UNIVERSIDAD
DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Doble Grado en Ingeniería Informática y Matemáticas

Curso 2022 - 2023

Algorítmica

Memoria de prácticas

Práctica 4: Exploración de grafos

Autores:

- Mario Martín Rodríguez mario10@correo.ugr.es
- Juan Ayuso Arroyave juanayuso@correo.ugr.es
- Mario Líndez Martínez mariolindez@correo.ugr.es

Índice

1. Autores:

- Mario Líndez Martínez: 33’33%
- Mario Martín Rodríguez: 33’33%
- Juan Ayuso Arroyave: 33’33%

Mario Líndez ha implementado el algoritmo Backtracking y Mario Martín ha implementado el algoritmo Branch and Bound con la ayuda de Mario Líndez.

Mario Líndez ha hecho el Makefile y el generador de puntos.

Juan Ayuso ha hecho la memoria, el mide tiempos y la obtención de todos los datos y gráficas.

Mario Martín ha hecho la presentación.

2. Objetivos

El objetivo de esta práctica es el uso y compresión de técnicas para resolver problemas basados en la exploración de grafos, concretamente las técnicas “Backtracking” y “Branch and Bound”. Para ello, hemos resuelto el problema del viajante de comercio utilizando cada una de estas dos técnicas que hemos nombrado.

3. Definición del problema:

3.1. Enunciado

Un agente comercial tiene que visitar n ciudades. Se proporciona una matriz cuadrada D, de dimensiones n × n tal que di,j indica el coste de viajar desde la ciudad i a la j. El agente debe realizar un recorrido visitando todas las ciudades y finalizando en la misma ciudad desde la que partió. Además, sólo puede visitar cada ciudad una sola vez. El coste global de un recorrido dado es la suma de los costes de todos los traslados. El objetivo es encontrar un recorrido que cumpla las restricciones establecidas y tenga el menor coste posible.

3.2. Casos usados en la evaluación de la eficiencia

Para estudiar la eficiencia de los algoritmos, hemos usado desde 1 nodo hasta 15 nodos.

3.3. Entorno de análisis

El análisis se realizará desde el ordenador de Juan Ayuso Arroyave, cuyas especificaciones son las siguientes:

- Nombre: HP Laptop 15s-eq2xxx
- Procesador: AMD® Ryzen 7 5700u with radeon graphics × 16
- Disco Duro Sólido (SSD) de 1 TB
- Memoria RAM: 16 GB
- Sistema Operativo: Ubuntu 22.04.1 LTS de 64 bits
- Compilador: g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

3.4. Medición de tiempos

Para medir los tiempos hemos usado la biblioteca de la STL chrono, la cual permite calcular los tiempos de ejecución de forma más precisa. La implementación es la siguiente:

```
1. #include <chrono>
2. using std::chrono;
```

```
 3.
 4. high_resolution_clock::time_point t_antes, t_despues;
 5. duration<double> transcurrido;
 6. t_antes = high_resolution_clock::now();
 7. //sentencia o programa a medir
 8. t_despues = high_resolution_clock::now();
 9. transcurrido = duration_cast<duration<double>> (t_despues - t_antes);
10. cout << "el tiempo empleado es " << transcurrido.count() << " s." << endl;
```

4. Algoritmo Diseñado (Técnica Backtracking)

4.1. Realización:

Nosotros hemos trabajado en la implementación del algoritmo con la clase BT (Backtracking), a la cual le hemos añadido diversos métodos que nos ayudarán a resolver este problema. Dicha clase posee los siguientes datos miembro:

```
 1. int NUM_NODOS;
 2. set<int> sin_visitar;
 3.
 4. vector<vector<int>> distancias;
 5. vector<int> solucion;
 6.
 7. int COTA_GLOBAL;
 8.
 9. vector<int> salidas_minimas;
10. int minimo_coste;
11.
```

La utilización de un set como contenedor para guardar los nodos que queden sin visitar se debe a su fácil y eficiente método de eliminación de nodos (set::erase(int k)).

En primer lugar, cargamos con el método load la matriz de distancias, el vector que contendrá el índice de salidas mínimas de cada fila (usado para el cálculo de la Cota2, almacenado como dato miembro por eficiencia), la COTA_GLOBAL (menor distancia a través de un método Greedy) e inicializamos la solución a la obtenida por el método Greedy (heurística usada la del vecino más cercano).

Código del método load:

```
 1.         ifstream file;
 2.         file.open(filename);
 3.
 4.         if (!file){
 5.             cout << "ERROR ABRIENDO EL ARCHIVO" << endl;
 6.             exit(-1);
 7.         }
 8.
 9.         int nodos_aux;
10.         file >> nodos_aux;
11.
12.         NUM_NODOS = nodos_aux;
13.
14.         reservaMemoria(NUM_NODOS);
15.
16.         for (int i = 0; i < NUM_NODOS; ++i) {
17.             for (int j = 0; j < NUM_NODOS; ++j) {
18.                 int num;
19.                 file >> num;
20.
21.                 distancias[i][j] = num;
22.             }
23.         }
24.
25.         file.close();
30.
```

```
31.         vector<bool> ya_pertenece (NUM_NODOS);
32.         ya_pertenece.at(0) = true;
33.
34.         CalculaSalidasMinimas();
35.
36.         minimo_coste = distancias[0][salidas_minimas.at(0)];
37.         for (int i = 1; i < NUM_NODOS; ++i){
38.             if (minimo_coste > distancias[i][salidas_minimas.at(i)]){
39.                 minimo_coste = distancias[i][salidas_minimas.at(i)];
40.             }
41.         }
42.
43.         pair<vector<int>, int> sol_greedy = Greedy (distancias, ya_pertenece, NUM_NODOS);
44.         COTA_GLOBAL = sol_greedy.second;
45.         solucion = sol_greedy.first;
46.
47.     }
```

Tras esto, pasamos al método más importante, el cual se trata de pvc. Este método nos servirá como acceso a la resolución del problema usando las diferentes cotas.

```
1. void pvc(int cota){
2.         vector<int> solucion_aux;
3.         if (cota == 1){
4.             pvc1(0, solucion_aux);
5.         } else if (cota == 2){
6.             pvc2(0, solucion_aux);
7.         }
8.     }
```

El funcionamiento tanto de pvc1 como de pvc2 es equivalente, sin embargo, cada uno usará la primera cota local o la segunda respectivamente. Esto se podría haber generalizado usando punteros a funciones, pero por falta de tiempo hemos realizado una implementación más rápida, aunque para nada recomendable.

Usaremos pvc1 para explicar el funcionamiento del programa, equivalente en ambos casos. Hemos optado por un diseño recursivo, por lo que nuestra función recibirá como argumentos el nodo actual en el que se encuentra (índice fila), un vector de solución auxiliar y el coste del camino recorrido actual. En primer lugar, quitaremos el nodo actual del set que almacena los nodos que todavía quedan sin visitar y lo añadiremos al vector solución auxiliar. Tras esto comprobaremos si ya se han visitado todos los nodos, es decir, el nodo en el que estamos es una hoja de nuestro árbol de estados.

En caso de que así sea sumaremos al coste total la distancia de volver al nodo inicial (0) desde el último nodo considerado y comprobaremos si es un menor coste que lo ya almacenado. Si es menor, la consideraremos como la nueva mejor solución, tomando como nueva COTA_GLOBAL su coste.

En caso contrario, comprobaremos la cota local del nodo actual, la cual nos dará una estimación de cuánto nos costará en términos mínimos seguir por ese camino. Si estimamos que el camino merecerá la pena (que la cota local sea menor que la global), pasaremos a procesar los caminos restantes. Antes de llamar recursivamente a la función sumaremos el coste de ir a ese nuevo nodo, se procesará recursivamente y al volver deberemos retroceder al estado anterior. Esto es restar su coste considerado antes de entrar a la función, añadirlo de nuevo al set de nodos sin_visitar y eliminarlo de la solucion_auxiliar.

Código pvc1, análogo a pvc2 exceptuando la llamada a CotaLocal1:

```
 1. void pvc1 (int k, vector<int> & solucion_aux, int coste=0){
 2.         sin_visitar.erase(k);
 3.         solucion_aux.push_back(k);
 4.
 5.         if (sin_visitar.empty()){
 6.             coste += distancias[k][0];
 7.             if (coste < COTA_GLOBAL){
 8.                 COTA_GLOBAL = coste;
 9.                 solucion = solucion_aux;
10.                 PintaSolucion(coste);
11.             }
12.         } else {
13.             int cota_local = CotaLocal1(coste);
14.             if (cota_local < COTA_GLOBAL){
15.                 set<int> sin_visitar_aux = sin_visitar;
16.
17.                 for (auto it = sin_visitar_aux.begin(); it != sin_visitar_aux.end(); ++it){
18.                     coste += distancias[k][*it];
19.                     pvc2(*it, solucion_aux, coste);
20.
21.                     coste -= distancias[k][*it];
22.                     sin_visitar.insert(*it);
23.                     solucion_aux.pop_back();
24.                 }
25.             }
26.         }
27.     }
```

4.2. Justificación de la validez del algoritmo:

Nuestro algoritmo es prácticamente un algoritmo de fuerza bruta optimizado ya que irá recorriendo la totalidad del árbol de estados dejándose por el camino aquellas ramas que ya estimamos que no nos ofrecerán una solución válida. Es por ello por lo que este algoritmo será valido y ofrecerá una solución correcta en cada ejecución.

4.3. Funciones de cota alternativas:

Para la resolución de nuestro problema, hemos usado dos funciones de cota. La primera de ellas tomaremos la distancia más baja de toda la matriz de distancias, es decir, la salida con el coste mínimo. Dicha distancia será usada para estimar el mínimo coste de ir a los nodos que aún queden sin visitar y se sumará al coste del camino actual.

Código CotaLocal1:

```
1. int CotaLocal1 (int coste){
2.         int num_nodos_sin_visitar = sin_visitar.size();
3.
4.         coste += minimo_coste*num_nodos_sin_visitar;
5.
6.         return (coste);
7.     }
```

La segunda cota local usará el mismo concepto, pero en vez de tomar una única salida mínima tomaremos el coste mínimo especifico de salir de cada nodo. Esto se almacenó en el vector salidas_minimas.

Código CotaLocal2:

```
int CotaLocal2 (int coste){
        coste += GetSalidaMinima(solucion.back());
        for (auto it = sin_visitar.begin(); it != sin_visitar.end(); ++it){
            coste += GetSalidaMinima(*it);
        }

        return (coste);
    }
int GetSalidaMinima (int k){
         return (distancias[k][salidas_minimas.at(k)]);
}
```

5. Eficiencias (Técnica Backtracking)

5.1. Eficiencia teórica:

La eficiencia del algoritmo es indeterminada, pues número de llamadas recursivas dependerán del caso en el que nos encontremos en ese momento. Por lo tanto, no tiene sentido hablar de eficiencia teórica.

Con respecto a las cotas, claramente la cota1 es O(1) ya que son dos operaciones sencillas e inmediatas, mientras que la cota2 dependerá del número de nodos que queden sin visitar, esto es O(k).

5.2. Eficiencia empírica:

Los datos que vamos a calcular a continuación son utilizando la segunda cota que hemos definido en el apartado anterior por ser la más precisa y la supuestamente más eficiente.

Para realizar un análisis de eficiencia empírica deberemos ejecutar el mismo algoritmo para diferentes tamaños de entrada. Para este algoritmo, lo ejecutaremos para diferentes tamaños del vector del número de nodos que hay. Estos tiempos los almacenamos en un fichero dist_bt.dat. El código es el siguiente:

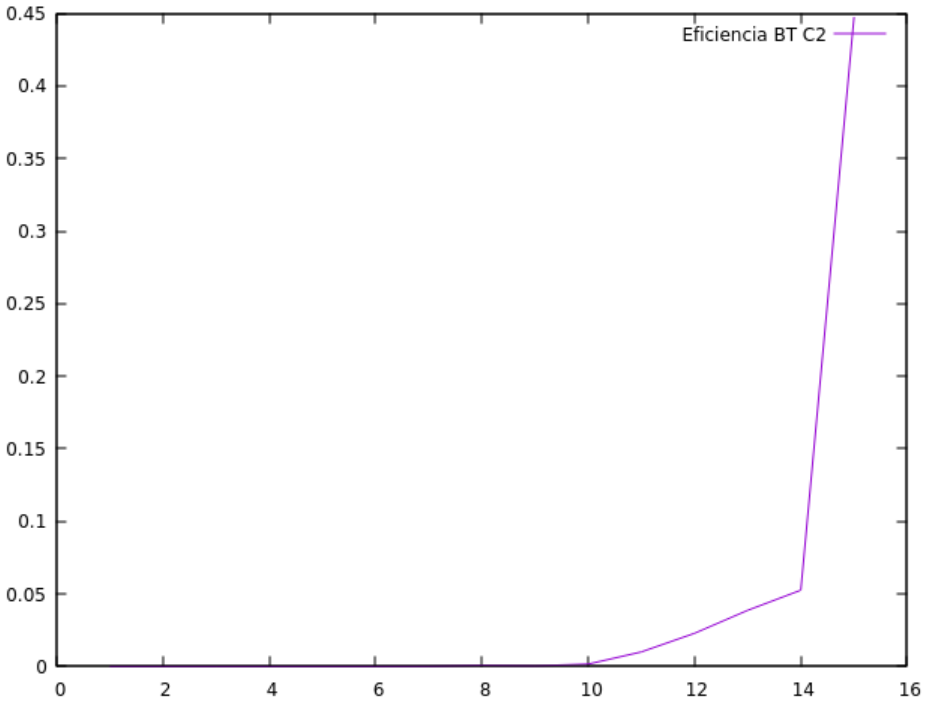
```
#!/bin/bash
#echo "" >> salida.dat
printf "" > dist_bt.dat

i=1
while [ "$i" -le 15 ]
do
      # Generamos los puntos
      ./generador $i data_bt.txt
      printf "PUNTOS GENERADOS\t"

      # Ejecutamos los puntos
      ./bt data_bt.txt >> dist_bt.dat

      echo "Terminado $i"

      i=$(( $i + 1 ))
done
```

Empezamos con un tamaño base de 1 punto y vamos aumentándolo de 1 en 1 hasta llegar al tamaño de 15 puntos.

Los tiempos obtenidos son los siguientes:

Tamaño	Tiempo (seg)
1	0,00000209
2	0,00000274
3	0,00000135
4	0,00001924
5	0,00001313
6	0,00001370
7	0,00015042
8	0,00080108
9	0,00037696
10	0,00165159
11	0,01007440
12	0,02290940
13	0,03869480
14	0,05255540
15	0,44691400

A partir de dichos tiempos, queda la siguiente gráfica:



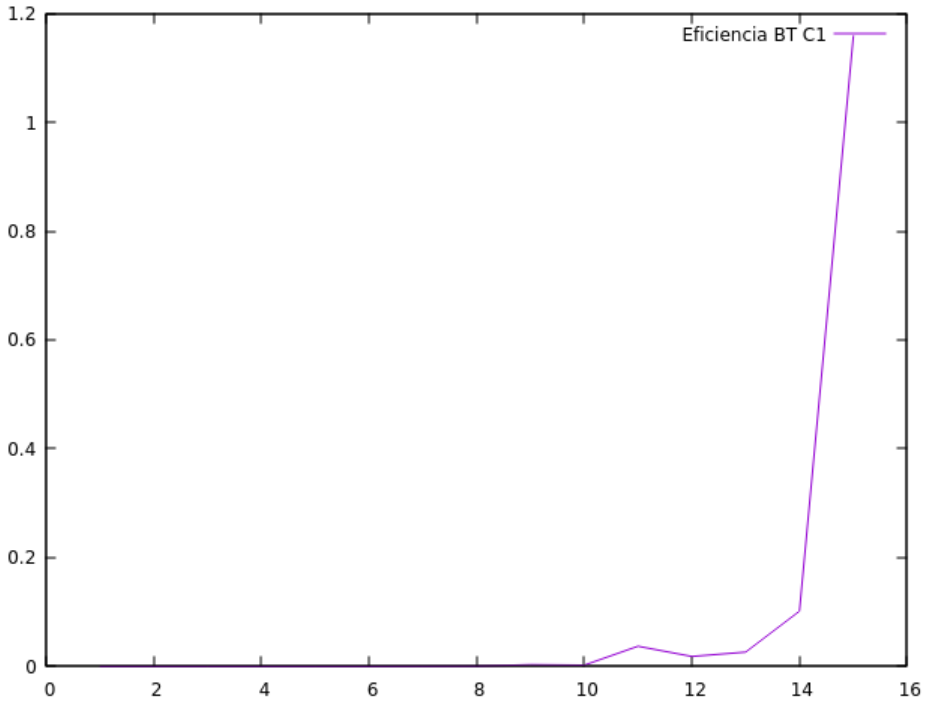
Gráfica de tiempos del algoritmo diseñado con la cota2

6. Análisis comparativo del rendimiento y tiempos de las distintas funciones de cota consideradas (Técnica Backtracking)

Para la cota1 que hemos implementado, los datos y gráfica son las siguientes:

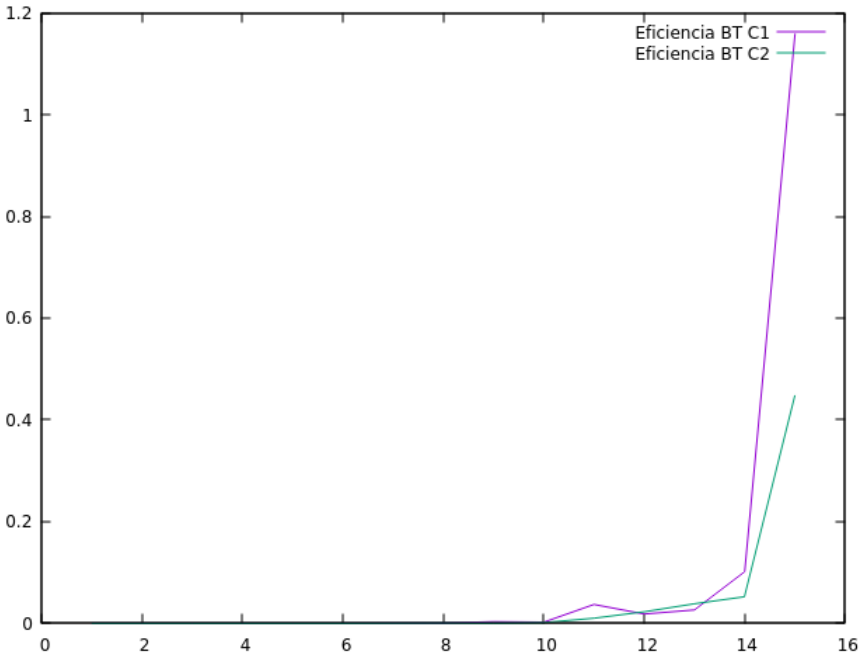
Tamaño	Tiempo (seg)
1	0,00000095
2	0,00000267
3	0,00000449
4	0,00000400
5	0,00001897
6	0,00001128
7	0,00014896
8	0,00067671
9	0,00322674
10	0,00251811
11	0,03738360
12	0,01861960
13	0,02640090
14	0,10151000
15	1,15773000

Cuya gráfica será la siguiente:



Gráfica de tiempos del algoritmo diseñado con cota1

Si analizamos las gráficas de ambas cotas, nos queda lo siguiente:



Gráfica de tiempos comparada

Donde claramente podemos observar como la ganancia en precisión de la cota2 es más determinante que su propia eficiencia, pues hace el programa más eficiente.

7. Algoritmo Diseñado (Técnica Branch and Bound)

7.1. Realización:

Para la técnica Branch and Bound nos volveremos a apoyar en una clase BB cuya implementación será muy similar a la clase BT. Sus datos miembros son:

```
 1.     int NUM_NODOS;
 2.
 3.     vector<vector<int>> distancias;
 4.     int COTA_GLOBAL;
 5.
 6.     vector<int> salidas_minimas;
 7.     int minimo_coste;
 8.
 9.     nodo origen;
10.
11.     vector<int> solucion;
12.
13.     priority_queue<pair<nodo,int>, vector<pair<nodo,int>>, Comparison> caminos;
```

En la línea 9 observamos un dato de tipo nodo. Este será un struct que representará la situación actual del camino recorrido hasta llegar a dicho nodo, esto es su vector solución, los nodos que queden sin visitar y por motivos de optimización, su coste actual. Esto es:

```
1. struct nodo {
2.     vector<int> solucion;
3.     set<int> sin_visitar;
4.     int coste;
5.
6.     nodo(){
7.         coste = 0;
8.     }
9.
```

```
10.     nodo(const vector<int> & la_solucion, const set<int> & los_sin_visitar, int el_coste){
11.         solucion = la_solucion;
12.         sin_visitar = los_sin_visitar;
13.         coste = el_coste;
14.     }
15.
16.     int NodosVisitados () const{
17.         return (solucion.size());
18.     }
19. };
```

Observamos que este struct posee varios métodos básicos para inicializarlo y obtener información clave para el algoritmo.

Además de esto nos apoyaremos del functor Comparison para ordenar la cola con prioridad perteneciente a la clas BB, el cual comparará por el segundo elemento del pair de elementos. Éste será la cota local del nodo añadido como primer elemento.

```
1. struct Comparison {
2.     bool operator() (const pair<nodo,int>& n1, const pair<nodo,int> & n2){
3.         return (n1.second > n2.second);
4.     }
5. };
```

Continuemos viendo el resto del funcionamiento de la clase.

En primer lugar, la inicialización vendrá dada de nuevo por un método load el cual será análogo en comportamiento con el de la clase BT. Se encargará de inicializar la matriz de distancias, el nodo origen, la solución, la cota global inicial (calculadas de nuevo con el algoritmo Greedy del vecino más cercano), el vector de salidas mínimas y el mínimo coste (objetos usados para la cota2 y 1 respectivamente).

Código del método load:

```
 1.     void load (const char filename[]){
 2.         ifstream file;
 3.         file.open(filename);
 4.
 5.         if (!file){
 6.             cout << "ERROR ABRIENDO EL ARCHIVO" << endl;
 7.             exit(-1);
 8.         }
 9.
10.         int nodos_aux;
11.         file >> nodos_aux;
12.
13.         NUM_NODOS = nodos_aux;
14.
15.         reservaMemoria(NUM_NODOS);
16.
17.         set<int> set_origen;
18.
19.         for (int i = 0; i < NUM_NODOS; ++i) {
20.             for (int j = 0; j < NUM_NODOS; ++j) {
21.                 int num;
22.                 file >> num;
23.
24.                 distancias[i][j] = num;
25.             }
26.
27.             if (i != 0){
28.                 set_origen.insert(i);
29.             }
30.
31.         }
32.
33.         file.close();
```

```
34.
35.         origen = nodo(vector<int>{0}, set_origen, 0);
36.
37.         // Siempre partimos de la ciudad 0
38.         //solucion.push_back(0);
39.         //sin_visitar.erase(0);
40.
41.         vector<bool> ya_pertenece (NUM_NODOS);
42.         ya_pertenece.at(0) = true;
43.
44.         CalculaSalidasMinimas();
45.
46.         minimo_coste = distancias[0][salidas_minimas.at(0)];
47.         for (int i = 1; i < NUM_NODOS; ++i){
48.             if (minimo_coste > distancias[i][salidas_minimas.at(i)]){
49.                 minimo_coste = distancias[i][salidas_minimas.at(i)];
50.             }
51.         }
52.
53.
54.         pair<vector<int>, int> sol_greedy = Greedy (distancias, ya_pertenece, NUM_NODOS);
55.         COTA_GLOBAL = sol_greedy.second;
56.         solucion = sol_greedy.first;
57.
58.     }
59.
```

Tras esto, veamos el funcionamiento del método pvc(), de nuevo puerta de acceso para la resolución del problema usando las distintas cotas diseñadas:

```
1. void pvc (int cota){
2.     if (cota == 1){
3.         pvc1();
4.     } else if (cota == 2){
5.         pvc2();
6.     }
7. }
```

De nuevo, esta no es una implementación recomendable, pero sí es funcional. Analicemos ahora pvc1, el cual será análogo en comportamiento a pvc2.

Para empezar, añadiremos nuestro primer nodo y punto de partida a la priority_queue caminos, la cual irá conteniendo los posibles nodos a desarrollar. Tras esto comenzaremos el algoritmo en sí el cual consistirá en ir desarrollando el mejor camino posible (información proporcionada por la priority_queue) hasta que no queden nodos prometedores (mientras caminos no este vacío).

Dentro del bucle tomaremos el tope de la cola, eliminándolo de la misma en el proceso, como el nodo más prometedor y procederemos a desarrollarlo. Si este nodo fuera una hoja, esto es que no le queden nodos por visitar) actualizaremos el coste del camino con la vuelta al nodo 0 y comprobaremos si es una mejor solución que la ya poseíamos. Si es así, la tomaremos como nueva mejor solución, tomando como cota global su coste.

En caso de que todavía le queden nodos por visitar, vemos si la cota local de dicho nodo es menor que la global, en caso de que se haya modificado esta última. Si lo es, procederemos a añadir como a la cola aquellos caminos que nos deriven del nodo tomado. Para ello, nos apoyaremos en un nodo auxiliar inicialmente igual que prometedor y le actualizaremos el nuevo camino tomado hacia cada uno de los nodos que le queden por visitar. Así mismo, añadimos también al coste la distancia desde el último nodo de la solución hasta el que acabamos de añadir. Si la cota de este nuevo posible camino es menor que la cota global, lo tomaremos en cuenta y lo añadiremos a caminos.

Código pvc1:

```
 1.    void pvc1(){
 2.         caminos.push(pair<nodo,int>{this->origen, CotaLocal2(this->origen)});
 3.
 4.         while (!caminos.empty()){
 5.             nodo prometedor = caminos.top().first;
 6.             caminos.pop();
 7.
 8.             if (prometedor.sin_visitar.empty()){
 9.                 prometedor.coste += distancias[prometedor.solucion.back()][0];
10.
11.                 if (prometedor.coste < COTA_GLOBAL){
12.                     solucion = prometedor.solucion;
13.                     COTA_GLOBAL = prometedor.coste;
14.                 }
15.
16.             } else {
17.                 if (prometedor.coste < COTA_GLOBAL){
18.                     for (auto it = prometedor.sin_visitar.begin(); it !=
prometedor.sin_visitar.end(); ++it){
19.                         nodo aux = prometedor;
20.
21.                         aux.sin_visitar.erase(*it);
22.                         aux.solucion.push_back(*it);
23.                         aux.coste += distancias[prometedor.solucion.back()][*it];
24.
25.                         int cota = CotaLocal1(aux);
26.
27.                         if (cota < COTA_GLOBAL){
28.                             caminos.push(pair<nodo,int>{aux,cota});
29.                         }
30.                     }
31.                 }
32.
33.             }
34.
35.         }
36.     }
37.
```

7.2. Justificación de la validez del algoritmo:

Por motivos análogos al algoritmo Backtracking, este algoritmo será válido, pues irá recorriendo el árbol a través de aquellos nodos que sean útiles, encontrando así todas las mejores soluciones hasta el momento. A diferencia del algoritmo antes mencionado, Branch and Bound desarrollará antes los nodos antes de tomarlos como procesables.

7.3. Funciones de cota alternativas:

Para la resolución de nuestro problema, hemos usado dos funciones de cota. Para la primera de ellas tomaremos la distancia más baja de toda la matriz de distancias, es decir, la salida con el coste mínimo. Dicha distancia será usada para estimar el mínimo coste de ir a los nodos que aún queden sin visitar y se sumará al coste del camino actual.

Código CotaLocal1:

```
1.    int CotaLocal1(const nodo & n) const{
2.         int coste = n.coste;
3.         int num_nodos_sin_visitar = n.sin_visitar.size();
4.
5.         coste += (minimo_coste*num_nodos_sin_visitar);
6.
7.         return (coste);
8.     }
9.
```

La segunda cota local usará el mismo concepto, pero en vez de tomar una única salida mínima tomaremos el coste mínimo especifico de salir de cada nodo. Esto se almacenó en el vector salidas_minimas.

Código CotaLocal2:

```
 1.    int CotaLocal2 (const nodo & n){
 2.        int coste = n.coste;
 3.
 4.        coste += GetSalidaMinima(solucion.back());
 5.
 6.        for (auto it = n.sin_visitar.begin(); it != n.sin_visitar.end(); ++it){
 7.            coste += GetSalidaMinima(*it);
 8.        }
 9.
10.        return (coste);
11.    }
12.
```

Como vemos estas cotas son las mismas que hemos usado anteriormente, pero adaptadas al nuevo tipo nodo.

8. Eficiencias (Técnica Branch and Bound)

8.1. Eficiencia teórica:

De nuevo, de forma análoga a la técnica Backtracking, no tiene sentido hablar de eficiencia teórica pues el número de nodos a desarrollar y procesar dependerá de cada caso.

Como las cotas usadas son realmente las mismas, mantendrán sus respectivas eficiencias, esto es $Cota1 \in O(1)$ y $Cota2 \in O(k)$, siendo k el número de nodos que le queden sin visitar al nodo actual.

8.2. Eficiencia empírica:

Para realizar un análisis de eficiencia empírica deberemos ejecutar el mismo algoritmo para diferentes tamaños de entrada. Para este algoritmo, lo ejecutaremos para diferentes tamaños del vector del número de nodos que hay. Estos tiempos los almacenamos en un fichero dist_bb.dat. El código es el siguiente:

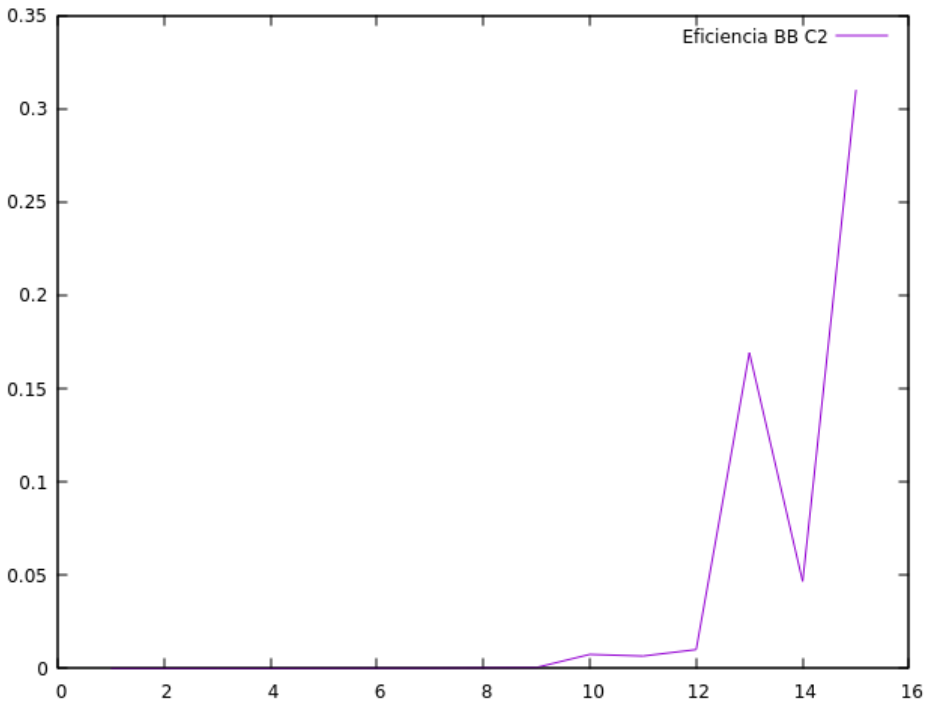
```
 1. #!/bin/bash
 2. #echo "" >> salida.dat
 3. printf "" > dist_bb.dat
 4.
 5. i=1
 6. while [ "$i" -le 15 ]
 7. do
 8.       # Generamos los puntos
 9.       ./generador $i data_bb.txt
10.       printf "PUNTOS GENERADOS\t"
11.
12.       # Ejecutamos los puntos
13.       ./bt data_bb.txt >> dist_bb.dat
14.
15.       echo "Terminado $i"
16.
17.       i=$(( $i + 1 ))
18. done
```

Empezamos con un tamaño base de 1 punto y vamos aumentándolo de 1 en 1 hasta llegar al tamaño de 15 puntos.

Los tiempos obtenidos son los siguientes:

Tamaño	Tiempo (seg)
1	0,00000096
2	0,00000137
3	0,00000875
4	0,00000143
5	0,00005219
6	0,00016528
7	0,00013243
8	0,00041251
9	0,00054091
10	0,00758994
11	0,00664845
12	0,01012400
13	0,16925900
14	0,04675850
15	0,30960900

A partir de dichos tiempos, queda la siguiente gráfica:



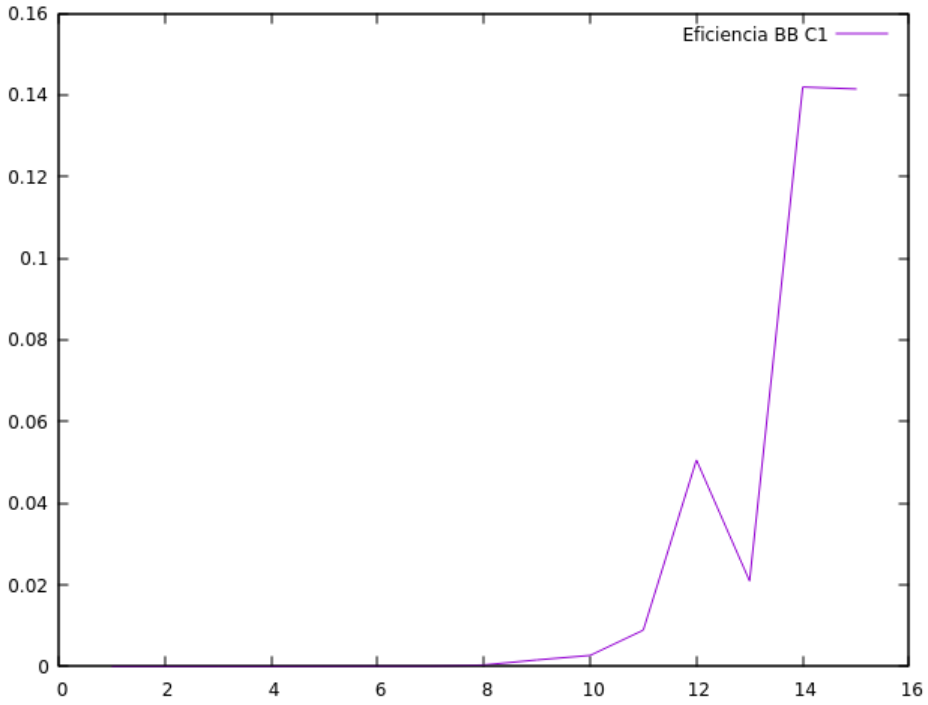
Gráfica de tiempos del algoritmo diseñado con la cota2

9. Análisis comparativo del rendimiento y tiempos de las distintas funciones de cota consideradas (Técnica Branch and Bound)

Para la otra función de cota que hemos implementado, los datos y gráfica son las siguientes:

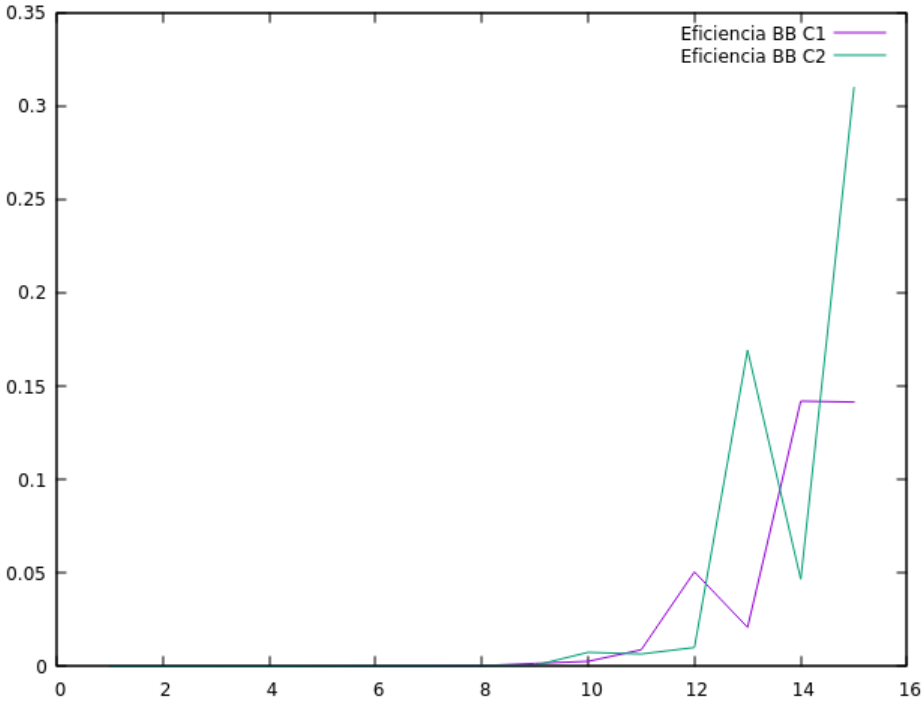
Tamaño	Tiempo (seg)
1	0,00000092
2	0,00000266
3	0,00000415
4	0,00000833
5	0,00002599
6	0,00007267
7	0,00008538
8	0,00037099
9	0,00163019
10	0,00272298
11	0,00890477
12	0,05060160
13	0,02095930
14	0,14192600
15	0,14147100

Cuya gráfica será:



Gráfica de tiempos del algoritmo diseñado

Si juntamos las dos gráficas de las distintas cotas, nos queda lo siguiente:

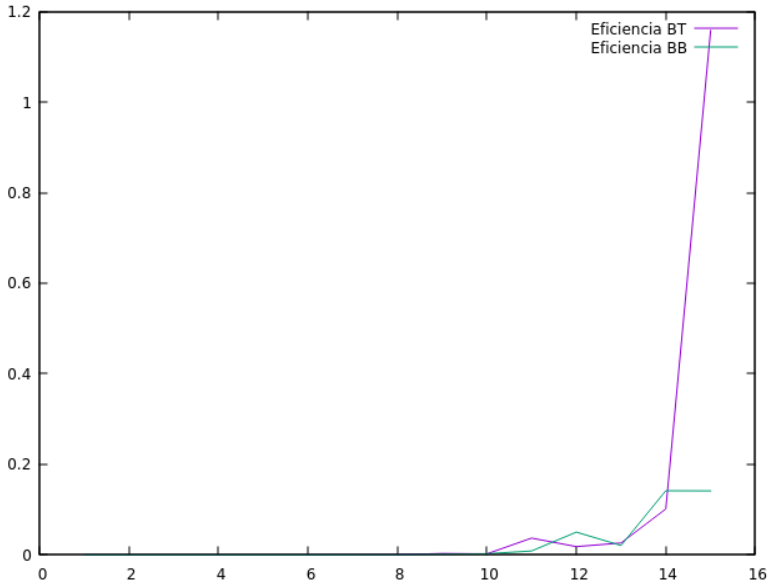


Gráfica de tiempos comparada

Observamos que en este caso la cota1 hace el programa más eficiente. Esto puede verse debido a la propia naturaleza de la técnica Branch and Bound, pues una cota más eficiente pero menos precisa le favorece más al ser una técnica ya precisa de por sí (solo se tienen en cuenta los nodos estrictamente útiles).

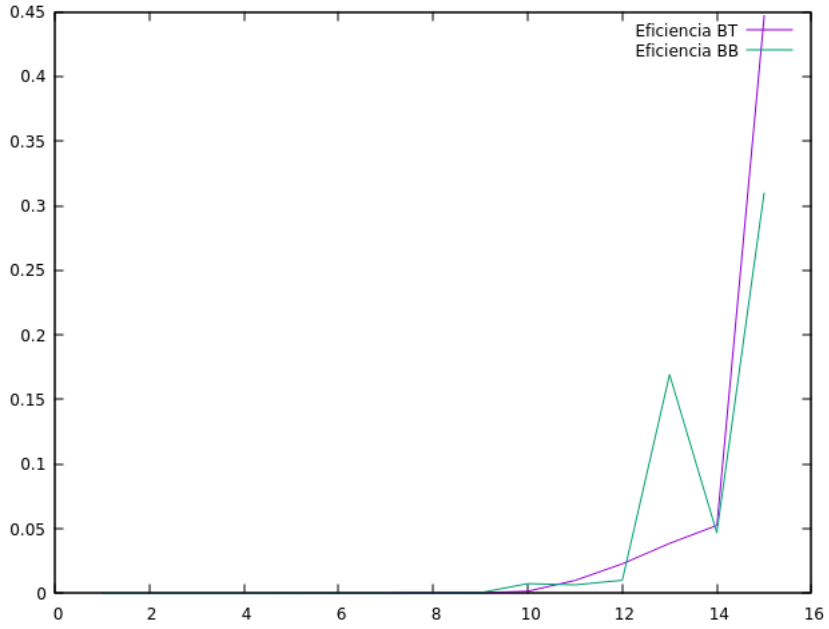
10. Comparación de las dos heurísticas (calidad de soluciones y tiempo de ejecución)

Utilizando en los dos algoritmos la primera cota, la gráfica de tiempos resultante es la siguiente:



Gráfica de tiempos comparada

Utilizando en los dos algoritmos la segunda cota, la gráfica de tiempos resultante es la siguiente:



Gráfica de tiempos comparada

Observamos en ambos casos como la técnica Branch and Bound es significativamente más eficiente en cualquier caso.

11. Conclusión

Como primera conclusión, hemos visto que el diseño de la cota es un factor muy importante ya que la eficiencia de nuestro algoritmo puede mejorar u empeorar dependiendo de esta.

Además, hemos observado que la técnica “Branch and Bound” es más eficiente que la técnica “Backtracking”. Esta conclusión realmente es obvia, ya que con “Backtracking” se recorre el árbol de estados por completo mientras que con “Branch and Bound” tan solo iremos desarrollando las ramas justas y necesarias para determinar qué solución es la más eficiente.