# UNIVERSIDAD DE GRANADA
# E.T.S. DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

**Asignatura:** Algorítmica

**Grupo nº:** 4

**Integrantes:** Raúl Rodríguez Pérez, Francisco Javier Gallardo Molina, Inés Nieto Sánchez, Antonio Lorenzo Gavilán Chacón

**Grupo:** C1

# Resolución Práctica 5: Algoritmos de Vuelta Atrás (Backtracking) y de Ramificación y Poda (Branch and Bound)

**Curso 2019-2020**

**Grado en Ingeniería Informática**

# Índice:

# 1. Introducción TSP

El problema del viajante de comercio ya se ha abordado en prácticas anteriores. Resumiendo, el problema trata de que dado un conjunto de ciudades y sus coordenadas en un mapa, obtenemos una matriz con las distancias entre todas ellas, de modo que un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima.
Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

Se ha resuelto este problema de dos formas diferentes, en primer lugar con el algoritmo de Ramificación y Poda (Branch and Bound), y en segundo lugar con el de Vuelta Atrás (Backtracking). Además hemos empleado la misma función para calcular tanto la cota superior como la inferior, en los dos algoritmos(se explicará dicha función con más detalle posteriormente).

En ambos casos se ha utilizado el algoritmo de Greedy para establecer una cota global superior. Sin embargo, queremos destacar algo que hemos comprobado al ejecutar el algoritmo con un número reducido de ciudades. Al no poder emplear los datos que nos ofrece nuestro profesor (los cuales poseen un gran número de ciudades) debido a que los dos algoritmos comentados no pueden tener muchos datos de entrada, ya que se ralentiza sus ejecuciones debido al gasto de recursos que poseen ambos algoritmos. Nos vimos obligados a realizar el estudio con un número reducido de ciudades (entre 6 y 12 ciudades), pero esto implicaba que en los casos en los que el número de ciudades era muy reducido, habían ejecuciones en las que el algoritmo Greedy nos devolvía la distancia óptima del problema, por lo que nuestros algoritmos no ejecutaban correctamente debido a que no se estaba inicializando la cota con un valor mayor estricto que la óptimo (cota_global > distancia), siendo necesario que esto se cumpla para poder encontrar dicha solución óptima en el árbol de estados.

Así que para evitar este problema, en los casos de ejecución donde empleamos un número reducido de ciudades en ambos algoritmos, hemos cambiado esta cota superior por el máximo valor que puede proporcionar un dato de tipo “double”, ya que también este valor puede funcionar perfectamente como en el caso del algoritmos de Greedy.

## 2. Algoritmo de Branch and Bound (B&B)

Este algoritmo es también llamado de **Ramificación y Poda**, destacamos que vamos a utilizar una versión simplificada del mismo, en la que para cada nodo sólo se usa una **cota local inferior**, ya que este es un problema de minimización. El valor de esta cota local es la que se utiliza, además de para podar (comparándolo con la **cota global**), para decidir cuál es el mejor nodo para expandir. Interesa que la cota sea precisa (no demasiado optimista).

Ya que un ciclo debe pasar exactamente una vez por cada ciudad, un **estimador** (cota inferior) de la longitud del ciclo se obtiene al considerar la mínima distancia de salir de cada ciudad. Es el mínimo valor de todas las entradas no nulas de la matriz de distancias. La suma de estas distancias más el coste del camino ya acumulado, es una cota inferior en el sentido antes descrito. Para realizar la poda, también es necesario una **cota global**, que representa el valor de la distancia total de la mejor solución estudiada hasta el momento. Inicialmente le damos el valor dado por el algoritmo de Greedy de cercanías visto en una práctica anterior. Esta cota global es una cota superior de la solución optimal. La cota global se actualiza siempre que se alcance una solución (nodo hoja) con un valor menor de la distancia total. De este modo se poda una rama siempre que para una solución parcial, su cota inferior es mayor que la cota global.

El código del algoritmo de B&B que hemos utilizado en esta práctica se basa en el método declarado como sigue:

**Solucion Branch_and_Bound (const vector< vector<double> > & matriz, const vector<double> & arista_menor, int tama).**

Como vemos, tiene como parámetros a la matriz de distancias, el vector “arista_menor” que contiene la mínima distancia de salir de cada ciudad y la variable entera “tama” que se refiere al número de ciudades que hay en total. También se observa que este método devuelve un objeto de la clase “Solución”, la cual es necesaria crearla para estructurar mejor el contenido del código y en la que se hacen todos los cálculos necesarios para el funcionamiento del algoritmo.

La clase “Solución” contiene el vector solución con las ciudades ordenadas y la distancia total calculada para ese vector. Para guardar los nodos ya generados (nodos vivos) se utiliza una cola con prioridad de elementos de la clase “Solución” con el criterio LC o “más prometedor” donde la comparación ordena de menor a mayor según la cota local obtenida para cada nodo.

El pseudocódigo del algoritmo es el siguiente:

```
Algoritmo B&B (matriz, arista_menor, tama) {
    priority_queue<Solucion> Q;
    Solucion n_e, mejor_solucion;
    mejor_solucion.Greedy(matriz);
    CG = mejor_solucion.getDistancia();
    Q.push(n_e);
        while (!Q.empty && Q.top().cota_local() < CG) {
            n_e = Q.top();
            aux = n_e.resto_ciudades();
            for (int i = 0; i < aux.size(); i++) {
                n_e.añadir_ciudad (aux[i], matriz);
                n_e.quitarCiudadRestante (aux[i]);
                if (n_e.Es_Solucion()) {
                    distancia_actual = n_e.Evalua();
                    if (distancia_actual < CG) {
                        CG = distancia_actual;
                        mejor_solucion = n_e;
                    }
                else {
                    if (n_e.Cota_Local < CG)
                        Q.push(n_e);
                }
                n_e.quitarCiudad (matriz);
                n_e.añadirCiudadRestante (aux[i]);
            }
        }
    return mejor_solucion;
}
```

Como se ve en el pseudocódigo, el algoritmo primero crea dos objetos de la clase “Solución” y añade a uno de ellos la solución Greedy del problema, que lo toma como cota superior (cota global). Introduce el otro objeto inicializado a la cola con prioridad de objetos “Solución” y comprueba si la cola no está vacía y que la cota local del nodo en cuestión es menor que la cota global. Si se cumple esto último se introduce en un bucle en el que comprueba que si ya se han visto todas las ciudades (función “Es_Solución()”) se evalúa la distancia actual (función “Evalua()) y si esta es menor que la cota global, se actualiza esta y se toma como mejor solución el objeto estudiado. Y si no es así se añade el objeto actual a la cola con prioridad, en el caso de que la cota local sea menor que la cota global. Al final el algoritmo devuelve la mejor solución.

## 3. Eficiencia de B&B

El esquema de Ramificación y Poda es la manera más eficiente que se conoce de recorrer un espacio de búsqueda que sea un árbol pero hay elementos que lo encarecen. Como hace un recorrido completo, en el peor de los casos del

espacio de búsqueda, el coste es del **orden exponencial**, teniendo en cuenta además que la manipulación de la cola con prioridad de nodos vivos hace que el coste aumente.

La efectividad de este algoritmo depende de la calidad de la función de estimación y además es bueno disponer de una solución inicial para aumentar la eficiencia, que en nuestro caso es la que proporciona el algoritmo de Greedy.

Por lo tanto el tiempo de ejecución depende de:

- Número de nodos recorridos: depende de la efectividad de la poda.
- Tiempo gastado en cada nodo: tiempo de hacer las estimaciones de coste y tiempo de manejo de la lista de nodos vivos.

En el caso promedio se suelen obtener mejoras respecto al Backtracking, pero en el peor de los casos se generan tantos nodos como en Backtracking, incluso el tiempo puede ser peor según lo que se tarde en calcular las cotas y manejar la lista de nodos vivos. Por lo tanto este algoritmo tiene una **complejidad exponencial tanto en tiempo como en uso de memoria**.

Aún así existen algunos métodos que podemos realizar para hacerlo más eficiente:

- Hacer estimaciones y cotas muy precisas, consiguiendo una poda muy exhaustiva del árbol. Se recorren menos nodos pero se tardará mucho en hacer estimaciones.
- Hacer estimaciones y cotas poco precisas, en la que no se hace mucha poda. Se gasta poco tiempo en cada nodo, pero el número de nodos es muy elevado.

Se debe buscar un equilibrio entre la exactitud de las cotas y el tiempo de calcularlas.

## 4. Casos de ejecución de B&B

Se ha procedido a ejecutar el algoritmo de B&B para el archivo de ciudades “ulysses16.tsp” tomando 8, 10 y 12 ciudades. Si se tomaran más ciudades el tiempo de ejecución se alargaría demasiado. Destacar que se emplearán los mismo archivos para realizar los casos de ejecución de backtracking por lo que detalles como la matriz de distancias, o el gráfico del recorrido sólo se mostrarán a continuación para no repetir las mismas imágenes debido a que coinciden en ambos algoritmos.

### a. Caso ulysses8

Para el caso de tener 8 ciudades la matriz de distancias obtenidas es la siguiente:

```
la matriz de distancias es:

-1      5.8823  5.4215  3.3482  10.967  8.258   7.3122  0.72028
5.8823  -1      1.2919  4.4874  16.756  14.104  13.091  6.0668
5.4215  1.2919  -1      4.8301  16.388  13.468  12.396  5.7494
3.3482  4.4874  4.8301  -1      12.884  11.007  10.24   2.9614
10.967  16.756  16.388  12.884  -1      4.401   5.5685  10.693
8.258   14.104  13.468  11.007  4.401   -1      1.2594  8.2501
7.3122  13.091  12.396  10.24   5.5685  1.2594  -1      7.385
0.72028 6.0668  5.7494  2.9614  10.693  8.2501  7.385   -1
```

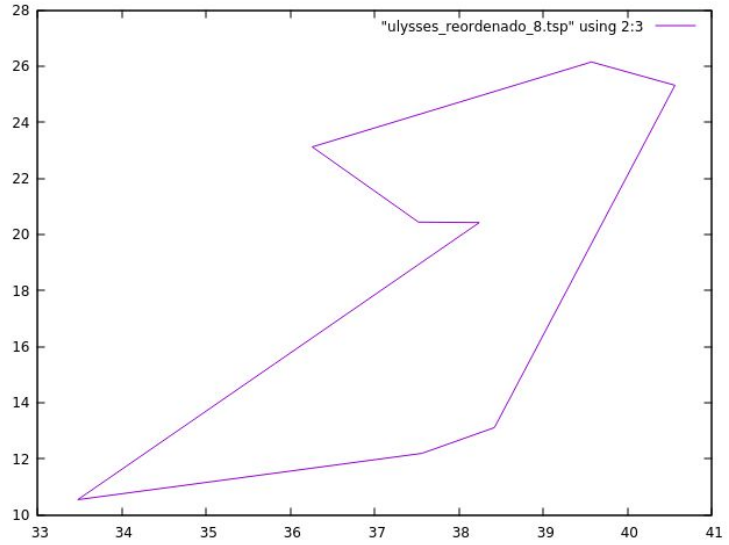
Como se observa en la imágen anterior los valores para la distancia entre la misma ciudad se han tomado con el valor -1. Por otro lado, la secuencia de ciudades resultado obtenidas junto con la distancia total y el tiempo empleado es el siguiente:

```
1 38.24 20.42
7 38.42 13.11
6 37.56 12.19
5 33.48 10.54
8 37.52 20.44
4 36.26 23.12
2 39.57 26.15
3 40.56 25.32
1 38.24 20.42

distancia: 37.827

Tiempo empleado: 0.022161 seg
```

Por último los resultados relativos a complejidad: número de nodos expandidos, tamaño máximo de la cola con prioridad de nodos vivos y número de veces que se realiza la poda son los de la imagen de abajo:

```
Numero de nodos expandidos: 1869
Tamanio maximo de la cola con prioridad de nodos vivos: 714
Numero de podas realizadas: 1883
```

El resultado después de aplicar el algoritmo se puede ver en el recorrido de la imagen siguiente:




### b. Caso ulysses10

Para el caso de tener 10 ciudades la matriz de distancias obtenidas es la siguiente:

```
la matriz de distancias es:
-1      5.8823  5.4215  3.3482  10.967  8.258   7.3122  0.72028 11.708  7.9311
5.8823  -1      1.2919  4.4874  16.756  14.104  13.091  6.0668  17.131  13.197
5.4215  1.2919  -1      4.8301  16.388  13.468  12.396  5.7494  16.234  12.285
3.3482  4.4874  4.8301  -1      12.884  11.007  10.24   2.9614  14.875  11.203
10.967  16.756  16.388  12.884  -1      4.401   5.5685  10.693  7.8826  8.0893
8.258   14.104  13.468  11.007  4.401   -1      1.2594  8.2501  4.7976  3.711
7.3122  13.091  12.396  10.24   5.5685  1.2594  -1      7.385   4.8965  2.7507
0.72028 6.0668  5.7494  2.9614  10.693  8.2501  7.385   -1      11.931  8.2422
11.708  17.131  16.234  14.875  7.8826  4.7976  4.8965  11.931  -1      3.9505
7.9311  13.197  12.285  11.203  8.0893  3.711   2.7507  8.2422  3.9505  -1
```

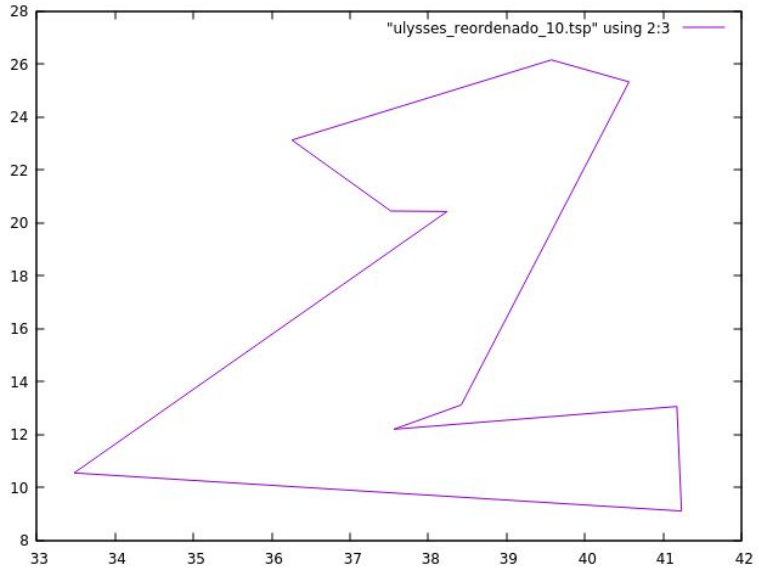
La secuencia de ciudades resultado obtenidas junto con la distancia total y el tiempo empleado es el que se ve a la derecha. Destacamos que el tiempo empleado es bastante superior al caso de tener 8 ciudades:

```
1 38.24 20.42
8 37.52 20.44
4 36.26 23.12
2 39.57 26.15
3 40.56 25.32
10 41.17 13.05
9 41.23 9.1
5 33.48 10.54
6 37.56 12.19
7 38.42 13.11
1 38.24 20.42

distancia: 46.552

Tiempo empleado: 0.5126 seg
```

Los resultados relativos a complejidad: número de nodos expandidos, tamaño máximo de la cola con prioridad de nodos vivos y número de veces que se realiza la poda son los que se ven en la imagen de abajo:

```
Numero de nodos expandidos: 33426
Tamanio maximo de la cola con prioridad de nodos vivos: 13536
Numero de podas realizadas: 50195
```

Igualmente se produce un incremento considerable al añadir dos ciudades en el número de nodos expandidos y podas realizadas. El resultado después de aplicar el algoritmo se puede ver en el recorrido de la imagen siguiente:




## c. Caso ulysses12

Para el caso de tener 12 ciudades la matriz de distancias obtenidas es la siguiente:

```
la matriz de distancias es:
-1      5.8823  5.4215  3.3482  10.967  8.258   7.3122  0.72028 11.708  7.9311 25.721  5.295
5.8823  -1      1.2919  4.4874  16.756  14.104  13.091  6.0668  17.131  13.197 31.554  11.075
5.4215  1.2919  -1      4.8301  16.388  13.468  12.396  5.7494  16.234  12.285 30.857  10.402
3.3482  4.4874  4.8301  -1      12.884  11.007  10.24   2.9614  14.875  11.203 28.331  8.29
10.967  16.756  16.388  12.884  -1      4.401   5.5685  10.693  7.8826  8.0893 15.963  6.78
8.258   14.104  13.468  11.007  4.401   -1      1.2594  8.2501  4.7976  3.711  17.463  3.0776
7.3122  13.091  12.396  10.24   5.5685  1.2594  -1      7.385   4.8965  2.7507 18.469  2.0206
0.72028 6.0668  5.7494  2.9614  10.693  8.2501  7.385   -1      11.931  8.2422 25.69   5.3943
11.708  17.131  16.234  14.875  7.8826  4.7976  4.8965  11.931  -1      3.9505 15.209  6.6316
7.9311  13.197  12.285  11.203  8.0893  3.711   2.7507  8.2422  3.9505  -1     18.956  3.4083
25.721  31.554  30.857  28.331  15.963  17.463  18.469  25.69   15.209  18.956 -1     20.48
5.295   11.075  10.402  8.29    6.78    3.0776  2.0206  5.3943  6.6316  3.4083 20.48   -1
```

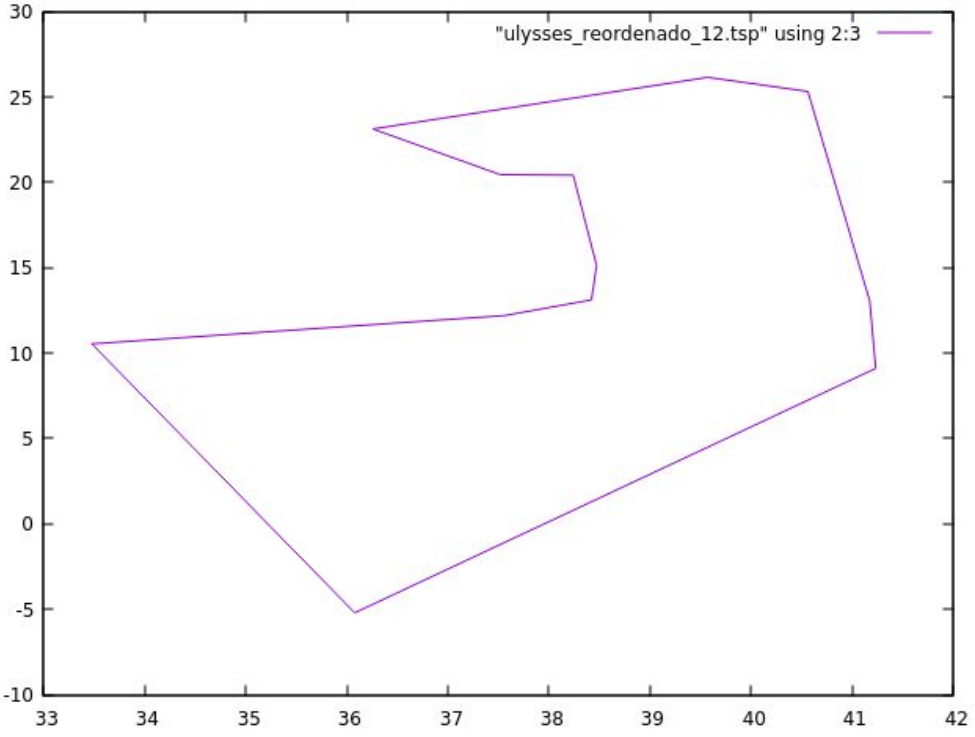
La secuencia de ciudades resultado obtenidas junto con la distancia total y el tiempo empleado se puede ver en la imagen a la derecha. Se observa que el tiempo empleado aumenta considerablemente. En este tipo de algoritmo no se puede ejecutar con muchos datos.

```
1 38.24 20.42
8 37.52 20.44
4 36.26 23.12
2 39.57 26.15
3 40.56 25.32
10 41.17 13.05
9 41.23 9.1
11 36.08 -5.21
5 33.48 10.54
6 37.56 12.19
7 38.42 13.11
12 38.47 15.13
1 38.24 20.42

distancia: 69.844

Tiempo empleado: 18.476 seg
```

Los resultados relativos a complejidad: número de nodos expandidos, tamaño máximo de la cola con prioridad de nodos vivos y número de veces que se realiza la poda son los siguientes:

```
Numero de nodos expandidos: 893537
Tamanio maximo de la cola con prioridad de nodos vivos: 362234
Numero de podas realizadas: 1759920
```

El resultado después de aplicar el algoritmo se puede ver en el recorrido de la imagen siguiente:



## 5. Algoritmo de Backtracking (‘vuelta atrás’)

En segundo lugar vamos a proceder a explicar como hemos elaborado la implementación del problema del viajante del comercio, pero esta vez con un enfoque basado en backtracking. Este enfoque se basa en encontrar la mejor combinación posible en un momento determinado, por eso, se dice que este tipo de algoritmo está muy relacionado con una búsqueda en profundidad. La diferencia con la búsqueda en profundidad es que en backtracking se suelen diseñar funciones de cota, de forma que no se generen algunos estados (o que no se expandan más nodos) si no van a conducir a ninguna solución, o a una solución peor de la que ya se tiene. De esta forma se ahorra espacio en memoria y tiempo de ejecución.

Durante la búsqueda, si se encuentra una alternativa incorrecta, la búsqueda retrocede (“vuelve atrás”) hasta el paso anterior y toma la siguiente alternativa. Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción (por ejemplo en el caso de un árbol, sería el siguiente hijo). Si no hay más alternativas la búsqueda falla. De esta manera, se crea un árbol implícito, en el que cada nodo es un estado de la solución (solución parcial en el caso de nodos interiores o solución total en el caso de los nodos hoja).

Vamos a proceder a explicar nuestro código mediante el siguiente pseudocódigo:

```
void backtracking(ciudad_actual, visitadas, matriz_distancias, cota_real, cota_opt, min_cota, sol_final)
{

        visitadas.add(ciudad_actual)   //añadimos la primera ciudad

        if(visitadas == numero_ciudades){   //caso base
                for(...)
                  distancia += matriz_distancia[][];  //calculamos la distancia del recorrido de ciudades
                                                        visitadas
                if(cota_global > distancia){
                        //actualizamos la distancia global con el valor de distancia
                        //actualizamos sol_final con el recorrido de ciudades de visitadas
                }
        }
        else{
                cota_local = cota_real + cota_opt      //calculamos la cota local
                if(cota_local < cota_global){
                        for(...; i < numero_ciudades; ...){
                                if('i' no se encuentra en 'visitadas'){
                                        //calculamos la cota real
                                        //calculamos la cota optimista
                                        backtracking(i, ..., ..., ...) //volvemos a llamar recursivamente
                                                                          a la funcion
                                        visitadas.pop_back()
                                }
                        }
                }
        }
}
```

Como normalmente se procede, hemos implementado este tipo de algoritmo como un procedimiento recursivo. A dicha función le pasamos diversos parámetros; en primer lugar tenemos la ciudad_actual, la cual representa la ciudad desde donde se partirá a realizar el estudio de los diversos recorridos. En segundo lugar una lista donde almacenamos las ciudad que vayamos visitando. En tercer lugar, la matriz que almacena las diversas distancias entre nuestras ciudades. A continuación le pasamos tanto la cota real como la cota optimista, valores que utilizaremos posteriormente para calcular la cota local (después entraremos en más detalle con esto). Y finalmente le pasamos la lista donde almacenaremos el recorrido final, nuestra solución del problema.

Una vez aclarado los parámetros vamos a entrar en materia explicando el funcionamiento del algoritmo. Antes de nada añadimos a nuestra lista de visitadas la ciudad actual puesto que será la ciudad de la que partiremos para analizar las diversas combinaciones. Tras esto, y como es indispensable en cualquier implementación recursiva, necesitamos definir un caso base. En nuestro algoritmo, dicho caso base se trata de que la lista de ciudades visitadas tenga el mismo tamaño que el número de ciudades que tenemos. Sí esto se cumple procederemos a, en primer lugar, calcular la distancia del recorrido de ciudades que obtenemos de la lista de visitadas. Una vez calculada la distancia debemos compararla con la cota global/superior que hemos establecido (como hemos explicado en la implementación anterior dicha cota se inicializa con el valor de la distancia que nos otorga el recorrido obtenido con un enfoque greedy y la interpretaremos como el valor de la mejor solución que tenemos de momento). Sí dicha cota es mayor, debemos entonces actualizarla con el valor de la distancia, debido a que esto significa que hemos alcanzado una nueva solución mejor que la que teníamos hasta el momento. Y, además de esto, debemos asignar el recorrido de ciudades visitadas a nuestra sol_final.

Por otro lado sí no satisfacemos el caso base, procederemos a calcular el valor de la cota local, que no es otra cosa que el mejor valor que se podría alcanzar al expandir un nodo. Dicha cota se calcula con la suma de la cota real y optimista. La cota optimista la calculamos considerando el mínimo costo de salir de cada vértice o ciudad. Esto lo hemos calculado previamente, es decir, hemos calculado y almacenado en un vector ‘min_cota’ el mínimo costo que tendriamos de salir de cada ciudad antes de llamar a la función backtracking. Así nos ahorramos repetir cálculos y solo tenemos que acceder a la información del vector que pasamos como parámetro para calcular la cota optimista. Por otro lado la cota real…

A continuación nos metemos en un bucle para poder explorar todo el espacio de estados. Dentro de dicho bucle, debemos buscar sí la ciudad que estamos analizando se encuentra ya añadida a la lista de ciudades visitadas. Sí no se encuentra, procederemos a calcular la cota real y optimista tal y como hemos comentado anteriormente. Tras esto y finalmente volvemos a llamar a nuestra

función backtracking, destacando que como ciudad actual se pasará como parámetro la ciudad que estemos evaluando en el bucle, y pasaremos los valores de las cotas que hemos calculado anteriormente.

## 6. Eficiencia Backtracking

Para hablar de la eficiencia de backtracking debemos tener en cuenta que esta depende de 4 factores:

1) El primero se basa en el tiempo necesario para generar la siguiente componente de la solución. Normalmente este tiempo no es muy relevante ya que en la mayoría de casos, como en el nuestro, es un tiempo cte.
2) El número de valores que satisfacen las restricciones explícitas, es decir, esto se corresponde con el factor de ramificación del árbol de estados. Por ejemplo, si para cada uno de los componentes de la solución puedo asociarle k valores diferentes, entonces el factor de ramificación del árbol sería de k en cada nivel.
3) El tiempo de determinar la función de factibilidad, es decir, el tiempo que le tome al algoritmo evaluar sí un nodo es factible o no
4) El número de nodos que satisfacen la función de factibilidad. Se trata nada más del número de nodos que genere el algoritmo durante el proceso

En nuestro caso debemos tener en cuenta las funciones de acotaciones que hemos empleado, puesto que debemos evaluar sí estas son buenas o no. Las funciones de acotación se consideran buenas si reducen considerablemente el número de nodos que hay que explorar. Sin embargo, las buenas funciones de acotación suelen consumir mucho tiempo en su evaluación, por lo que hay que buscar un equilibrio entre el tiempo de evaluación de las funciones de acotación y la reducción del número de nodos generados. En nuestro caso pensamos que las funciones de acotaciones dadas tienen un equilibrio bastante aceptable, puesto que, por un lado la cota superior, aunque en un principio se pueda pensar que calcular su valor mediante un enfoque greedy puede aumentar considerablemente el tiempo, gracias a prácticas anteriores sabemos que dicho enfoque tiene un tiempo de ejecución muy rápido y no va a ser apreciable el tiempo perdido. Y por otro lado con las cotas reales y optimistas que hemos elegido para calcular la cota local o inferior, y la manera eficiente en la que las calculamos (por ejemplo guardando los valores en un vector para la optimista), también concluimos en que el tiempo que se pierde es despreciable.

Finalmente debemos destacar que de los 4 factores que determinan el tiempo de backtracking, solo el último varía de un caso a otro. Pudiendo darse diversos casos, por ejemplo, un algoritmo de backtracking podría en un caso concreto (considerado el mejor caso) generar solo O(n) nodos, mientras que en otros caso relativamente parecidos podría generarse casi todos los nodos del árbol de espacio

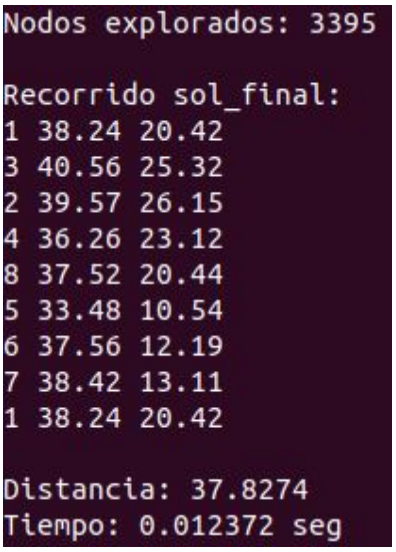
de estados. Entonces para el peor caso, sí el número de nodos en el espacio solución es n! (como es el caso del viajante de comercio) el algoritmo tendría un orden de eficiencia de O(q(n)*n!), siendo q un polinomio en n, como comentamos anteriormente, del mismo orden de eficiencia que B&B, del orden exponencial.

## 7. Casos de ejecución Backtracking

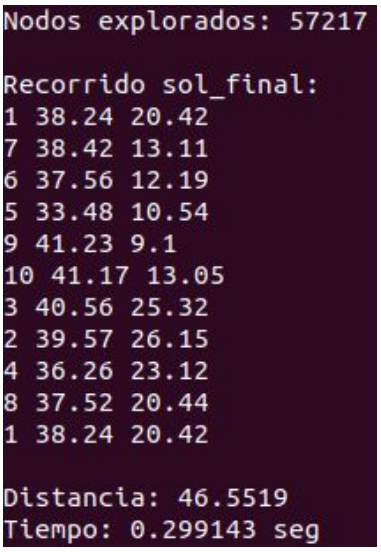
Como comentamos anteriormente en estos de casos de ejecución solo nos vamos a centrar en el recorrido final, la distancia y el número de nodos expandidos, ya que, tanto la matriz de distancias y el gráfico del recorrido de ciudades coinciden entre los dos algoritmos al devolver el resultado óptimo (es cierto que puede haber más de un resultado óptimo, pero en nuestro caso lo hemos comprobado y tanto la distancia como el recorrido de ciudades es el mismo).

Así que vamos a proceder a mostrar el recorrido y la distancia para los 3 casos de estudio, y realizaremos un pequeño análisis de estas:

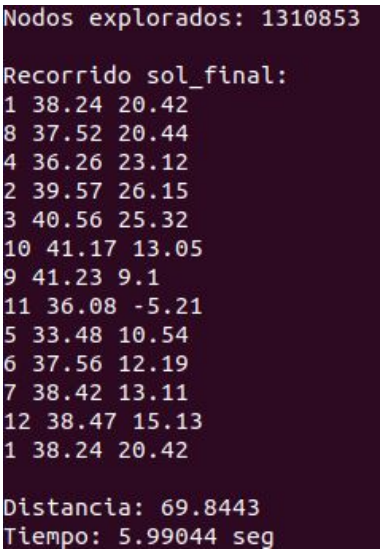
**Ulysses8**




**Ulysses10**




**Ulysses12**




Podemos apreciar que, en primer lugar, tal y como habíamos dicho coinciden el recorrido y la distancia con B&B puesto las tres son resultados óptimo. En segundo lugar, apreciamos que los tiempos son mejores que para el algoritmo de B&B, y además apreciamos un incremento bastante considerable en cuanto a los nodos explorados se refiere.

## 8. Comparación Branch and Bound - Backtracking

Como ya hemos comentado en los respectivos análisis de los algoritmos, ambos son algoritmos exactos, es decir, son capaces de calcular la solución óptima. Además como hemos visto con las eficiencias, ambos algoritmos tienen el mismo orden (exponencial) y su desempeño en uno u otro problema depende de pequeños factores previamente explicados. Así que para realizar una comparación entre ambos algoritmos vamos a usar el árbol de búsqueda de cada uno de ellos. Para

ello hemos realizado una tabla en donde mostramos el número de nodos expandidos para cada uno de los algoritmos con diversos conjuntos de datos:

| | **Backtracking** | **Branch and Bound** |
|---|---|---|
| **Ulysses6** | 199 | 99 |
| **Ulysses7** | 778 | 331 |
| **Ulysses8** | 3395 | 1869 |
| **Ulysses9** | 16640 | 9219 |
| **Ulysses10** | 57217 | 33426 |
| **Ulysses11** | 282268 | 194659 |
| **Ulysses12** | 1310853 | 893537 |
| **Ulysses13** | 8068748 | 5391542 |
| **Ulysses14** | 56058989 | 38452570 |

Como podemos apreciar el algoritmo de Branch and Bound siempre posee un número menor de nodos expandidos frente a backtracking que siempre posee un número mayor. Esto posiblemente se deba a la poda que realiza B&B, dándole una ventaja de no recorrer gran cantidad de nodos al poder podar una rama entera del árbol de estados, mientras que backtracking como mucho puede decidir expandir o no un nodo. Aún así no son todo ventajas para B&B, ya que debido a las comparaciones que debe hacer para realizar o no las podas, dicho algoritmo pierde un tiempo significativo tanto en las comparaciones como en la poda. Esto ocasiona lo que hemos comentado anteriormente en los casos de ejecución, que aunque el algoritmo de backtracking recorra más nodos, este encuentra la solución óptima en un tiempo de ejecución mejor que B&B para todos los diversos datos.

Para concluir queremos dejar en claro algo, en principio un algoritmo que utiliza backtracking sí no le añadimos ninguna función para establecer las cotas, dicho algoritmo recorre todas las posibles combinaciones hasta encontrar la óptima. Por otro lado B&B, no recorre todas las posibles combinaciones, es más, solo se queda con las soluciones que le interesan. Pero en este caso concreto, al haber empleado las mismas funciones para establecer cotas, estamos por un lado haciendo que B&B pode las ramas del árbol que no le interesan, o en donde no se encuentre la solución óptima. Y por otro lado estamos forzando a que backtracking

no expanda los hijos cuando las cotas nos digan que no se encuentra la solución óptima por ese camino.

## 1. Introducción transporte de mercancías

El siguiente ejercicio que hemos realizado se trata del transporte de mercancías. El problema del transporte de mercancías muestra que una empresa dispone de “n” centros de fabricación que tienen que abastecer a “n” puntos de ventas. Siendo i los centros de fabricación o distribución y j los puntos de ventas, se debe calcular la distancia entre los dos puntos, es decir, d(i,j). Un punto de distribución solo abastece a un punto de venta.

Se trata de diseñar una solución para establecer que la suma total de las distancias entre los puntos de distribución y los puntos de ventas sea el mínimo posible. Para poder realizar dicha tarea, podemos plantear el problema como un árbol para cada punto de inicio (nodo raíz) y cada hoja sería una posible solución. Teniendo en cuenta sólo un punto de partida, tendríamos una solución parcial, por lo que hay que considerar distintos puntos de partidas, es decir, un árbol de soluciones por cada punto de partida. La solución final será el valor mínimo de entre todas las soluciones parciales. Para llevar a cabo la solución a este problema, hemos implementado el código basándonos en un enfoque backtracking, para calcular todos los posibles casos y escoger el de menor valor.

## 2. Descripción del código

Para llevar a cabo la resolución a través de Backtracking de nuestro problema hemos implementado una función recursiva llamada “ backtracking “. A ésta se le pasan como parámetros el nivel en el que nos encontramos del árbol (cuyo valor inicialmente es 1), una variable de tipo int llamada venta_actual que representa la raíz del árbol y el primer punto de venta al que se distribuye, un vector de variables booleanas que indica si ya se ha distribuido a un punto de venta, dos vectores de tipo int para almacenar las soluciones que se van obteniendo y la solución final, la matriz de distancias y por último, la cota local.

Primeramente, sabiendo que en el nivel n se lleva a cabo la distribución del punto de distribución n-1 al punto de venta venta_actual, almacenamos en la posición nivel-1 el valor de venta_actual e indicamos en la posición venta_actual del vector distribuidos que ya se ha proveído a ese punto de venta.

Recorremos el árbol para el siguiente punto de distribución. Primero comprobamos que el punto de venta a analizar no ha sido todavía distribuido, en caso afirmativo pasamos al siguiente punto. En caso contrario, sumamos la distancia que hay de nuestro punto de distribución a ese punto de venta a la cota local. Si esta suma es menor a la cota global descendemos en profundidad para evaluar a sus hijos. Tras esto comprobamos si nos hallamos en un nodo hoja, si es así y la cota

local es menor a la global hemos encontrado una solución, si no volvemos a quedarnos con la cota_local antes de la suma para analizar otras posibilidades.
Este proceso lo seguiríamos haciendo hasta haber terminado de recorrer el árbol y haber obtenido la solución final. La función descrita anteriormente es llamada por getDistribuciónminima a la que se le pasan como parámetros la matriz de distancias, una referencia a la solución y una referencia al número de nodos explorados. Ésta nos devolverá la suma de las distancias mínima, así como la solución al problema y el número de nodos explorados.

En cuanto a los datos que hemos elegido, hemos usado matrices cuadradas. Nos dimos cuenta de que no tenían que ser simétricas, como ocurre en el problema del TSP, pues cada punto de distribución estaba a distancias diferentes de cada punto de venta.

## 3. Descripción del pseudocódigo

Esquema Recursivo

```
void back_recursivo(Solucion & Sol, int k)
{
if ( k == Sol.size())
   Sol.ProcesaSolucion();
else {
   Sol.IniciaComp(k);
   Sol.SigValComp(k);
   while (!Sol.TodosGenerados(k) {
     if (Sol.Factible(k))
         back_recursivo(Sol, k+1);
     Sol.SigValComp(k);
    }
  }
}
```

Aclaraciones: k -> nivel del árbol

En el pseudocódigo se pasan como parámetros a la función “ back_recursivo “ una referencia a un objeto de tipo Solución llamado sol donde almacena la solución al problema y el nivel en el que comenzamos en el árbol. En nuestro caso, como hemos mencionado antes, le pasamos dos vectores de tipo int (uno para la posible solución y otro para la solución final) y el nivel 1 inicialmente.

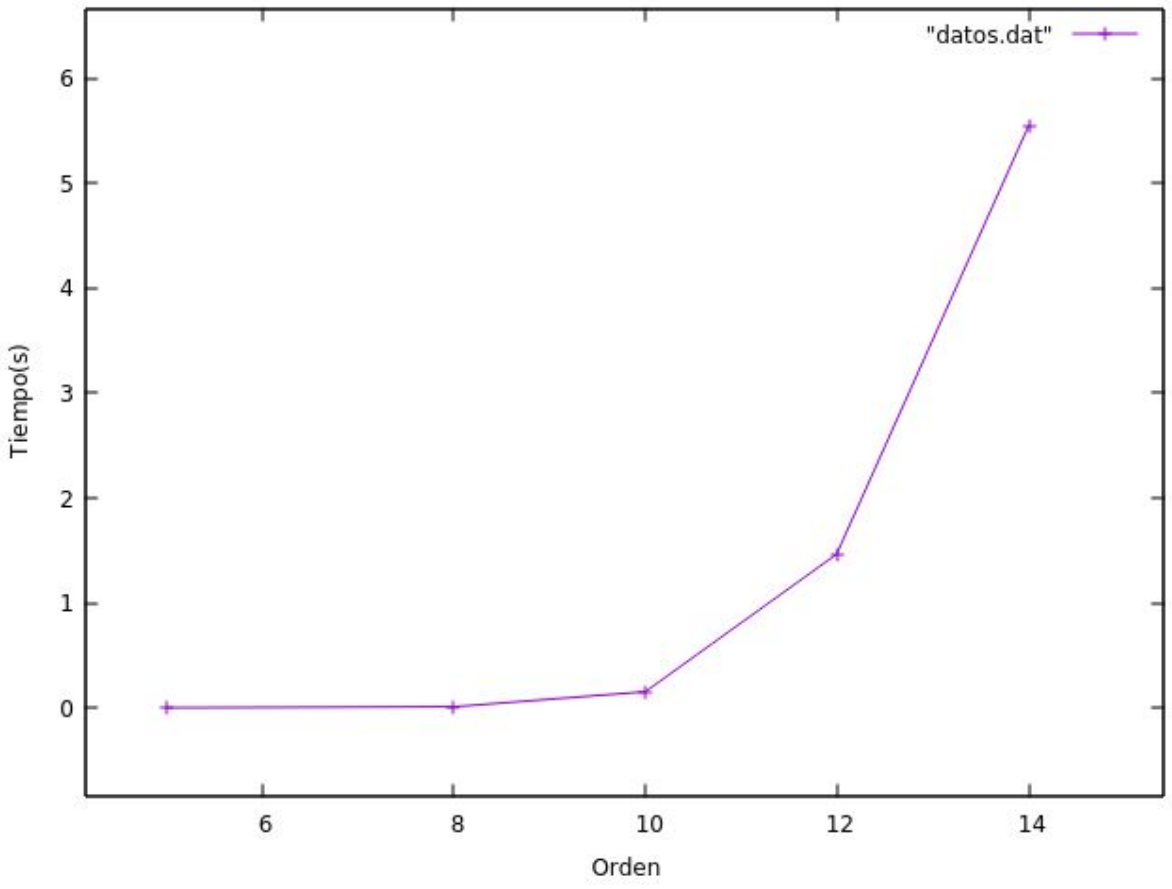
En primer lugar, se comprueba si se ha llegado a un nodo hoja, esto es, hemos llegado a una solución. En caso contrario, “ IniciaComp “ obtiene el valor del primer nodo y lo almacena y “ SigValComp “ obtiene el siguiente componente a este. A continuación, se recorren todos los posibles valores que toman los nodos del nivel

k. Se comprueba que el valor del nodo actual sea factible, es decir, que su suma a la cota local sea menor a la global y que si lo es descienda en profundidad al nivel k+1.

En nuestra propuesta a la resolución del problema, las funciones de IniciaComp y SigValComp las hacemos al principio de la función almacenando el primer punto de venta al que distribuye el primer punto de distribución y posteriormente analizando los nodos de un mismo nivel en el for. Dentro del for vemos si un nodo es factible, como se ha indicado anteriormente, y si lo es analizar los nodos del siguiente nivel llamando de nuevo a la función backtracking. Por último, la comprobación inicial del pseudocódigo en la que se comprueba que estamos en un nodo hoja la hacemos al final del for.

## 4. Eficiencia empírica

Vamos a hacer un estudio de la eficiencia empírica con matrices de distinto orden para ver como se queda reflejado la eficiencia en la gráfica como se realizó en anteriores prácticas. Cabe destacar que al utilizar matrices con valores fijos, no se puede hacer un estudio de muchas ejecuciones, por lo que la eficiencia empírica tendrá unos valores aproximados. A continuación, se muestra la eficiencia empírica:




Podemos apreciar, siendo el eje y el tiempo de ejecución en segundos y el eje x el tamaño/orden de la matriz, que la tendencia que tiene la gráfica toma una forma muy parecida a n!, a pesar de que sean valores aproximados.

## 5. Casos de ejecución

Un ejemplo de ejecución para este problema es con la matriz de distancias M:

```
 20 500 100  40  50
500  30  10  15  60
100  10  35 200  11
 40  15 200  80  70
 50  60  11  70  55
```

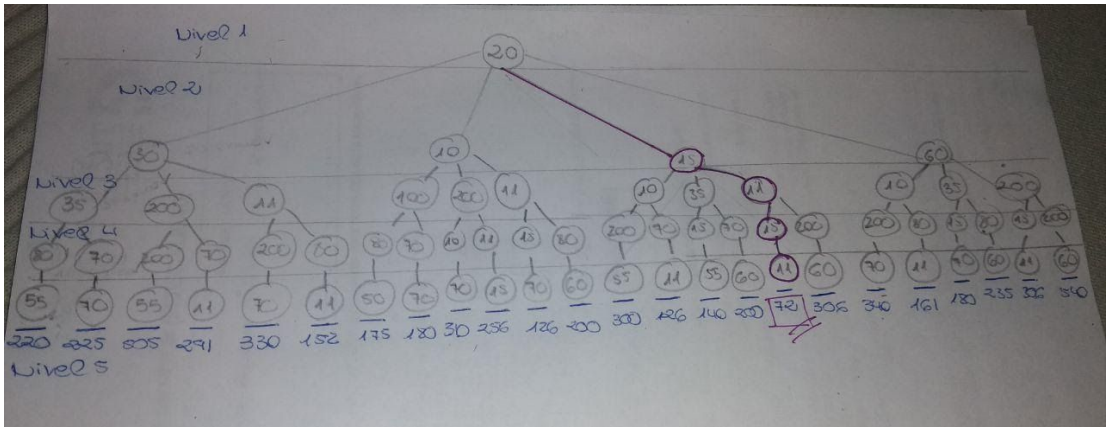
Sabiendo que los puntos de distribución son las filas y los puntos de venta son las columnas, podemos mostrar el recorrido de la solución así:

```
Punto de distribución: 0 ---> Punto de venta: 0
Punto de distribución: 1 ---> Punto de venta: 3
Punto de distribución: 2 ---> Punto de venta: 4
Punto de distribución: 3 ---> Punto de venta: 1
Punto de distribución: 4 ---> Punto de venta: 2
```

Por lo que si asumimos los puntos de distribución como “i” y los puntos de venta como “j” y sumamos todas las distancias, obtenemos que la distancia mínima es 72. Si hacemos aplicamos el algoritmo de backtracking a cada nodo padre por separado, obtenemos este resultado:

```
Punto de distribución inicial 0 distancia minima: 72
Punto de distribución inicial 1 distancia minima: 577
Punto de distribución inicial 2 distancia minima: 191
Punto de distribución inicial 3 distancia minima: 126
Punto de distribución inicial 4 distancia minima: 126
```

Vemos que efectivamente la distancia mínima es 72 si empezamos desde el punto de distribución 0. Un ejemplo del árbol solución generado es el siguiente:



- ## Conclusión final prácticas:

Con motivo de que esta es la última práctica de esta asignatura, queremos realizar una pequeña conclusión en donde expresamos las cosas que hemos sacado en claro tras haber cursado y realizado las diversas prácticas.

En primer lugar, destacamos la primera práctica que hemos realizado, la cual, se trataba principalmente de conocer las diversas formas que tenemos para realizar el análisis de la eficiencia sobre un algoritmo. Algo que aprendimos y que queremos resaltar es la diferencia existentes a la hora de estudiar cada algoritmo, puesto que cada caso es un mundo, en algunos debes ampliar el rango de valores, en otros reducirlos, en otro debes realizar más ejecuciones para encontrar diferencias significativas, etc. Todos estos detalles son los que un programador debe tener en cuenta a la hora de analizar el algoritmo, para así poder conseguir los datos más exactos posibles sobre el rendimiento del algoritmo en el problema a analizar.

En segundo lugar, con la segunda práctica comenzamos a ver los primeros enfoques que estudiaríamos en la asignatura, fuerza bruta - divide y vencerás. En dicha práctica nos dimos cuenta que para cada problema debemos saber seleccionar cuál enfoque debemos utilizar, puesto que, aunque existan diversos enfoques con resultados muy buenos o eficientes, cada problema es un mundo y debe analizarse y solucionarse con el enfoque que mejor se adapte a él. Y por ello, nosotros debemos conocer diversos métodos por los cuales podemos atacar un algoritmo, y saber seleccionar cual le viene mejor.

Y finalmente, estas tres últimas prácticas hemos abordado el problema de TSP (o del viajante de comercio) con diversos enfoques: 3.Enfoque greedy, 4.Enfoque PD, 5.Enfoque B&B y Backtracking. Al realizar todos los enfoques hemos sacado diversas conclusiones; en primer lugar el enfoque greedy, es un enfoque muy veloz en términos de tiempo de ejecución, además no tenía ningún problema con los diversos tamaños de los datos que nos aportó nuestro profesor. El único inconveniente que poseía dicho enfoque es que sus resultados no eran exactos, es decir, nos otorgaba una solución aproximada no la óptima. Por otro lado el enfoque de programación dinámica, sí nos otorgaba la solución óptima del problema, aunque a costa de una gran carga de memoria (con la memorización que emplea el enfoque) y un tiempo de ejecución peor que greedy. Y finalmente, B&B y Backtracking, otorgaban también la solución óptima del problema, pero compartían el inconveniente de la velocidad de ejecución puesto que según aumentaba el número de ciudades, el tiempo crecía exponencialmente. Por lo que, hemos llegado a la conclusión de que aún siendo el mismo problema, debemos tener en cuenta que otros factores como los datos (número de ciudades), hace que uno u otro algoritmo se comporte de manera más eficiente o no. Además de que debemos pensar en que, todos los algoritmos que nos dan la solución óptima empeoran significativamente en otro aspecto, por lo que debemos ser conscientes y saber que nos interesa más para elegir qué enfoque utilizar.