

Algorítmica

Curso 2023-2024

Grupo Viterbi



PRÁCTICA 4-ALGORITMOS DE EXPLORACIÓN DE GRAFOS

Integrantes:

Miguel Ángel De la Vega Rodríguez

miguevrod@correo.ugr.es

Alberto De la Vera Sánchez

joaquinrojo724@correo.ugr.es

Joaquín Avilés De la Fuente

adelaveras01@correo.ugr.es

Manuel Gomez Rubio

e.manuelgmez@go.ugr.es

Pablo Linari Perez

e.pablolinari@go.ugr.es

Facultad de Ciencias UGR
Escuela Técnica Ingeniería Informática UGR
Granada
2023-2024

Índice general

1 Autores	3
2 Equipo de trabajo	4
3 Objetivos	5
4 Backtracking	6
4.1 Diseño	6
4.1.1 Cota global	6
4.1.2 Cota local	7
4.1.3 Algoritmo Backtracking	8
4.2 Justificación	9
4.3 Eficiencia teórica y empírica	10
4.3.1 Eficiencia empírica	10
4.3.2 Eficiencia teórica	11
5 Branch and Bound	12
5.1 Diseño	12
5.1.1 Cota Global	12
5.1.2 Cota local	13
5.1.3 Algoritmo Branch and Bound	14
5.1.4 Algoritmo Branch and Bound	15
5.2 Eficiencia teórica y empírica	17
5.2.1 Eficiencia teórica	17
5.2.2 Eficiencia empírica	17
5.3 Justificación	19
5.4 Gráficas	20

Autores

- **Miguel Ángel De la Vega Rodríguez: 20%**
 - Makefile | Organización Branch and Bound
 - Programación Branch and Bound
 - Branch and bound redacción
- **Joaquín Avilés De la Fuente: 20%**
 - Programación Branch and Bound
 - Programación creación de casos (puntos y matrices de adyacencia)
 - Programación de funciones de cota para Branch and Bound
- **Alberto De la Vera Sánchez: 20%**
 - Branch and bound redacción
- **Manuel Gomez Rubio 20%**
 - Programacion Backtracking
 - Redacción Backtracking
- **Pablo Linari Pérez: 20%**
 - Programacion Backtracking
 - Redacción Objetivos
 - Redacción Backtracking

Apartado 2

Equipo de trabajo

- **Miguel Ángel De la Vega Rodríguez:** (Ordenador donde se ha realizado el compute)
 - AMD Ryzen 7 2700X 8-Core
 - 16 GB RAM DDR4 3200 MHz
 - NVIDIA GeForce GTX 1660 Ti
 - 1 TB SSD NVMe
 - Debian 12 Bookworm
 - Compilador GCC 12.2.0

Apartado 3

Objetivos

El objetivo de esta práctica es resolver el problema del viajante de comercio el cual viene descrito por el siguiente enunciado : Tenemos un conjunto de n ciudades (puntos en un plano), cada una definida por las coordenadas en el mapa (x_i, y_i) , con $i = 1, \dots, n$. La distancia entre dos ciudades viene dada por la distancia euclídea entre sus coordenadas. El problema de viajante de comercio consiste en encontrar el orden en el que un viajante, partiendo de la ciudad de origen (por ejemplo (x_1, y_1)) pase por todas y cada una de las ciudades una única vez, para volver a la ciudad de partida, formando un ciclo.

Para ello usaremos dos diseños distintos de algoritmos dedicados a la exploración de grafos , backtracking y branch and bound con el objetivo de ver las diferencias entre la eficiencia de estos dos algoritmos. Además se probarán distintas funciones de costo para estudiar que influencia tienen en cada caso las distintas funciones. Por último se realizará un estudio tanto teórico como empírico de la eficiencia de los algoritmos implementados.

Backtracking

4.1 Diseño

En esta sección analizaremos el algoritmo de exploración de grafos llamado Backtracking, que consiste en recorrer todos los caminos en profundidad obteniendo todas las posibles soluciones al problema.

Para seleccionar la solución más óptima del problema debemos ir comparando con la anterior posible solución cada vez que obtenemos una nueva solución al problema, quedándonos con la que más nos convenga, en este caso, la que nos de un camino de menor distancia.

Hay varias formas de hacerlo:

- La primera forma consiste en hacerlo mediante fuerza bruta, es decir, usar la definición al uso de Backtracking recorriendo todos los caminos sin preocuparnos si estos nos permiten alcanzar una solución mejor que la obtenida anteriormente.
- La otra forma es implementar una función de cota, esto nos permite decidir si seguimos explorando el camino seleccionado porque nos puede dar un resultado mejor o si es mejor abandonarlo, ya que no se obtendrá una mejora. Esta forma de realizarlo será, como es lógico, más eficiente.

Para la implementación del algoritmo, optamos por la segunda forma, usando diferentes funciones de cota que nos permitan aproximar si el camino seleccionado es bueno. A continuación veremos como se ha elegido cada cota .

4.1.1 Cota global

```

1  vector<int> nearest_neighbourTSP(const vector<vector<double>>&
2      distancias, int inicial) {
3      int num_puntos = distancias.size();
4      vector<int> camino;
5      vector<bool> visitados(num_puntos, false);
6      camino.reserve(num_puntos);
7      camino.push_back(inicial);
8      visitados[inicial] = true;
9
10     for (int i = 0; i < num_puntos - 1; ++i) {
11         int actual = camino.back();
12         int siguiente = -1;
13         double min_distancia = numeric_limits<double>::max();

```

```

13
14     for (int j = 0; j < num_puntos; ++j) {
15         if (!visitados[j] && distancias[actual][j] < min_distancia)
16             {
17                 min_distancia = distancias[actual][j];
18                 siguiente = j;
19             }
20
21     camino.push_back(siguiente);
22     visitados[siguiente] = true;
23 }
24
25     camino.push_back(inicial);
26     return camino;
27 }
```

Esta función de la práctica anterior la usaremos para obtener una cota global, ya que nos proporciona una solución al problema del viajante, aunque no sea la mejor, nos servirá para comparar Si la decisión que toma nuestro algoritmo es mejor que la que nos proporciona esta función.

4.1.2 Cota local

Primera función de cota:

La primera función de cota que hemos implementado consiste en calcular el mínimo valor de los arcos del grafo que no han sido usados todavía , se multiplican por el numero de nodos restantes y se suman al coste actual.

```

1     double cota1(const vector<vector<double>> &graph, const vector<
2         int> &solucion, double c_actual, double arco_menorpeso) {
3         return arco_menorpeso * (graph.size() - solucion.size() +
4             1) + c_actual;
5     }
```

Segunda función de cota:

La segunda función de cota que hemos implementado consiste en calcular el mínimo valor de salir de todos los nodos del grafo que no han sido visitados y sumarle la distancia del recorrido actual. En el siguiente código se recorre la matriz de adyacencia y se selecciona el menor valor de los arcos de los nodos que no han sido visitados y se suman al coste actual.

```

1     double cota2(const vector<vector<double>> &graph, const vector<
2         int> &solucion, double c_actual) {
3         double cota = 0;
4         vector<double> v;
5         double min = numeric_limits<double>::max();
6         for (int i = 1; i < graph.size(); ++i) {
7             if ((find(solucion.begin(), solucion.end(), i) ==
8                 solucion.end())) {
9                 v = graph[i];
10                sort(v.begin(), v.end());
11                cota += v[1];
12            }
13        }
14        return cota + c_actual;
15    }
```

Tercera función de cota:

La tercera función de cota que hemos implementado consiste en calcular el mínimo valor de salir y entrar de todos los nodos del grafo que no han sido visitados ,hacer la media y sumarle la distancia del recorrido actual. El siguiente código busca en una matriz de adyacencia los valores de los arcos de los nodos que no han sido todavía visitados , se seleccionan todos ellos y se ordenan de menor a mayor para seleccionar los dos menores y hacer la media de estos.

```

1      double cota3(const vector<vector<double>> &graph, const vector<
2      int> &solucion, double c_actual) {
3          double cota = 0;
4          vector<double> v;
5          double min = numeric_limits<double>::max();
6          for (int i = 1; i < graph.size(); ++i) {
7              if ((find(solucion.begin(), solucion.end(), i) ==
8                  solucion.end())) {
9                  v = graph[i];
10                 sort(v.begin(), v.end());
11                 cota += (v[1] + v[2]) / 2;
12             }
13         }
14         return cota + c_actual;
15     }

```

4.1.3 Algoritmo Backtracking

A continuación se muestra el algoritmo de Backtracking implementado:

```

1  /**
2   * @brief Funcion para resolver el tsp con backtracking .
3   * @param solucion vector de ciudades , la posicion de la ciudad indica
4   * el orden
5   * en el que es visitada
6   * @param graph graph de adyacencia
7   * @param c_mejor mejor coste encontrado
8   * @param s_mejor mejor solucion encontrada
9   * @param c_actual coste actual
10  * @param arco_menorpeso arco de menor peso del grafo (para no tener
11  * que calcularlo siempre)
12  * @param cota cota a utilizar
13  */
14 void tsp_backtracking(vector<int> &solucion, const vector<vector<double
15 >> &graph, double &c_mejor, vector<int> &s_mejor, double c_actual,
16 int arco_menorpeso, int cota = 0) {
17     c_actual += solucion.size() <= 1? 0: graph[solucion.back()][solucion[
18     solucion.size() - 2]];
19     if (solucion.size() == graph.size()) {
20         c_actual += graph[solucion.back()][solucion[0]];
21         if (c_actual < c_mejor) {
22             c_mejor = c_actual;
23             s_mejor = solucion;
24         }
25     } else {
26         for (int i = 1; i < graph.size(); i++) {
27             if (find(solucion.begin(), solucion.end(), i) == solucion.end())
28                 {

```



```

23     double acotacion;
24     if (cota == 1) {
25         acotacion = cota1(graph, solucion, c_actual, arco_menorpeso
26             );
27     } else if (cota == 2) {
28         acotacion = cota2(graph, solucion, c_actual);
29     } else if (cota == 3) {
30         acotacion = cota3(graph, solucion, c_actual);
31     } else {
32         acotacion = 0;
33     }
34     if (acotacion <= c_mejor) {
35         solucion.emplace_back(i);
36         tsp_backtracking(solucion, graph, c_mejor, s_mejor,
37             c_actual, arco_menorpeso, cota);
38         solucion.pop_back();
39     }
40 }
41 }

```

El algoritmo recibe como parametros una matriz de adyacencia , un vector de enteros que representa la solución actual, un vector de enteros que representa la mejor solución encontrada, un double que representa el mejor coste encontrado, un double que representa el coste actual, un entero que representa el arco de menor peso del grafo y un entero que representa la cota a utilizar. El vector de enteros solucion contiene el punto inicial sobre el que se aplica el algoritmo , si el vector solución está completo se comprueba si la solución obtenida es mejor que la que se tenía y se actualiza en caso de serlo. Si no está completo se calcula la cota local y se añade un nuevo punto al vector solución y se llama a la función recursivamente hasta que encuentra un camino completo.

4.2 Justificación

Para la justificación del Backtracking debemos comprobar que se obtiene la mejor solución posible, en nuestro caso, esa demostración la haremos usando reducción al absurdo:

1. Llamaremos n al conjunto de todos los nodos
2. Llamaremos C_l a la cota local
3. Llamaremos C a la cota global
4. La solución óptima se obtendrá tras haber explorado todas las ramas llegando al final, o no, dependiendo de los valores de las cotas

Para la demostración empezaremos del algoritmo supondremos que la solución K obtenida no es la solución óptima, esto implica que existe otra forma de explorar el grafo teniéndose que $K > K'$ siendo K' la solución óptima. Al devolver nuestro algoritmo K , la cota global contendrá precisamente este valor, cosa que nos lleva a una contradicción, ya que si una vez explorado todo el grafo hubiera una solución mejor que K , la cota global del programa deberá ser igual a ella, que en nuestro caso no ocurre, o bien porque la C_l ha hecho que se descarte la exploración de esa rama, o bien porque se exploró completamente sin mejorar la solución que ya teníamos.

4.3 Eficiencia teórica y empírica

4.3.1 Eficiencia empírica

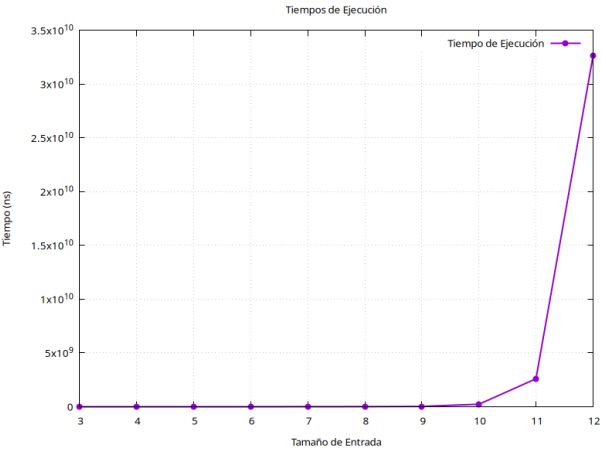


Figura 4.1: Cota 1

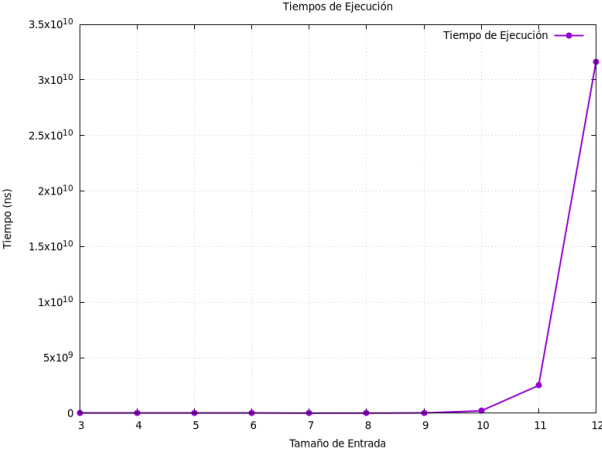


Figura 4.2: Cota 3

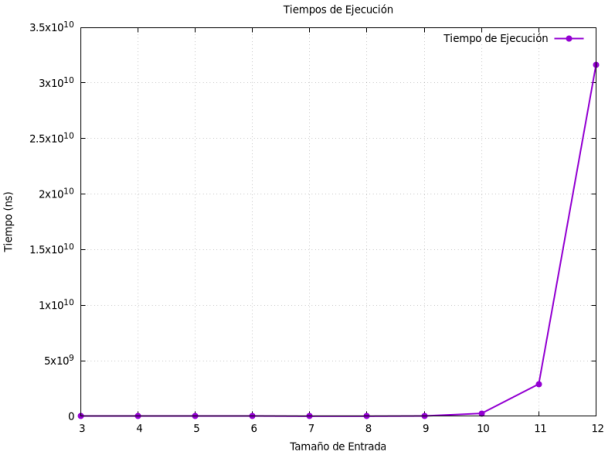


Figura 4.3: Cota 2

4.3.2 Eficiencia teórica

La eficiencia teórica del algoritmo es factorial, puesto las dos funciones de cota de dentro del código son $O(n)$, la primera es constante y la llamada recursiva es $n!$ luego al estar en el for ($O(n)$) obtenemos que la eficiencia teórica del algoritmo es de $O(n!)$

Branch and Bound

En esta sección, abordamos la resolución del problema del viajante utilizando el algoritmo Branch and Bound. A diferencia del método de backtracking empleado anteriormente, este enfoque permite una exploración más eficiente del espacio de soluciones. Describiremos las funciones de cota utilizadas y el algoritmo principal.

5.1 Diseño

5.1.1 Cota Global

Para la cota global, utilizamos la función "nearest neighbour" desarrollada en la práctica anterior. Esta función proporciona una cota superior inicial, que sirve como referencia para comparar las soluciones que nuestro algoritmo vaya generando. La cota global es crucial, ya que ayuda a reducir el número de nodos a explorar al proporcionar un límite superior sobre el cual los nodos con cotas inferiores son evaluados.

```

1  vector<int> nearest_neighbourTSP(const vector<vector<double>>&
    distancias, int inicial) {
2      int num_puntos = distancias.size();
3      vector<int> camino;
4      vector<bool> visitados(num_puntos, false);
5      camino.reserve(num_puntos);
6      camino.push_back(inicial);
7      visitados[inicial] = true;
8
9      for (int i = 0; i < num_puntos - 1; ++i) {
10         int actual = camino.back();
11         int siguiente = -1;
12         double min_distancia = numeric_limits<double>::max();
13
14         for (int j = 0; j < num_puntos; ++j) {
15             if (!visitados[j] && distancias[actual][j] < min_distancia) {
16                 min_distancia = distancias[actual][j];
17                 siguiente = j;
18             }
19         }
20
21         camino.push_back(siguiente);
22         visitados[siguiente] = true;
23     }
24

```

```

25     camino.push_back(inicial);
26     return camino;
27 }

```

5.1.2 Cota local

Además de la cota global, hemos implementado tres funciones de cota local diferentes para estimar de manera más precisa el costo de completar un camino parcial. Estas funciones nos ayudan a determinar si es prometedor continuar expandiendo un nodo dado en el árbol de soluciones.

- La primera cota local calcula el mínimo de cada fila en una matriz de distancias, excluyendo tanto los valores de 0 (que representan la distancia de un nodo a sí mismo) como los nodos específicos indicados en un vector de índices a ignorar. Este método asegura que no se considere volver a visitar los mismos nodos, proporcionando una suma de los mínimos costos posibles restantes.
- La segunda cota local opera de manera distinta. Encuentra la mínima distancia global en la matriz y multiplica este valor por el número de filas restantes, suponiendo que este mínimo es representativo del costo de conectar todos los nodos restantes. Aunque esta cota es más restrictiva, suele proporcionar una estimación más cercana al costo real, permitiendo podar más efectivamente el árbol de búsqueda.

Veamos la implementación de las 3:

```

1  double cota_inferior_1(const vector<vector<double>>& matriz, const
   vector<int>& indices_a_ignorar = {}) {
2      double suma_minimos = 0.0;
3
4      for (int i = 0; i < matriz.size(); ++i) {
5
6          if (find(indices_a_ignorar.begin(), indices_a_ignorar.end(), i)
              != indices_a_ignorar.end()) {
7              continue;
8          }
9
10         double minimo_fila = numeric_limits<double>::max();
11         for (int j = 0; j < matriz[i].size(); ++j) {
12
13             if (find(indices_a_ignorar.begin(), indices_a_ignorar.end()
14                     , j) != indices_a_ignorar.end()) {
15                 continue;
16             }
17             if (matriz[i][j] > 0 && matriz[i][j] < minimo_fila) {
18                 minimo_fila = matriz[i][j];
19             }
20         }
21
22         if (minimo_fila < numeric_limits<double>::max()) {
23             suma_minimos += minimo_fila;
24         }
25     }
26
27     return suma_minimos;
28 }
29
30 double cota_inferior_2(const vector<vector<double>>& matriz, const
   vector<int>& indices_a_ignorar = {}) {

```

```

30     double min_global = numeric_limits<double>::max();
31     int filas_contadas = 0;
32
33     for (int i = 0; i < matriz.size(); ++i) {
34
35         if (find(indices_a_ignorar.begin(), indices_a_ignorar.end(), i)
36             != indices_a_ignorar.end()) {
37             continue;
38         }
39
40         double minimo_fila = numeric_limits<double>::max();
41         for (int j = 0; j < matriz[i].size(); ++j) {
42
43             if (find(indices_a_ignorar.begin(), indices_a_ignorar.end(),
44                     j) != indices_a_ignorar.end()) {
45                 continue;
46             }
47             if (matriz[i][j] > 0 && matriz[i][j] < minimo_fila) {
48                 minimo_fila = matriz[i][j];
49             }
50
51             if (minimo_fila < numeric_limits<double>::max()) {
52                 if (minimo_fila < min_global) {
53                     min_global = minimo_fila;
54                 }
55                 ++filas_contadas;
56             }
57         }
58     }
59     return (min_global * filas_contadas);

```

Más adelante en el apartado de eficiencia seguiremos comparando dichas cotas.

5.1.3 Algoritmo Branch and Bound

Primero de todo, el struct con el que trabajaremos para almacenar la información sobre la cota mínima, el camino y la distancia recorrida será:

```

1     struct Nodo {
2         vector<int> path;
3         double distancia_recorrida;
4         double cota_inferior;
5
6         Nodo(vector<int>& inicial, double dist_rec, double cota_inf) {
7             path = inicial;
8             distancia_recorrida = dist_rec;
9             cota_inferior = cota_inf;
10        }
11    };

```

Ahora, el algoritmo Brach and Bound que hemos implementado ha sido:

```

1  vector<int> branch_and_bound_greedy(vector<int>& points, vector< vector
    <double>> &distancias, int inicial){
2      priority_queue<Nodo, vector<Nodo>, Comparador> no_visitados;
3      vector<int> mejor_camino = {inicial};
4      Nodo actual( mejor_camino, 0, cota_inferior(distancias));
5      no_visitados.push(actual);
6
7      double costo_minimo = calcularDistanciaTotal(distancias,
        nearest_neighbourTSP(distancias, inicial));
8      mejor_camino.clear();
9
10     while (!no_visitados.empty()) {
11         actual = no_visitados.top();
12         no_visitados.pop();
13
14         if (actual.path.size() == points.size()-1) {
15             vector<int> faltantes= numeros_faltantes(actual.path,
                points.size()-1);
16             actual.distancia_recorrida += distancias[actual.path.back()
                ][faltantes[0]];
17             actual.distancia_recorrida += distancias[faltantes[0]][
                inicial];
18             actual.path.push_back(faltantes[0]);
19             actual.path.push_back(inicial);
20             if (actual.distancia_recorrida <= costo_minimo) {
21                 costo_minimo = actual.distancia_recorrida;
22                 mejor_camino = actual.path;
23             }
24         }
25         else {
26             if (actual.cota_inferior <= costo_minimo ){
27                 vector<int> faltantes = numeros_faltantes(actual.path,
                    points.size()-1);
28                 for (int i = 0; i < faltantes.size(); ++i) {
29                     Nodo nuevo= actual;
30                     nuevo.path.push_back(faltantes[i]);
31                     nuevo.distancia_recorrida += distancias[actual.path
                        .back()][faltantes[i]];
32                     nuevo.cota_inferior = nuevo.distancia_recorrida +
                        cota_inferior(distancias, nuevo.path);
33                     no_visitados.push(nuevo);
34                 }
35             }
36         }
37     }
38
39     return mejor_camino;
40 }
41

```

5.1.4 Algoritmo Branch and Bound

El algoritmo Branch and Bound se basa en explorar sistemáticamente el espacio de soluciones posibles utilizando una estructura de árbol. Cada nodo en el árbol representa un estado parcial del problema, y el objetivo es

encontrar el camino de menor costo que visite todos los nodos exactamente una vez y regrese al nodo inicial.

Primero, definimos una estructura para almacenar la información de cada nodo, que incluye el camino recorrido hasta el momento, la distancia acumulada y la cota inferior calculada.

El algoritmo comienza creando un nodo inicial que contiene solo el nodo de partida. Este nodo se agrega a una cola de prioridad, que ordena los nodos en función de sus cotas inferiores. La cola de prioridad garantiza que siempre se explora primero el nodo con la menor cota inferior.

El algoritmo procede iterativamente extrayendo el nodo con la menor cota inferior de la cola de prioridad. Si el camino representado por este nodo incluye todos los nodos excepto uno, se completa el camino agregando el último nodo y retornando al inicial. Si la distancia total de este camino es menor que el costo mínimo conocido, se actualiza la mejor solución.

Si el nodo extraído no representa una solución completa, se generan nodos hijos para cada posible extensión del camino actual, agregando un nodo no visitado. Cada nuevo nodo hijo incluye el camino extendido, la nueva distancia acumulada y una cota inferior recalculada. Solo se agregan a la cola de prioridad aquellos nodos cuya cota inferior es menor o igual al costo mínimo conocido.

Notación:

- **Solución parcial:** El camino actual recorrido almacenado en el vector de nodos del nodo en cuestión.
- **Función poda:** Se activa cuando la cota inferior de un nodo supera el costo mínimo global conocido, eliminando así la necesidad de explorar este nodo.
- **Restricciones explícitas:** El siguiente nodo a seleccionar debe estar entre los nodos aún no visitados.
- **Restricciones implícitas:** El camino parcial de un nodo debe tener un costo menor que el costo mínimo global para ser considerado.
- **Árbol de estado:** El espacio de soluciones representado por todos los posibles caminos desde el nodo inicial hasta completar un circuito.
- **Estado del problema:** Cada nodo en el árbol de búsqueda.
- **Estado solución:** El nodo actual siendo evaluado.
- **Estado respuesta:** El mejor camino encontrado hasta el momento que cubre todos los nodos.
- **Nodo vivo:** Nodos que aún no han sido podados y pueden potencialmente llevar a una solución mejor.
- **Nodo muerto:** Nodos que han sido podados o totalmente explorados y no pueden mejorar la solución actual.
- **e-nodo:** El nodo actual que está siendo expandido.

5.2 Eficiencia teórica y empírica

5.2.1 Eficiencia teórica

En primer lugar, como ya calculamos en la práctica anterior, tenemos que la eficiencia de la cota global es de $O(n^2)$. En cuanto a las cotas locales tenemos que:

• Para la primera cota tenemos una eficiencia de $O(n^3)$. Esto se debe a que el bucle for interno se ejecuta n veces, y en dicho bucle se realiza un if con una función find que a su vez también tiene eficiencia $O(n)$, por lo que dicho bucle for tiene eficiencia $O(n^2)$. Además, tenemos que el bucle for exterior se ejecuta n veces, por lo que finalmente obtenemos una eficiencia $O(n^3)$.

• Para la segunda cota tenemos por el mismo razonamiento una eficiencia de $O(n^3)$.

Ahora veamos la eficiencia del algoritmo Branch and Bound. Empezando por la parte más interna del código, tenemos una estructura if-else. En dicha estructura, el bloque de sentencias if tienen una eficiencia $O(n \log(n))$ debido a que llama a la función números faltantes cuya eficiencia se ve claramente, debido a que se ejecuta en un bucle for n veces un if cuya condición es $O(\log(n))$.

```

1 vector<int> numeros_faltantes(const vector<int>& vec, int n) {
2
3     set<int> presentes(vec.begin(), vec.end());
4
5     vector<int> faltantes;
6     for (int i : views::iota(0, n + 1)) {
7         if (!presentes.contains(i)) {
8             faltantes.push_back(i);
9         }
10    }
11
12    return faltantes;
13 }
```

Por otro lado, tenemos para el else una eficiencia de $O(n^4)$, esto se debe a que hay un bucle for que se ejecuta n veces y en su cuerpo se llama a la función cota inferior que tiene una eficiencia de $O(n^3)$. De aquí obtenemos que la eficiencia del bloque if-else es $O(n^4)$. Como la condición del bucle while es $O(n!)$, tenemos que la eficiencia del método Branch and Bound es de $O(n! \cdot n^4)$.

5.2.2 Eficiencia empírica

En cuanto a la eficiencia empírica, podemos decir que aunque ambas cotas tienen el mismo orden de eficiencia, se observa claramente que a partir de la primera cota obtenemos mejores resultados.

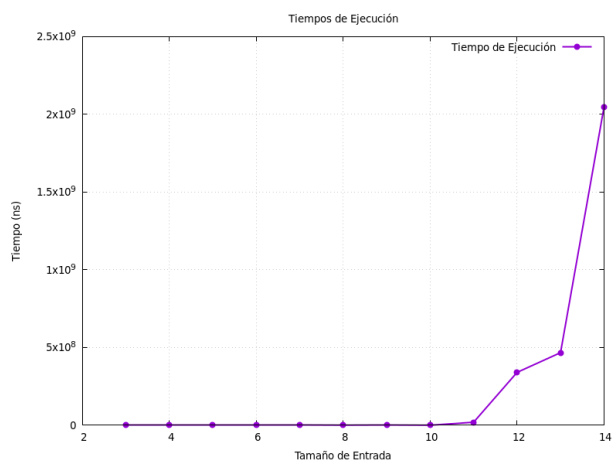


Figura 5.1: Cota 1

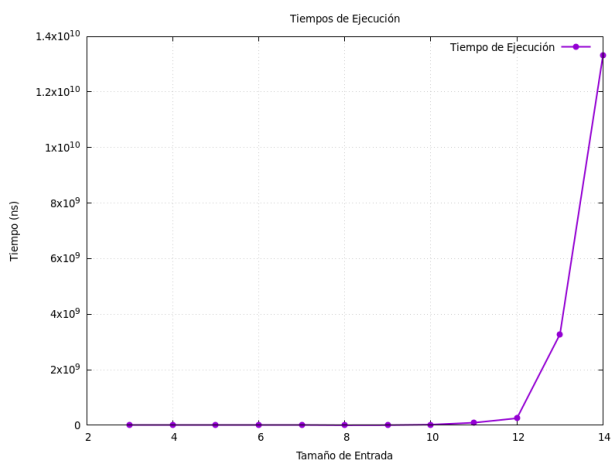


Figura 5.2: Cota 2

5.3 Justificación

Para demostrar que el algoritmo Branch and Bound que hemos creado nos proporciona una solución óptima, realizaremos una demostración por reducción al absurdo. Proposiciones:

1. Llamaremos N al conjunto de todos los nodos pertenecientes al último nivel
2. Llamaremos T al conjunto de todos los nodos pertenecientes a niveles superiores que están podados.
3. Llamaremos N_A al nodo que nos devuelve nuestro algoritmo
4. Llamaremos k_i al camino correspondiente de N_i
5. Llamaremos c_i a la cota inferior local correspondiente de N_i
6. Llamaremos C a la cota global
7. Supondremos que una solución es la óptima cuando en el último nivel del árbol de estado exista un camino tal que $k_i \leq k_j \quad \forall N_j \in N$ y que cumpla que $k_i \leq c_{ln} \quad c_{ln} \in T$ donde n indica el nivel de su último desarrollo, es decir, su rama está podada

Procedemos ahora a realizar la reducción al absurdo. Empezamos suponiendo que la solución devuelta no es la óptima, entonces pueden ocurrir dos cosas:

1. Existe un nodo en el último nivel al que llamaremos N_j tal que $N_j \in N$ que cumple que $k_j < k_A$ entonces N_j sería una mejor solución que N_A . De hecho, en particular, se cumpliría que $\exists N_j \in N$ tal que $k_l \leq k_i \quad \forall N_i \in N$. Por tanto, nuestro algoritmo hubiera devuelto N_j en vez de N_A y llegamos a una contradicción
2. Existiría un nodo $N_{jm} \in T$ tal que $c_{jm} < K_A = C$ del nivel m , donde la rama estaría podada por lo que llegamos a una contradicción, ya que debería seguir desarrollándose al ser menor que la cota global.

Como podemos ver en ambos casos llegamos a una contradicción que es lo que buscábamos

5.4 Gráficas

Finalmente, mostraremos varias ejecuciones del algoritmo del viajante mediante el Branch and Bound. Mostraremos gráficas realizadas con ambas cotas aunque eso no se ve reflejado en el resultado final sino más bien en el tiempo de ejecución como hemos mostrado previamente.

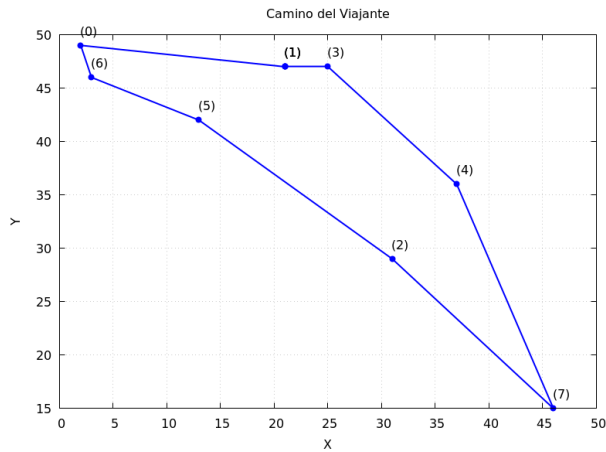


Figura 5.3: Cota 1, 8 puntos

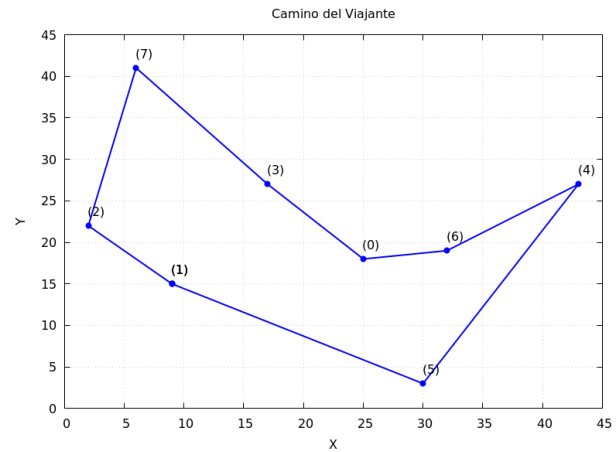


Figura 5.4: Cota 2, 8 puntos

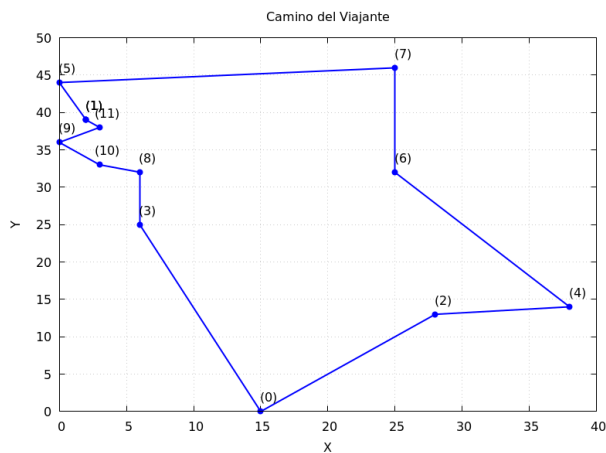


Figura 5.5: Cota 1, 12 puntos

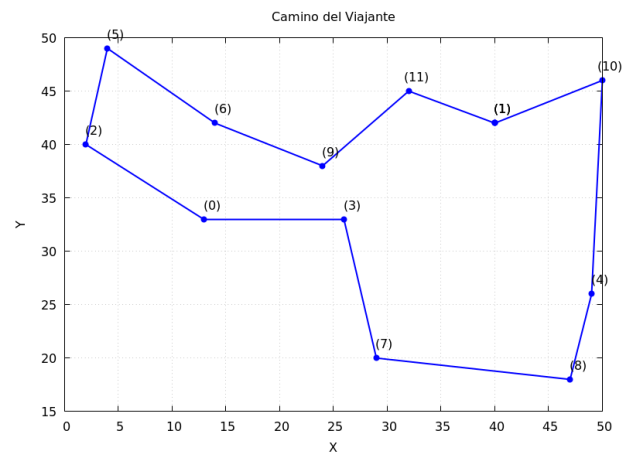


Figura 5.6: Cota 2, 12 puntos

A simple vista podemos apreciar que verdaderamente obtenemos resultados correctos independientemente de la cota utilizada, sin embargo, sigue siendo un factor a tener en cuenta debido a la diferencia de tiempo a la hora de la ejecución.