# Práctica 5 - Backtracking y Branch & Bound

*Memoria de práctica*


Germán Castilla López
Jorge Gangoso Klöck
Pedro Morales Leyva
Clara Mª Romero Lara

# Índice de Contenidos

# 1. Descripción general de la práctica

Esta práctica se centra en los algoritmos *branch and bound* (ramificación y poda) y *backtracking* (vuelta atrás). Los problemas a considerar son los siguientes:

Un problema común, **el problema del viajante de comercio**. Lo resolveremos por *backtracking* y *branch and bound* y realizaremos una comparación empírica.

Un problema elegido aleatoriamente, el **2.1: laberinto**. Este problema incluye la generación de los datos para el mismo, y su resolución mediante backtracking.

# 2. Problema del viajante de comercio

## 2.1. Descripción

### Branch & Bound

Como ya hemos visto en prácticas anteriores, el problema del viajante de comercio (TSP) consiste en hallar un ciclo hamiltoniano sobre un grafo no dirigido, en este caso utilizando una técnica de ramificación y poda *(Branch and Bound)*.

Para nuestra solución nos hemos apoyado principalmente en el uso de un `struct Nodo`, el cual nos almacenará para cada nodo del árbol que expandamos:

- El índice de la profundidad que representa: cuántas ciudades exploradas lleva
- La ruta empleada para llegar hasta él, siendo él mismo el último elemento
- El coste de dicha ruta.

Emplearemos el coste de cada nodo y una función auxiliar que calcula los arcos de menor coste entre las ciudades restantes para generar nuestra función de **Cota Local**. Con ella, podremos decidir si una rama tiene o no posibilidades de mejorar el resultado actual, y en caso de que no pueda, ignorar esta rama (podar).

Para explorar las ramas no podadas, simplemente generamos los Nodos hijo posibles a partir de un Nodo actual, es decir, el camino de la ciudad actual a cualquiera de las otras ciudades sin explorar aún.

### Backtracking

La versión backtracking consiste básicamente en realizar una búsqueda exhaustiva en profundidad hasta, o bien llegar al final y guardar el resultado como una posible solución (si es la mejor hasta el momento), o bien llegar a un punto de poda en el que no es necesario continuar.

En ambos casos, mientras sea posible, se volverá al último punto de inflexión disponible para escoger la siguiente alternativa.

Para ello, hemos optado por convertir la cola de prioridad en una pila, de ésta manera se explorará una rama hasta el final y, una vez no se pueda continuar, se retomará desde el último valor de la pila, es decir, el último punto de inflexión disponible.

El único inconveniente de éste sistema es que los Nodos hijo generados se quedan almacenados en memoria hasta que les llega el momento de ser explorados, mientras el flujo del programa recorre cada rama olvidándose de los nodos hijo de niveles superiores.

## 2.2. Pseudocódigo

```
TSP_BB()
{
Prio_queue Vivos <- inicial //Pila en el caso de backtracking
Cota_Global <- TSP_Cercanía
Cota_Local <- NULL
Ruta_solucion <- NULL

 mientras (vivos no vacío) hacer
 {
  actual <- Vivos.primero
  Vivos\primero

  Cota_Local <- mejor camino posible desde actual, idílico

  si fin de ciclo
  {
   si ruta_actual < Cota_Global
    {
    Cota_Global <- ruta_actual
    Ruta_solucion <- ruta_actual
    }
  }
  si criterio de poda (Cota_Local > Cota_Global)
  {
    podar() //No explorar el nodo actual y por tanto ninguna de
sus ramificaciones
  }
  si no criterio de poda ni fin de ciclo
  {
   Vivos <- actual.generarHijos()
  }

devolver Ruta_solucion
}
```

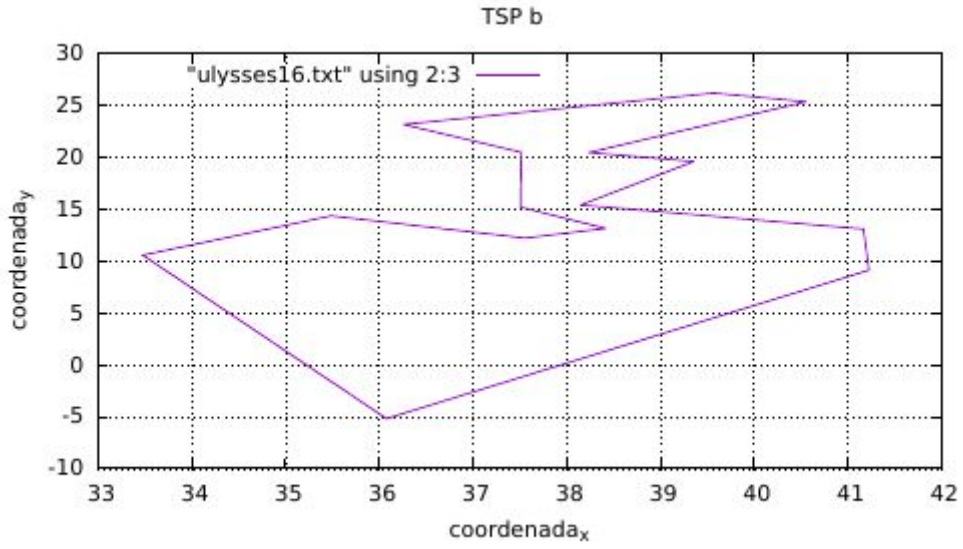
## 2.3. Supuesto de ejecución

Siguiendo un programa que se ajusta al pseudocódigo expuesto anteriormente, realizamos la ejecución con el archivo de datos `"Ulysses16.tsp"`.

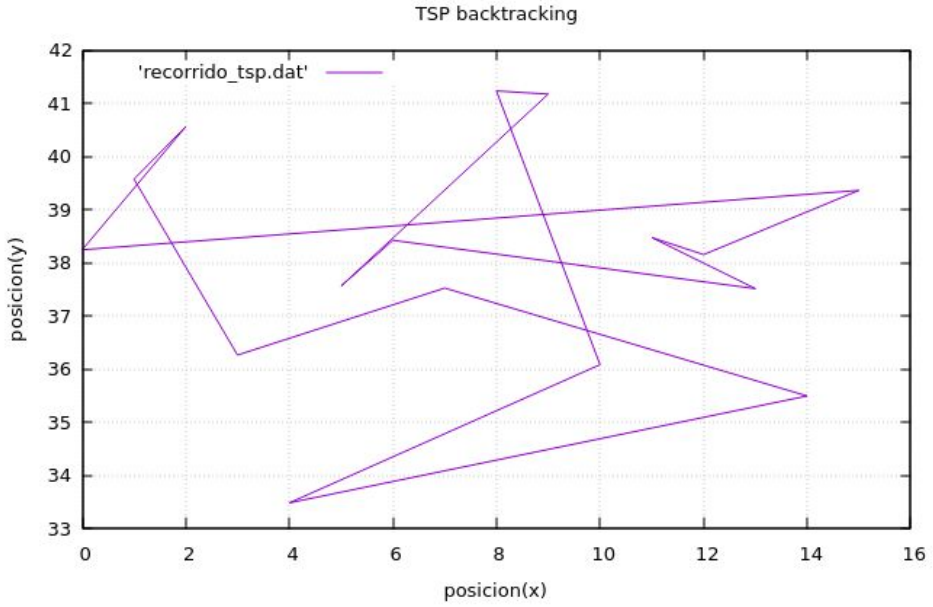
El programa devolverá el camino alcanzado, el coste de dicho camino, la cantidad de nodos expandidos en el proceso, el tamaño máximo que ha llegado a tener la cola de abiertos, la cantidad de podas realizadas y el tiempo empleado en realizar el procedimiento de B&B o backtracking.

***Nota:*** *cuando hicimos las gráficas del recorrido, no nos percatamos de que usamos dos archivos Ulysses16 diferentes: estos poseen el mismo coste óptimo y son muy similares, pero existen ciertas diferencias entre ambos archivos. Lamentamos el error.*

### Branch & Bound



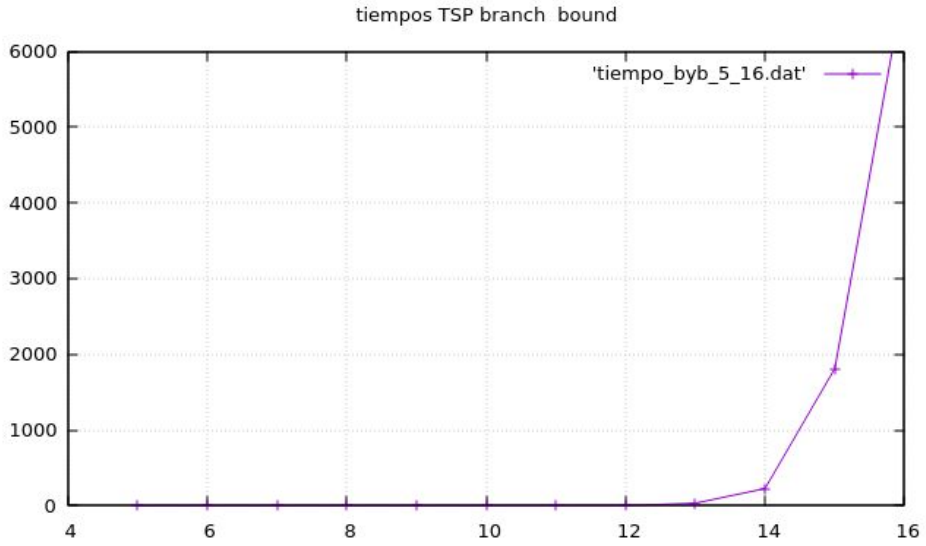

### Backtracking



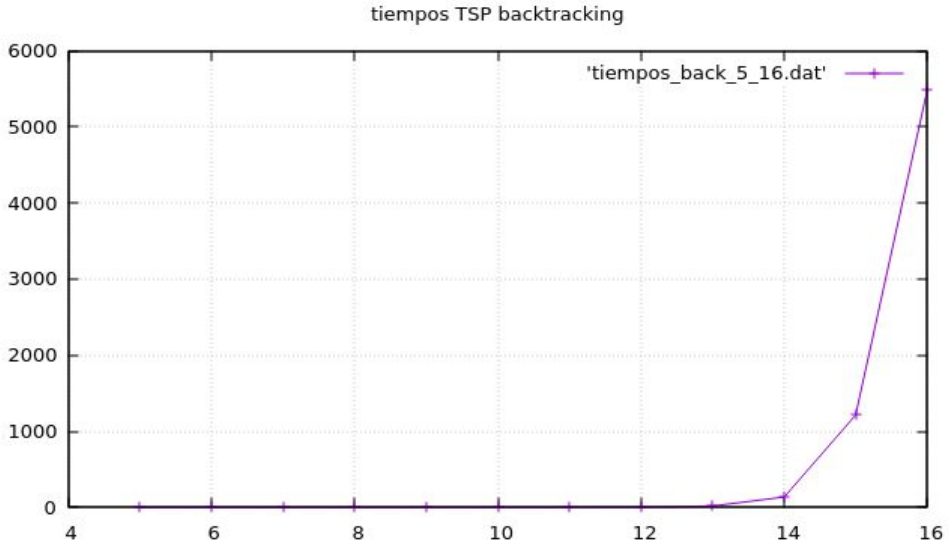

## 2.4. Gráficas

Con el objeto de realizar una comparación empírica, tomamos el archivo `"Ulysses16.tsp"` y lo modificamos retirando siempre la última ciudad, teniendo versiones de 5 a 16 ciudades.

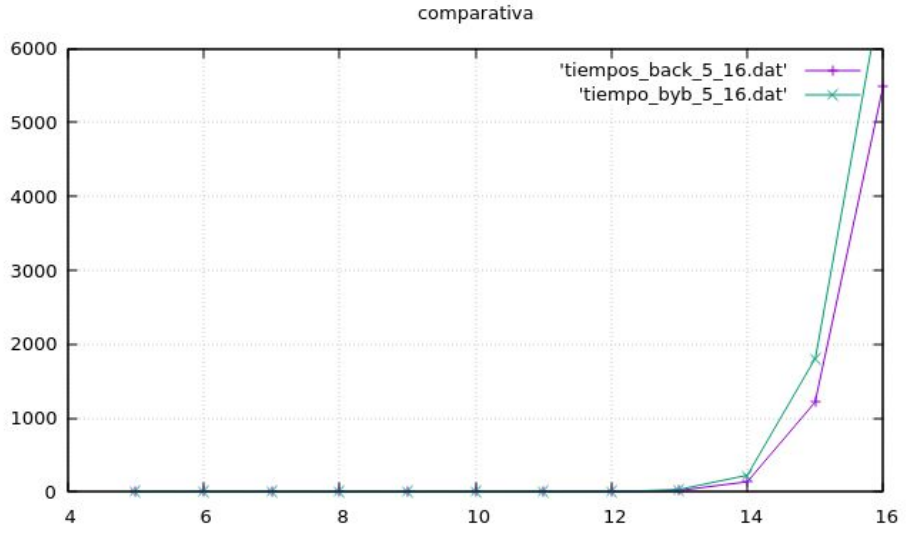
## Branch & Bound




## Backtracking




Como podemos comprobar, los tiempos de ejecución escalan muy rápido al pasar de las 14 ciudades, pero crecen de forma muy similar. Si comparamos las gráficas, veremos qué branch and bound es ligeramente más eficiente que backtracking:



# 3. Laberinto

El problema del laberinto consiste en, dada una matriz nxn, donde los 0 son casillas transitables, y los 1 son paredes; empezando en el (0,0) y acabando en en el (n-1,n-1), obtener el recorrido del laberinto por backtracking.

## 3.1. Descripción

Para recorrer el laberinto hemos creado una función llamada `buscarRuta`, a la que se le pasa la posición de la fila y de la columna en la que se empieza ((0,0) siempre) y devuelve `true` en caso de que se encuentre un camino, y `false` en el caso contrario.

En esta función, primero se comprueba si las posiciones pasadas están dentro de la matriz. En caso afirmativo, se comprueba si la posición es transitable (0). Si se puede transitar, asigna esa posición como ruta (con el valor 2), y lo añade a un vector que contiene la ruta final. Tras esto, se comprueba si la posición actual es el final de la matriz. Si es el final, se devuelve un `true`, y si no, empezamos suponiendo que sí habrá un camino, y acto seguido llamamos recursivamente a la función cuatro veces, una por cada casilla vecina a la casilla actual. Si alguna de estas llamadas devuelve un `false`, es decir, no hay camino posible, el resultado final es un `false`.

La salida del programa son todas las casillas que se recorren para ir desde el punto inicial al final del laberinto.

## 3.2. Pseudocódigo

```
buscarRuta(posActual){
   if(fuera de la matriz)
       res = false
   else
       posActual = parte de la ruta
       rutaFinal.add(posActual)
       if(es el final)
          res = true
       else
          res = true
          if(!buscaRuta(vecinos))
              res = false
return res
```

## 3.3. Generación del problema

Para la generación del laberinto hemos optado por una matriz de enteros 7x7 generada recursivamente. Las características del laberinto son:

- Es un **laberinto perfecto**: no contiene ni ciclos ni rutas inalcanzables
- El 0 representa un espacio transitable y el 1 un espacio no transitable. Durante la implementación, el 3 corresponde a las puertas generadas *(no confundir con el 3 de la salida, que se corresponde con la ruta errónea)*
- Las paredes verticales se generan en las posiciones impares del eje x
    - Las puertas en las paredes verticales se generan en posiciones impares del eje y
- Las paredes horizontales se generan en las posiciones pares del eje y
    - Las puertas en las paredes horizontales se generan en posiciones pares del eje x

  De esta forma obtenemos un laberinto perfecto: las paredes horizontales no taparán las puertas verticales, y viceversa.

El laberinto se genera creando paredes y abriendo una puerta en cada una de las paredes que ponemos. La función `generaLaberintoRecursivo(laberinto, posX_izda, posX_dcha, posY_arr, posY_abj)` funciona así:

1. Primero se hacen comprobaciones para el caso base. Este puede darse bien por ya tener un pasillo no divisible (`ancho` o `alto` son menores que 2), o por habernos salido de la matriz (las cotas inferiores son menores que 0 o las superiores mayores que `n`).
    a. Si se da un caso base, nos aseguramos de que el inicio en (0,0) y la salida en (n,n) del laberinto están despejadas, y les asignamos una puerta (valor 3).
2. Si no es un caso base, llamamos a la función `escogeOrientacion`, la cual decide la orientación de la pared a colocar. Se favorece donde haya más espacio, si en ancho o en alto. Si ancho y alto son iguales, se escoge aleatoriamente.
3. Comprobamos la orientación. En cualquier caso (pared horizontal o pared vertical), el procedimiento a seguir será el mismo:
    a. Buscamos la posición para la nueva pared. Además de ajustarse a las cotas proporcionadas por `posX_izda, posX_dcha, posY_arr, posY_abj`, tenemos que comprobar que la nueva pared no tapará ninguna puerta.
    b. A la posición obtenida para la pared, le sumamos el desplazamiento provocado por la cota superior (`posX_izda` o `posY_arr`).
    c. Dibujamos la pared con `ponParedHorizontal` o `ponParedVertical`.
    d. Creamos una puerta con `abrePuertaHorizontal` o `abrePuertaEnVertical`.
    e. Finalmente, llamamos dos veces recursivamente a `generaLaberintoRecursivo`, acotando las dos nuevas “mitades” que ha generado la pared que acabamos de colocar.

## 3.4. Supuesto de ejecución

Compilamos y ejecutamos el programa *(en este caso hemos compilado desde la terminal, pero con la práctica se adjunta un makefile completo)*


El programa imprime el laberinto generado...

```
0101010
0100000
0101110
0100010
0101111
0000000
0101010
```

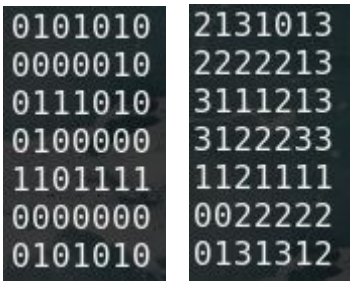
… la ruta completa (todas las casillas que ha visitado en orden)...

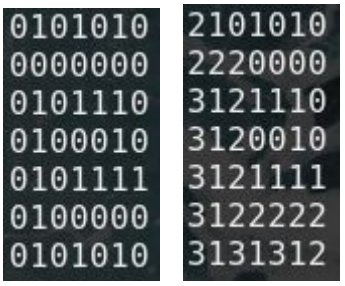
```
0, 0
1, 0
2, 0
3, 0
4, 0
5, 0
6, 0
5, 1
5, 2
6, 2
4, 2
3, 2
2, 2
1, 2
0, 2
1, 3
1, 4
0, 4
1, 5
1, 6
2, 6
3, 6
0, 6
3, 3
3, 4
5, 3
5, 4
6, 4
5, 5
5, 6
6, 6
```

… y finalmente, imprime el laberinto resuelto. El 2 marca la ruta correcta, y el 3 las casillas visitadas, pero que no llevan a la salida:

```
2131313
2133333
2131113
2133313
2131111
2222222
3131312
```

En este supuesto particular, no se ha dejado ninguna casilla por visitar, pero en los siguientes ejemplos podemos comprobar que se puede dar el caso de que una casilla no se visite:




# 4. Conclusiones

Las conclusiones que hemos inferido de esta práctica son las siguientes:

- Como hemos podido comprobar, aunque los programas basados en *Backtracking* pueden llegar a ser menos eficientes en tiempo que otros algoritmos debido a las vueltas y repeticiones que conllevan, son un algoritmo que cumple con lo que se pide y que encuentra soluciones óptimas.
- Respecto a la ramificación y poda, se trata de un algoritmo realmente potente en cuanto a obtener soluciones óptimas en espacios de problema de tamaño moderado, pero conlleva un costo de tiempo proporcional que hace que sea inviable para espacios más amplios.
- La versión Backtracking puede acelerar el proceso en algunos casos pero depende de la distribución del árbol y de la “suerte” a la hora de podar ramas. Escoger la mejor ruta es algo que ya entra en el campo de la Inteligencia Artificial.