Mariam
Cruz Morgado

Índice

Complejidad

Noción de complejidad

La **complejidad algorítmica** es una medida de cómo crecen los recursos que necesita un algoritmo para resolver un problema cuando el tamaño del problema crece. Los más usuales son tiempo (complejidad temporal) y memoria (complejidad espacial).

- **Tiempo de ejecución:** Es una función (T(n)) que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de problema *n*.

  Depende de:

  - Factores externos:
    - La máquina en la que se va a ejecutar.
    - El compilador.
    - Los datos de entrada suministrados en cada ejecución.
  - Factores internos:
    - El número de instrucciones que ejecuta el algoritmo y su duración.

  Puede calcularse:

  - A posteriori (Análisis empírico): Ejecutar el algoritmo para distintos valores de entrada y cronometrar el tiempo de ejecución. Es una medida del comportamiento del algoritmo en su entorno y el resultado depende de los factores externos e internos.
  - A priori (Análisis teórico): Obtener una función que represente el tiempo de ejecución (en operaciones elementales) del algoritmo para cualquier valor de entrada. El resultado depende solo de los factores internos. No es necesario implementar y ejecutar los algoritmos. No obtiene una medida real del comportamiento del algoritmo en el entorno de aplicación.

  * Las operaciones elementales son las que realiza el ordenador en un tiempo acotado por una constante.

- **Complejidad espacial:** En ella no se tiene en cuenta lo que ocupa la codificación del problema, solo se tiene en cuenta lo que es imputable al algoritmo.

Cotas de complejidad

Cuando aparecen diferentes casos para una misma talla *n*, se introducen las siguientes medidas de la complejidad:

- Cota superior del algoritmo (Caso peor) → Cs (n)
- Cota inferior del algoritmo (Caso mejor) → Ci (n)
- Coste promedio (Caso promedio) → Cm (n). Es difícil de evaluar a priori, porque es necesario conocer la distribución de probabilidad de la entrada, ya que no es la media de la cota inferior y de la superior.

Todas son funciones del tamaño del problema.

El estudio de la complejidad resulta realmente interesante para tamaños grandes de problema por:

- Las diferencias "reales" en tiempo de ejecución de algoritmos con diferente coste para tamaños pequeños del problema no suelen ser muy significativas.
- Es lógico invertir tiempo en el desarrollo de un buen algoritmo solo si se prevé que este realizará un gran volumen de operaciones.

Análisis asintótico: Estudio de la complejidad para tamaños grandes de problema. Permite clasificar las funciones de complejidad de forma que podamos compararlas entre si fácilmente. Para ello, se definen clases de equivalencia que engloban a las funciones que "crecen de la misma forma".

Notación asintótica: Notación matemática utilizada para representar la complejidad cuando el tamaño del problema (n) crece (n → ∞).

- Cota superior: O

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geqslant n_0, g(n) \leqslant cf(n)\}$$

- Cota inferior: Ω

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geqslant n_0, g(n) \geqslant cf(n)\}$$

- Coste exacto: θ

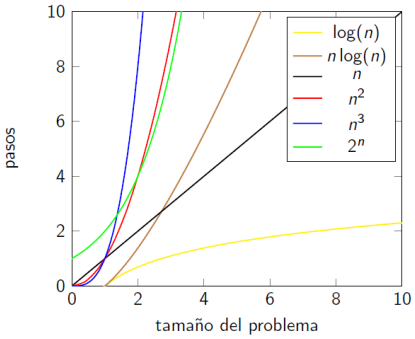
$$\Theta(f) = \{g : \mathbb{N} \to R^+ | \exists c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \ \forall n \geqslant n_0, cf(n) \leqslant g(n) \leqslant df(n)\}$$

O lo que es lo mismo: θ(f) = O(f) ∩ Ω(f)

Esta notación permite agrupar en clases funciones con el mismo crecimiento en **jerarquías de funciones:**

$$\underbrace{O(1)}_{\text{constantes}} \subset \underbrace{O(\log\log n)}_{\text{sublogarítmicas}} \subset \underbrace{O(\log n) \subset O(log^{a(>1)}n)}_{\text{logarítmicas}} \subset \underbrace{O(\sqrt{n})}_{\text{sublineales}} \subset \underbrace{O(n)}_{\text{lineales}} \subset \underbrace{O(n\log n)}_{\text{lineal-logarítmicas}}$$

$$\subset \underbrace{O(n^2) \subset O(n^{a(>2)})}_{\text{polinómicas}} \subset \underbrace{O(2^n) \subset O(a(>2)^n)}_{\text{exponenciales}} \subset \underbrace{O(n!) \subset O(n^n)}_{\text{superexponenciales}}$$



Cálculo de complejidades

Pasos para obtener las cotas de complejidad:

1. Determinar la talla o tamaño del problema.
2. Determinar los casos mejor y peor (instancias para las que el algoritmo tarda más o menos).
   * Para algunos algoritmos, el caso mejor y el caso peor son el mismo, ya que se comportan igualmente para cualquier instancia del mismo tamaño.
3. Obtención de las cotas para cada caso.
   a. Algoritmos iterativos.
   b. Algoritmos recursivos. El coste depende de las llamadas recursivas y, por lo tanto, debe definirse recursivamente.

Relaciones de recurrencia: Expresión que relaciona el valor de una función *f* definida para un entero *n* con uno o más valores de la misma función para valores menores que *n*.

Se usan para expresar la complejidad de un algoritmo recursivo, aunque también son aplicables a los iterativos.

Si el algoritmo dispone de mejor y peor caso, puede haber una relación de recurrencia para cada caso.

La complejidad de un algoritmo se obtiene en tres pasos:

1. Determinación de la talla del problema.
2. Obtención de las relaciones de recurrencia del algoritmo.
3. Resolución de las relaciones mediante el método de sustitución.

Ejemplos de complejidades

Algoritmos iterativos.

- **Buscar un elemento en un vector**:
  - Cs (n) = 3n+2 → Cs (n) ∈ O (n)
  - Ci (n) = 3 → Ci (n) ∈ Ω (1)
- **Sumar elementos de un vector**:
  - C (n) = n+2 → C (n) ∈ θ (n)

Algoritmos recursivos.

- **Búsqueda binaria**: $T(n) \in \begin{cases} \Theta(1) & n = 1 \\ \Theta(1) + T(n/2) & n > 1 \end{cases}$, donde n = ult – pri + 1 → T(n) ∈ O (log n)
- **Ordenación por selección**: $T(n) = \begin{cases} \Theta(1) & n = 1 \\ \Theta(n) + T(n-1) & n > 1 \end{cases}$, donde n = v.size() – pri → T(n) ∈ θ (n^2)

Quicksort

Algoritmo de ordenación por partición.

Pasos:

1. Elección del pivote. El elemento pivote sirve para dividir en dos partes el vector. Su elección (al azar, primer elemento, elemento central o elemento mediana) define variantes del algoritmo.
2. Se divide el vector en dos partes:
   - Parte izquierda del pivote (elementos menores).
   - Parte derecha del pivote (elementos mayores).
3. Se hacen dos llamadas recursivas, una con cada parte del vector.

```
1  void quicksort( int v[], int pri, int ult ) {
2    if( ult <= pri )
3      return;
4    int p = pri;
5    int j = ult;
6    while(p < j) {
7      if( v[p+1] < v[p] ) {
8        swap( v[p+1], v[p] );
9        p++;
10     } else {
11       swap( v[p+1], v[j] );
12       j--;
13     }
14   }
15   quicksort(v, pri, p-1);
16   quicksort(v, p+1, ult);
17 }
```

- Mejor caso: subproblemas (n/2, n/2).

$$T(n) \in \begin{cases} \Theta(1) & n \leqslant 1 \\ \Theta(n) + T(\frac{n}{2}) + T(\frac{n}{2}) & n > 1 \end{cases}$$ → T(n) ∈ Ω (n $\log_2$n)

- Pero caso: subproblema (0, n-1) o (n-1, 0).

$$T(n) \in \begin{cases} \Theta(1) & n \leqslant 1 \\ \Theta(n) + T(0) + T(n-1) & n > 1 \end{cases}$$ → T(n) ∈ O (n^2)

Quicksort mediana: En este algoritmo estamos forzando el caso mejor de la versión anterior, en el que el elemento seleccionado era la mediana.

Obtener la mediana:

- Coste menor que O ($n \log_2 n$).
- Se aprovecha el recorrido para reorganizar elementos y para encontrar la mediana en la siguiente subllamada.
- Su complejidad es por tanto de θ ($n \log_2 n$).

Diseño de algoritmos

Estudia la aplicación de métodos para resolver problemas en programación.

Objetivos:

- Dar a conocer las familias más importantes de problemas algorítmicos y estudiar diferentes esquemas o paradigmas de diseño aplicables para resolverlos.
- Aprender a instanciar (particularizar) un esquema genérico para un problema concreto, identificando los datos y operaciones del esquema con las del problema, previa comprobación de que se satisfacen los requisitos necesarios para su aplicación.
- Justificar la elección de un determinado esquema cuando varios de ellos pueden ser aplicables a un mismo problema.

Resolución de problemas:

- Diseño ad hoc (frecuentemente “fuerza bruta”).
    - Algoritmos dependientes del problema y no generalizables.
    - Dificultad de adecuar cambios en la especificación.

- Esquemas.
    - Cada esquema representa un grupo de algoritmos con características comunes (análogos).
    - Permiten la generalización y reutilización de algoritmos.
    - Cada instanciación de un esquema da lugar a un algoritmo diferente.

Esquemas algorítmicos más comunes

Divide y vencerás

Técnica de diseño de algoritmos que consiste en:

- Descomponer el problema en subproblemas de menor tamaño que el original.
- Resolver cada subproblema de forma individual e independiente.
- Combinar las soluciones de los subproblemas para obtener la solución del problema original.

Consideraciones:

- No siempre un problema de talla menor es más fácil de resolver.
- La solución de los subproblemas no implica necesariamente que la solución del problema original se pueda obtener fácilmente.

Aplicable si encontramos:

- Forma de descomponer un problema en subproblemas de talla menor.
- Forma directa de resolver problemas menores a un tamaño determinado.
- Forma de combinar las soluciones de los subproblemas que permita obtener la solución del problema original.

Esquema:

```
1  Solucion DyV( Problema x ) {
2
3      if (pequeno(x))
4          return trivial(x);
5
6      list<Solucion> s;
7      for( Problema q : descomponer(x) )
8          s.push_back( DyV(q) );
9      return combinar(s);
10 }
```

Teorema de reducción:

Los mejores resultados en cuanto a coste se consiguen cuando los subproblemas son aproximadamente del mismo tamaño y no contienen subproblemas comunes.

Si se cumple (nº de subproblemas = cte. de división de tamaño de problema = a):

$$T(n) = aT\left(\frac{n}{a}\right) + g(n) \qquad g(n) \in \Theta(n^k) \qquad T(n) = \begin{cases} \Theta(n^k) & k > 1 \\ \Theta(n \log n) & k = 1 \\ \Theta(n) & k < 1 \end{cases}$$

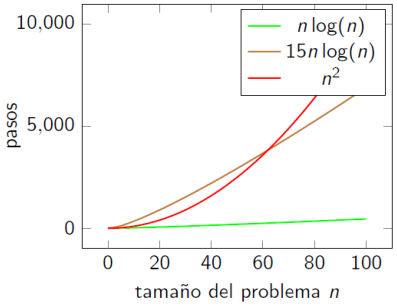
- **Ordenación de un vector ascendentemente:** f(n) ∈ θ (n^2)

Mergesort:

Utiliza la función merge que obtiene un vector ordenado como fusión de dos vectores también ordenados.

merge(v,pi,m,pf) ∈ θ (n), donde n = pf - pi + 1.

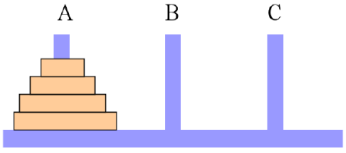
Ecuación de recurrencia: $T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$ → f(n) ∈ θ (n log n)



Comparación de algoritmos: Solo si el coeficiente de *n log n* es mayor que *e* puede pasar que, para ciertos valores pequeños de *n*, *n log n* sea menos ventajoso que *n^2* (asumiendo logaritmos naturales).

Las torres de Hanoi:

Colocar los discos de la torre A en la C empleando como ayuda la torre B.



$\text{hanoi}(n, A \xrightarrow{B} C)$ = $\text{hanoi}(n-1, A \xrightarrow{C} B)$ o $\text{hanoi}(n-1, B \xrightarrow{A} C)$ o *mover(A->B)*

Los discos han de moverse uno a uno sin colocar nunca un disco sobre otro más pequeño.

Ecuación de recurrencia para el coste exacto: $T(n) = \begin{cases} 1 & n = 1 \\ 1 + 2T(n-1) & n > 1 \end{cases}$ → T(n) = 2^n – 1 ∈ θ (2^n)

Programación dinámica

Hay problemas con soluciones recursivas elegantes, compactas e intuitivas, **pero** prohibitivamente lentas, debido a que resuelven repetidamente los mismos problemas.

La programación dinámica se trata de **evitar repeticiones** guardando resultados de subproblemas (***memoización***) a expensas de aumentar la complejidad espacial.

Principio de optimalidad: Condición necesaria para poder aplicar la programación dinámica. Es un problema que tiene una subestructura óptima si una solución óptima puede construirse eficientemente a partir de las soluciones óptimas de sus subproblemas.

Problemas clásicos para los que resulta eficaz:

- Quicksort y Mergesort.
- Cálculo de los números de Fibonacci.
- Problemas con cadenas: La subsecuencia común máxima de dos cadenas y la distancia de edición entre dos cadenas.
- Problemas sobre grafos: El viajante de comercio, algoritmos de Dijkstra, de Warshall y de Floyd.
- **Cálculo del coeficiente binomial:**

```
unsigned binomial( unsigned n, unsigned r){

    if ( r == 0 || r == n )
        return 1;

    return binomial( n-1, r-1 ) + binomial( n-1, r );
}
```

→ $T(n,r) \sim g(n,r) \in O(2^{\min(r,n-r)})$

¡Esta solución recursiva no es aceptable!

Para solucionar el inconveniente de tener subproblemas repetidos usaremos almacenes.

Solución trivial de programación dinámica:

```
unsigned binomial(unsigned n, unsigned r){
  unsigned M[n+1][r+1];

  for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
  for (unsigned i=1; i <= r;   i++) M[i][i]= 1;

  for (unsigned j=1; j<=r; j++)
      for (unsigned i=j+1; i<=n-r+j; i++)
                M[i][j]= M[i-1][j-1] + M[i-1][j];
  return M[n][r];
}
```

Corte temporal exacto:

$$T(n,r) = 1 + \sum_{i=0}^{n-r} 1 + \sum_{i=1}^{r} 1 + \sum_{j=1}^{r} \sum_{i=j+1}^{n-r+j} 1 = rn+n-r^2+1 \in \Theta(rn)$$

- **Coste de tubos:**

Una empresa compra tubos de longitud n y los corta en tubos más cortos, que luego vende; pero probar todas las formas de cortar es prohibitivo. El corte le sale gratis. El precio de venta de un tubo de longitud i es pi.

Una forma de resolver el problema recursivamente es cortar el tubo de longitud n de las n formas posibles y buscar el corte que maximiza la suma del precio del trozo cortado y del resto, suponiendo que el resto del tubo se ha cortado de forma óptima.

Complejidad temporal:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + \sum_{j=0}^{n-1} T(j) & \text{en otro caso} \end{cases}$$ → T(n) = 1+2T(n-1) → T(n)=2^n – 1 + 2^n ∈ O (2^n) → INEFICIENTE

Solución iterativa o recursiva con almacén:

- Complejidad temporal → O (n^2)
- Complejidad espacial → O (n)

La mochila discreta:

En el problema de la mochila…

- Tenemos: *N* objetos con valores y pesos conocidos, y una mochila con capacidad máxima de carga.
- Finalidad: Calcular cuál es el valor máximo que puede trasportar la mochila sin sobrepasar su capacidad.

Solución iterativa o recursiva con almacén:

- Complejidad temporal → O (n^2)
- Complejidad espacial → O (n)

La mochila discreta es un caso particular, la variante más difícil del problema de la mochila. Los pesos son cantidades discretas o discretizables. Es un problema de optimización.

Pueden pasar dos cosas:

- Rechazar el objeto y no hay ganancia adicional, pero la capacidad de la mochila no ser reduce.
- Seleccionar el objeto y la ganancia adicional en su valor, a costa de reducir la capacidad en su peso.

Se selecciona la alternativa que mayor ganancia global resulte.

- En el mejor de los casos:

  Ningún objeto cabe en la mochila → Tmejor (n) ∈ Ω (n).

- En el peor de los casos:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + 2T(n-1) & \text{en otro caso} \end{cases} \rightarrow T(n) = 2^i - 1 + 2^i T(n-i) \rightarrow T(n) = 2^n - 1 + 2^n \in O(2^n)$$

Algoritmos voraces (Greedy)

Un algoritmo voraz es aquel que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción local óptima en cada paso con la esperanza de llegar a una solución general óptima. O sea, descompone el problema en un conjunto de decisiones, elige la más prometedora y nunca reconsidera las decisiones ya tomadas.

Características:

- Son algoritmos eficientes y fáciles de implementar.
- Puede que no se encuentre la solución óptima.
- Incluso puede que no se encuentre ninguna solución.
- Es necesario un buen criterio de selección para tener garantías.
- Se aplican mucho en problemas con muy alta complejidad y cuando es suficiente una solución aproximada.

El problema de la mochila continua:

En él, a diferencia de la mochila discreta, se permite fraccionar los objetos. El problema se reduce a seleccionar un subconjunto de (fracciones de) los objetos disponibles, que cumpla las restricciones y que maximice la función. Se calcula mediante un algoritmo voraz (vector óptimo), con el que se encuentra la solución óptima.

Árbol de recubrimiento de coste mínimo:

Se resuelve mediante los algoritmos voraces: Prim y Kruskal. En ambos se van añadiendo arcos de uno en uno a la solución. La diferencia está en la forma de elegir los arcos a añadir.

Algoritmo de Prim:

Va construyendo un único árbol de recubrimiento añadiendo vértices uno a uno. Se mantiene un conjunto de vértices explorados. Se coge un vértice al azar y se añade al conjunto de explorados. En cada caso:

- Buscar el arco de mínimo peso que va de un vértice explorado a uno que no lo está.
- Añadir el arco a la solución y el vértice a los explorados.

Mejora:

- No hace falta recorrer todos los arcos cada vez.
- Si cambia el mínimo es a causa del último vértice añadido.
- Hay que guardarse, para cada vértice, el último mínimo.

Algoritmo de Kruskal:

Va construyendo un bosque al que va añadiendo aristas, hasta obtener un único árbol de recubrimiento.

La implementación básica produce una complejidad temporal de O (*A log A + V^2*), pero otras implementaciones más eficientes de conjuntos disjuntos llevan a una complejidad O (*A log A*).

Algoritmo de Dijkstra:

Dado un grafo no dirigido y ponderado con pesos no negativos, calcular el coste de los caminos más cortos desde un vértice a todos los demás.

El fontanero diligente:

Un fontanero necesita hacer *n* reparaciones urgentes y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes.

La asignación de tareas:

Una asignación válida es aquella en la que a cada trabajador solo le corresponde una tarea y cada tarea está asignada a un trabajador. Dada una asignación válida, definimos el coste de esta. Diremos que una asignación es óptima si es de mínimo coste.

Algoritmos de vuelta atrás (Algoritmo de búsqueda y enumeración)

Algunos problemas solo se pueden resolver mediante el estudio exhaustivo del conjunto de posibles soluciones al problema. De entre todas ellas, se podrá seleccionar un subconjunto o bien, aquella que consideremos la mejor, la solución óptima.

Vuelta atrás proporciona una forma sistemática de generar todas las posibles soluciones a un problema. Generalmente se emplea en la resolución de problemas de selección u optimización en los que el conjunto de soluciones posibles es finito, en los que se pretende encontrar una o varias soluciones que sean factibles (que satisfagan unas restricciones) y/u óptimas (que optimicen una cierta función objetivo).

Características:

La solución debe poder expresarse mediante una tupla de decisiones. Estas pueden pertenecer a dominios diferentes entre sí, pero estos siempre serán discretos o discretizables.

Es posible que se tenga que explorar todo el espacio de soluciones. Los mecanismos de poda van dirigidos a disminuir la probabilidad de que esto ocurra.

La estrategia puede proporcionar una solución factible, todas las soluciones factibles, la solución óptima y/o las *n* mejores soluciones factibles al problema. A costa, en la mayoría de los casos, de complejidades prohibitivas.

La generación y búsqueda de la solución se realiza mediante un sistema de prueba y error.

Se trata de un recorrido sobre una estructura arbórea imaginaria.

```
7  void backtracking( node n, solution& the_best ){
8
9      if ( is_leaf(n) ) {
10         if( is_best( solution(n), the_best ) )
11             the_best = solution(n);
12         return;
13     }
14
15     for( node a : expand(n) )
16         if( is_feasible(a) && is_promising(a) )
17             backtracking( a, the_best );
18
19     return;
20 }
```

El problema de la mochila (general):

Dados *n* objetos con valores y pesos, y una mochila que solo aguanta un peso máximo.

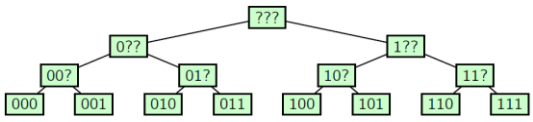
Restricción: No sobrepasar el límite de peso.

Función objetivo: El valor transportado debe ser máximo.

Cálculo de la solución óptima: Mediante programación dinámica (objetos no fragmentables y pesos discretos).

Algoritmos voraces: Objetos fragmentables.

Pero si no podemos fragmentar los objetos y los pesos son valores reales, debemos recorrer todas las combinaciones y generar soluciones factibles.



→ Complejidad temporal: θ (n 2^n)

Se puede acelerar el programa evitando explorar ramas que no pueden dar soluciones factibles.

- Peor caso: Todos los objetos caben → O (n 2^n)
- Mejor caso: Ningún objeto cabe → Ω (n^2)

Además, el peso se puede ir calculando a medida que se rellena la solución y eso mejora el mejor caso a Ω (n).

En resumen, para calcular la solución óptima hay que recorrer todas las soluciones factibles, calcular su valor y quedarse con la mejor de todas.

El caso peor se puede mejorar mediante el uso de podas más ajustadas. Interesa que los mecanismos de poda "actúen" lo antes posible. Se pueden obtener usando la solución voraz al problema de la mochila continua.

La solución al problema de la mochila continua es siempre mayor que la solución al problema de la mochila discreto.

Si la poda se basa en la mochila continua:

- Peor caso: O (2^n * n * log n)
- Mejor caso: Ω (n)

La efectividad de la poda también puede aumentarse partiendo de una solución factible muy "buena", cuasi-óptima. Una posibilidad es usar la solución voraz para la mochila discreta. También puede ser relevante la forma en la que se "despliega el árbol" (completar la tupla primero con los 1 y después con los 0).

Permutaciones:

Dado un número entero positivo n, escribir un algoritmo que muestre todas las permutaciones de la secuencia (0, …, n-1).

Solución: $X = (x_0, x_1, \ldots, x_{n-1}) \in \{0, 1, \ldots, n-1\}$ cada permutación será cada una de las reordenaciones de X. No hay función objetivo, sino que se buscan todas las combinaciones factibles.

Restricción: X no puede tener elementos repetidos.

El viajante de comercio:

Dado un grafo ponderado g = (V, E) con pesos no negativos, el problema consiste en encontrar un ciclo hamiltoniano de mínimo coste.

Un ciclo hamiltoniano es un recorrido en el grafo que recorre todos los vértices solo una vez y regresa al de partida. Restricción: No se puede visitar dos veces el mismo vértice.

El coste de un ciclo viene dado por la suma de los pesos de las aristas que lo componen.

La solución algorítmica propuesta resulta inviable por su prohibitiva complejidad: O(n^n). Por ello, y para acelerar la búsqueda, se aplica algún mecanismo de poda basado en la mejor solución hasta el momento (empezar con la solución voraz) y diseñar algún heurístico voraz que permita cumplir el objetivo, por ejemplo, utilizando las ideas de los algoritmos de Prim o Kruskal (relajación de las restricciones).

El problema de las n reinas:

En un tablero de ajedrez de *n* x *n*, obtener todas las formas de colocar *n* reinas de forma que no se ataquen mutuamente (Restricciones: ni en la misma fila, ni columna, ni diagonal).

Solución: Cada reina se coloca en una fila, el problema es determinar en qué columna se colocará.

El coloreado de grafos:

Dado un grafo G, encontrar el menor número de colores con el que se pueden colorear sus vértices de forma que no haya dos vértices adyacentes con el mismo color.

- **La función compuesta mínima:**

Dadas dos funciones *f(x)* y *g(x)* y dos números cualquiera *x* e *y*, encontrar la función compuesta mínima que obtiene el valor *y* a partir de *x*, tras aplicaciones sucesivas e indistintas de *f(x)* y *g(x)*.

Solución: Tupla de la cual no se conoce el tamaño a priori.

Función objetivo: Minimizar el tamaño de la tupla solución.

- **El recorrido del caballo del ajedrez:**

Encontrar una secuencia de movimientos "legales" de un caballo de ajedrez de forma que este pueda visitar las 64 casillas de un tablero sin repetir ninguna.

- **El laberinto con cuatro movimientos:**

Se dispone de una cuadrícula *n* x *m* de valores {0, 1} que representa un laberinto. Un valor 0 en una casilla cualquiera de la cuadrícula indica una posición inaccesible; por el contrario, con el valor 1 se simbolizan las casillas accesibles.

Finalidad: Encontrar un camino que permita ir de la posición (1, 1) a la posición (n, m) con cuatro tipos de movimiento (arriba, abajo, derecha, izquierda).

- **La empresa naviera:**

Supongamos una empresa naviera que dispone de una flota de *N* buques cada uno de los cuales transporta mercancías de un valor que tardan en descargarse un tiempo, pero solo hay un muelle de descarga y tiene un tiempo máximo de utilización.

Finalidad: Diseñar un algoritmo que determine el orden de descarga de los buques de forma que el valor descargado sea máximo sin sobrepasar el tiempo de descarga. Si se elige un buque para descargarlo, es necesario que se descargue en su totalidad.

- **La asignación de turnos:**

Estamos al comienzo del curso y los alumnos deben distribuirse en turnos de prácticas. Para solucionar este problema se propone que valoren los turnos de práctica disponibles a los que desean ir en función de sus preferencias.

Se dispone de una matriz de preferencias *N* x *T*, siendo *N* el número de alumnos y *T* los turnos disponibles, y de un vector con *T* elementos que contiene la capacidad máxima de alumnos en cada turno.

Finalidad: Encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia sin exceder la capacidad de los turnos. (Complejidad exponencial)

- **Sudoku.**

Tenemos 9 casillas de 3x3 en las que hay que rellenar las casillas que no tengan número siguiendo las normas de que no se repita el número que pongamos ni en la fila, ni la columna, ni el cuadrado 3x3.

Ramificación y poda (Algoritmo de búsqueda y enumeración)

Variante del diseño de Vuelta Atrás en la que se usan cotas para podar ramas que no son de interés.

Nodo vivo: Nodo con posibilidades de ser ramificado (visitado, pero no completamente expandido). Estos se almacenan en estructuras que faciliten su recorrido y eficiencia de la búsqueda: pila (en profundidad - estrategia LIFO), cola (en anchura – estrategia FIFO) o cola de prioridad (dirigida – primero el más prometedor).

Para ver si un nodo es prometedor, se compara la mejor solución obtenida con una solución optimista del nodo.

Etapas:

1. Partimos del nodo inicial del árbol.
2. Se asigna una solución pesimista (subóptima, soluciones voraces).
3. Selección:
    - Extracción del nodo a expandir del conjunto de nodos vivos.
    - La elección depende de la estrategia empleada.
    - Se actualiza la mejor solución con las nuevas soluciones encontradas.
4. Ramificación: Se expande el nodo seleccionado en la etapa anterior, dando lugar al conjunto de sus nodos hijos.
5. Poda:
    - Se eliminan (podan) nodos que no contribuyen a la solución.
    - El resto de nodos se añaden al conjunto de nodos vivos.
6. El algoritmo finaliza cuando se agota el conjunto de nodos vivos.

Proceso de poda:

- Cota optimista:
    - Estima, a mejor, el mejor valor que podría alcanzarse al expandir el nodo.
    - Puede que no haya ninguna solución factible que alcance ese valor.
    - Normalmente se obtienen relajando las restricciones del problema.
    - Si la cota optimista de un nodo es peor que la solución en curso, se puedes podar dicho nodo.
- Cota pesimista:
    - Estima, a peor, el mejor valor que podría alcanzarse al expandir el nodo.
    - Ha de asegurar que existe una solución factible con un valor mejor que la cota.
    - Normalmente se obtienen mediante soluciones voraces del problema.
    - Se puede eliminar un nodo si su cota optimista es peor que la mejor cota pesimista.
    - Permite la poda aún antes de haber encontrado una solución factible.
- Cuanto más ajustadas sean las cotas, más podas se producirán.

Características:

- La estrategia puede proporcionar:
    - Todas las soluciones factibles.
    - Una solución al problema.
    - La solución óptima al problema.
    - Las *n* mejores soluciones.

- Objetivo de esta técnica: Mejorar la eficiencia en la exploración del espacio de soluciones.
- Desventaja/Necesidades:
  - Encontrar una buena cota optimista (problema relajado).
  - Encontrar una buena solución pesimista (estrategias voraces).
  - Encontrar una buena estrategia de exploración (cómo ordenar).
  - Mayor requerimiento de memoria que los algoritmos de Vuelta Atrás.
  - Las complejidades en el peor caso suelen ser muy altas.
- Ventaja: Suelen ser más rápidos que Vuelta Atrás.

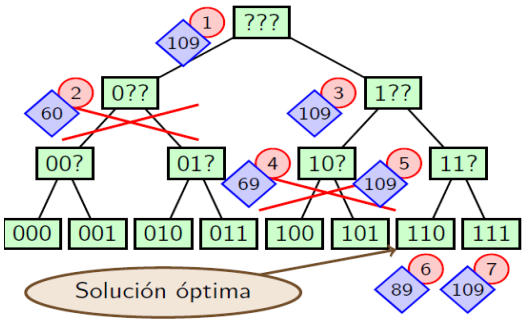
El viajante de comercio: Se mejora con este algoritmo respecto a Vuelta Atrás.

La función compuesta: La mejora es irrelevante, no vale la pena hacerlo con este algoritmo.

El problema de la mochila (general):

Se puede llegar antes a la solución óptima adecuando el orden de exploración del árbol de soluciones según nuestros intereses. Se priorizará para su exploración aquellos nodos más prometedores.

Se poda mediante la función de cota: Cada nodo toma cota el valor que resultaría de incluir en la solución aquellos objetos pendientes de tratar (sustituir '?' por '1') independientemente de que quepan o no.



Espacio de soluciones: En el orden de expansión se prioriza los nodos con mayor cota.

El puzzle:

Disponemos de un tablero con n^2 casillas y de n^2 – 1 piezas numeradas del uno al n^2 - 1. Dada una ordenación inicial de todas las piezas. En el tablero, queda solo una casilla vacía (con valor 0), a la que denominamos hueco.

Objetivo: Transformar dicha disposición inicial de las piezas en una disposición final ordenada, es decir, que en la casilla (i, j) se encuentre la pieza numerada $n(i-1) + j$ y en la casilla (n, n) se encuentre el hueco.

Según se relaje el problema, tenemos dos funciones de coste diferentes:

- Calcular el número de piezas que están en una posición distinta de la que les corresponden la disposición final.
- Calcular la suma de las distancias de Manhattan, desde la posición de cada pieza, su posición en la disposición final.