# Algorítmica


**Los Del DGIIM**, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

# Algorítmica

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2023-2024

# Índice general

# 1. Backtracking y Branch & Bound

Si tenemos un problema que podemos resolver con un conjunto de decisiones, podemos usar *backtracking*.

La solución al problema la representarmos por una $n-$upla, de forma que cada decisión se toma de un conjunto finito de candidatos a seleccionar. Aquí sí que podemos volver atrás tras una decisión, no como en la técnica Greedy. Además, las mejores decisiones a tomar no tenemos por qué tomarlas al inicio. El orden de las decisiones tomadas puede importar o no.

Cuando veamos que una solución es inviable, no seguimos por ella, sino que retrocedemos hasta cierto punto y proseguimos por otras posibles soluciones. Una vez alcanzada una solución, puede que queramos encontrar una mejor. En dicho caso continuamos buscando otra solución. Desde este punto, podemos pensar en los algoritmos *backtracking* de forma similar a los fuerza bruta, pero de forma inteligente.

Branch & Bound es una modificación de este método: Este método busca una solución como en el método de backtracking, pero cada solución tiene asociado un costo y la solución que se busca es una de mínimo costo llamada óptima. Además de ramificar una solución padre (branch) en hijos se trata de eliminar de consideración aquellos hijos cuyos descendientes tienen un costo que supera al óptimo buscado acotando el costo de los descendientes del hijo (bound). La forma de acotar es un arte que depende de cada problema. La acotación reduce el tiempo de búsqueda de la solución óptima al "podar" (pruning) los subarboles de descendientes costosos.

*Backtracking* busca agilizar algoritmos de un orden grande de forma inteligente.

### Coloreo de grafos

Queremos colorear un grafo plano (o mapa) con no más de 4 colores, de forma que nodos (o países) adyacentes tengan distinto color. El problema a resolver depende del criterio a optimizar (si un color es más caro, si queremos usar el mínimo número de colores, ...).

### Problema de la mochila

$N$ paquetes de datos a transmitir en un tiempo $t_i$ y con ganancia $g_i$. Seleccionar el conjujnto de paquetes a transmitir en un tiempo $T$ maximizando la ganancia.

#### Problema del laberinto

Dado un laberinto, encontrar el camino desde la entrada hasta la salida (puede que queramos optimizar la longitud del camino).

#### Buscaminas

En la resolución del juego "buscaminas", hay casos donde no podemos asegurar por ciertos algoritmos la existencia de una bomba o no. Puede ser de utilidad emplear algoritmos *backtracking*: suponer que hay una bomba y a partir de ahí decidir si se puede llegar a una solución sin llegar a contradicciones. De otra forma, hay una bomba en dicho sitio.

Esto mismo lo podemos aplicar a resolver sudokus.

### 1.0.1. Diferencias Backtracking B&B frente a fuerza bruta

La principal diferencia es el uso de funciones de acotación o poda, de forma que si no podemos alcanzar una decisión por alguna rama, cortamos dicha rama (no seguimos por dicha solución, sino que retrocedemos y tomamos decisiones en base a ello).

#### Generar todas las combinaciones de n bits

Por ejemplo, para 2 bits sería `00`, `01`, `10`, `11`.
Una solución puede ser:

```
void completa_binario(vector<int>& v, int pos){
   if(pos == v.size()){
      imprimeVector(v);
   }else{
      v[pos] = 0;
      completa_binario(v, pos+1);
      v[pos] = 1;
      completa_binario(v, pos+1);
   }
}
```

Del orden $O(2^n)$. Intuitivamente, nos estamos moviendo por un árbol de estados.

#### Coloreo de grafos

```
void completa_grafo(vector<int>& v, int pos){
   if(pos == v.size()){
      comprueba_factibilidad(v)
   }else{
      v[pos] = 0;
      completa_grafo(x, pos+1);
      v[pos] = 1;
      completa_grafo(x, pos+1);
```

```
            v[pos] = 2;
            completa_grafo(x, pos+1);
            v[pos] = 3;
            completa_grafo(x, pos+1);
        }
    }
```

Del orden $O(4^n)$. La función de factibilidad podría hacer, por ejemplo, que si tenemos dos valores repetidos seguidos como `011`, entonces la solución no es factible, no seguimos por esa rama.

Para mejorar la solución, podemos reducir el espacio de decisiones, lo que agiliza el programa.

### Viajante de comercio

```
void completa_kario(vector<int>& v, int pos, int k){
    if(pos == v.size()){
        procesa_vector(v)
    }else{
        for(int j = 0; j < k; j++){
            x[pos] = j;
            completa_kario(v, pos+1, j);
        }
    }
}
```

En todos estos problemas vemos que se establece una estructura de árbol imaginario sobre el conjunto de posibles soluciones. La forma en la que se generan las soluciones es equivalente a realizar un recorrido preorden del árbol. Sólo se procesan las hojas que corresponden con soluciones completas.

### Definiciones

**Solución parcial.** Tupla o vector al que no se le han asignado todos sus componentes.

**Función de poda.** Evalúa si una solución parcial es viable o no (por no satisfacer las restricciones, porque tenemos una solución mejor o porque no satisface otras restricciones).

**Restricciones explícitas.** Reglas que restringe las soluciones a un subconjunto de todas las combinaciones.

**Restricciones implícitas.** Reglas que nos dicen cuándo una solución parcial nos puede llevar a la solución objetivo. Relacionan una decisión a tomar con las anteriores.

**Árbol de estados.** Árbol imaginario formado.

**Estado del problema.** Cada uno de los nodos del árbol.

**Estado solución.** Nodos del árbol que representan una solución al problema.

**Estado respuesta.** Una solución al problema que satisface restricciones implícitas.

**Nodo vivo.** Nodo que ya ha sido generado pero del que aún no se han generado todos sus descendientes.

**Nodo muerto.** Nodo que ha sido generado, y o bien se ha podado o se han generado todos los descendientes.

**e-nodo o nodo en expansión.** Nodo vivo del que actualmente se están generando los descendientes.

#### Problema de la suma de subconjuntos

Dados $n$ números positivos y uno más, $M$, encontrar todos los subconjuntos cuya suma valga $M$.

Podemos dar la solución con una tupla de longitud variable (que contenga los índices de los elementos que forman parte de la solución) o con una tupla de longitud fija $n$ (que contenga por cada elemento si se selecciona o no).

#### Problema de las $n$ reinas

Dado un tablero de ajedrez, cómo colocar $n$ reinas (supondremos 8) en el tablero de forma que ninguna pueda atacar a otra (no puede haber dos reinas en la misma fila, columna o diagonal).

- El vector solución puede ser un vector de 8 posiciones de números a elegir (cada uno representa una posición del tablero) entre 1 y 64. Tenemos $64^8$ posibilidades.
- Como cada reina sólo puede estar en una fila, podemos tener 8 vectores de longitud 8 donde en cada uno indicamos la posición de la reina de dicha fila. Son $8^8$ posibilidades.
- También sabemos que no pueden estar en la misma columna. Reducimos a 8! posibilidades.

### 1.0.2. Diferencias entre Backtracking y Branch & Bound

Ambos métodos recorren el árbol de estados y ámbos métodos utilizan funciones de poda para eliminar ramas que no conducen a soluciones.

- Backtracking: cuando el nuevo hijo del enodo ha sido generado, este se combierte en enodo hasta explorar todos sus descendientes.
- Branch & Bound: El enodo continúa siéndolo hasta que se han generado todos sus descendientes o se poda. De todos los nodos vivos, se escoge uno (el primero, el mejor, ...).

Si en un problema Greedy da el óptimo, Branch & Bound también lo hará.

# 2. Relaciones de Problemas