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

Apartado 1

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 computo)
  - 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 $i = 1, ..., n$. 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 exploraciónde grafos , backtracking y brannch and bound con el objetivo de ver las diferencias entre la eficiencia de estos dos algoritmos. Además se probarán distintas funciones de cota 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.

Apartado 4

Backtracking

4.1 Diseño

En esta sección analizaremos el algorimo 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>>&
        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  }
```

Esta funcion de la practica anterior la usaremos par 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 decision 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 , se multiplica por el numero de nodos restantes y se suman al coste actual.

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

```
1           double encontrarArcoMenorPeso(const vector<vector<double>> &
               graph) {
2               double minPeso = numeric_limits<double>::max();
3
4               for (int i = 0; i < graph.size(); ++i) {
5                 for (int j = i + 1; j < graph[i].size(); ++j) {
6                   if (graph[i][j] < minPeso) {
7                     minPeso = graph[i][j];
8                   }
9                 }
10              }
11
12              return minPeso;
13            }
```

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

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 codigo busca en una matriz de adyacencia los valores de los arcos de los nodos que no han sido todavia visitados , se seleccionan todos ellos y se ordenande menor a mayor para seleccionar los dos menores y hacer la media de estos.

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

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
         el orden
4    * en el que es visitada
5    * @param graph graph de adyacencia
6    * @param c_mejor mejor coste encontrado
7    * @param s_mejor mejor solucion encontrada
8    * @param c_actual coste actual
9    * @param arco_menorpeso arco de menor peso del grafo (para no tener
         que calcularlo siempre)
10   *@param cota cota a utilizar
```

```
11  */
12  void tsp_backtracking(vector<int> &solucion,const vector<vector<double
       >> &graph, double &c_mejor,vector<int> &s_mejor, double c_actual,
       int arco_menorpeso,int cota) {
13   if (solucion.size() == graph.size()) {
14     c_actual += graph[solucion.back()][solucion[0]];
15     if (c_actual < c_mejor) {
16       c_mejor = c_actual;
17       s_mejor = solucion;
18     }
19   } else {
20     for (int i = 1; i < graph.size(); i++) {
21       if (find(solucion.begin(), solucion.end(), i) == solucion.end())
            {
22         double acotacion;
23         double c_siguiente = c_actual + (solucion.empty() ? 0 : graph[
              solucion.back()][i]);
24         if (cota == 1) {
25             acotacion = cota1(graph, solucion, c_actual, arco_menorpeso
                  );
26         } else if (cota == 2) {
27             acotacion = cota2(graph, solucion, c_actual);
28         } else if (cota == 3) {
29             acotacion = cota3(graph, solucion, c_actual);
30         } else {
31             acotacion = 0;
32         }
33         if (acotacion <= c_mejor) {
34             solucion.emplace_back(i);
35             tsp_backtracking(solucion, graph, c_mejor, s_mejor,
                  c_siguiente,arco_menorpeso, cota);
36             solucion.pop_back();
37           }
38       }
39     }
40   }
41 }
```

El algoritmo recibe como parámetros 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 algorimo 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 algorimo 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

A continuación se mostrarán los resultados obtenidos en la ejecución del algoritmo Backtracking con las distintas funciones de cota y como afecta cada una de ellas a la generación de nodos y al tiempo de ejecución.

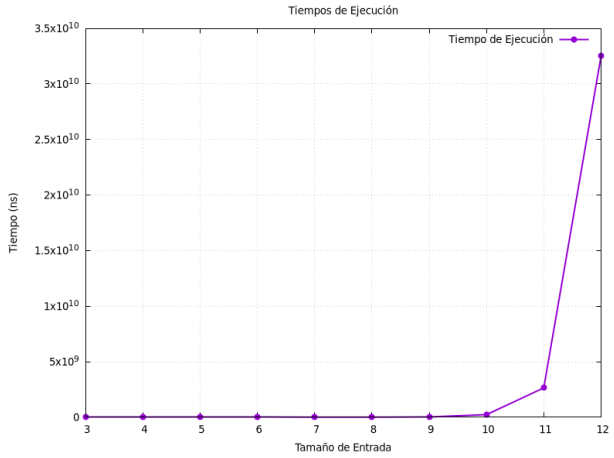

Figura 4.1: Sin cota

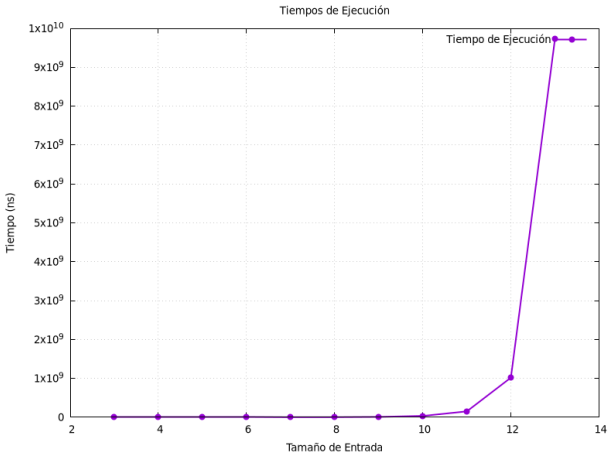

Figura 4.2: Cota 1

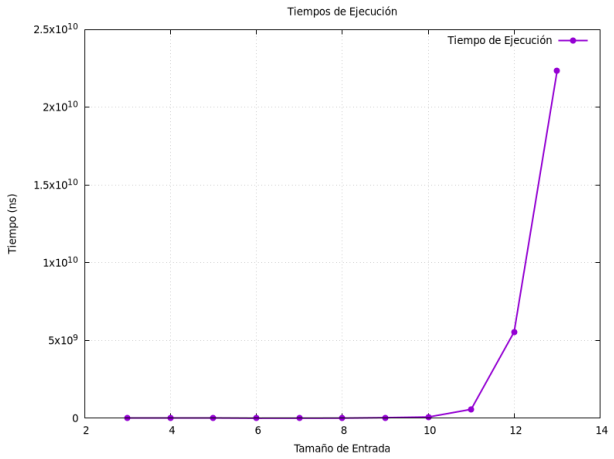

Figura 4.3: Cota 2

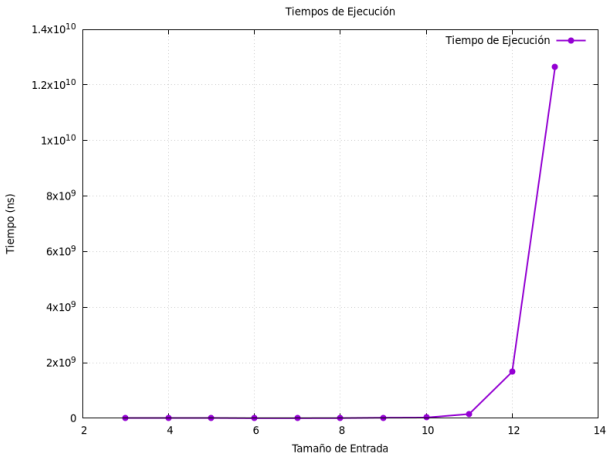

Figura 4.4: Cota 3

4.3.2 Caminos generados y comparación de nodos generados

Graficas de Caminos generados Aquí podremos ver los caminos que han sido representados con el algoritmo usando las diferentes cotas al igual que una tabla con la comparación de los valores obtenidos de las ejecuciones

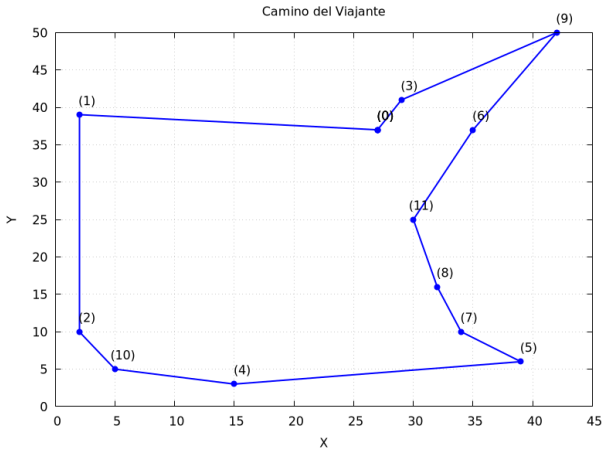

Figura 4.5: Cota 1

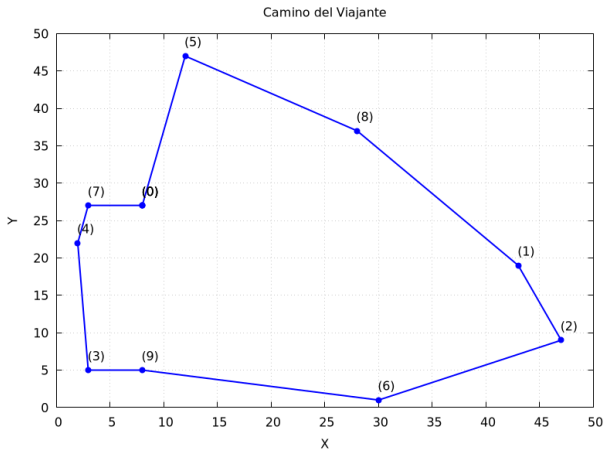

Figura 4.6: Cota 2 camino

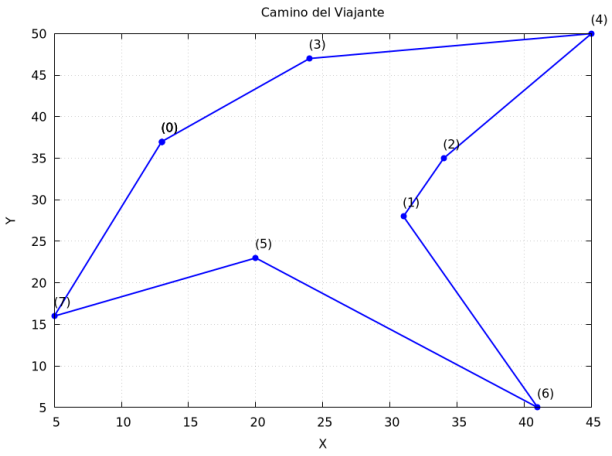

Figura 4.7: Cota 3 camino

Comparación de nodos generados En las siguientes tablas se comparan el numero de nodos generados en cada ejecución del algoritmo con las distintas funciones de cota.

COTA 0			COTA 1			COTA 2			COTA 3		
N PUNTOS	MEJOR SOL	NODOS	N PUNTOS	MEJOR SOL	NODOS	N PUNTOS	MEJOR SOL	NODOS	N PUNTOS	MEJOR SOL	NODOS
3	102.531	4	3	102.531	4	3	102.531	4	3	102.531	4
4	105.104	15	4	105.104	15	4	105.104	15	4	105.104	15
5	111.937	64	5	111.937	54	5	111.937	58	5	111.937	58
6	94.141	325	6	94.141	204	6	94.141	212	6	94.141	183
7	156.288	1956	7	156.288	631	7	156.288	917	7	156.288	708
8	154.06	13699	8	154.06	7252	8	154.06	5918	8	154.06	5003
9	116.596	109600	9	116.596	13490	9	116.596	9340	9	116.596	5396
10	146.742	986409	10	146.742	27306	10	146.742	7420	10	146.742	1456
11	185.693	9864100	11	185.693	719932	11	185.693	300273	11	185.693	199639
12	164.292	108505111	12	164.292	915352	12	164.292	174285	12	164.292	73504

Figura 4.8: Tabla nodos BK

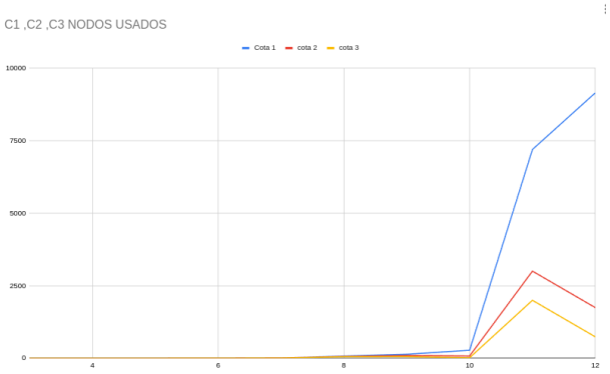

Figura 4.9: Sin cota

Como podemos observar en el grafico de la creación de nodos la cota 2 y 3 generan muchos menos nodos que la cota 1 por tanto son mas eficientes a la hora de completar el camino ya que descartan muchas ramas que no nos aportan una solución mejor que la que ya tenemos. Por otro lado la cota 1 disminuye los nodos en cuanto a usar el algoritmo sin cota pero sigue generando muchos nodos que no nos aportan una solución mejor que la que ya tenemos.

4.3.3 Eficiencia teórica

La eficiencia teórica del algorimo 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 $O(n!)$

Apartado 5

Branch and bound

Al igual que en el apartado anterior resolveremos el problema del viajante, solo que en vez de solucionar el problema mediante el bakctraking lo resolveremos mediante Branch and bound. A continuación se mostrarán las funciones de cota al igual que el algoritmo principal que hemos usado:

5.1 Diseño

5.1.1 Cota global

Para selecionar la cota global usaremos la función de nearest neighbour de la práctica anterior la cual nos proporciona una cota principal con la que poder comparar las distintas soluciones que vaya obteniendo nuestro algoritmo. La idea se basa en dado un conjunto de puntos queremos encontrar el camino más corto que los recorra, para ello, elegimos un punto inicial y a partir de ese punto, en un proceso iterativo, elegimos en cada paso el siguiente punto más cercano al actual, o lo que es lo mismo, el punto para el cual la distancia al punto actual es la menor. Este proceso se repite hasta que todos los puntos han sido visitados.

```
vector<int> nearest_neighbourTSP(const vector<vector<double>>&
   distancias, int inicial) {
    int num_puntos = distancias.size();
    vector<int> camino;
    vector<bool> visitados(num_puntos, false);
    camino.reserve(num_puntos);
    camino.push_back(inicial);
    visitados[inicial] = true;

    for (int i = 0; i < num_puntos - 1; ++i) {
      int actual = camino.back();
      int siguiente = -1;
      double min_distancia = numeric_limits<double>::max();

      for (int j = 0; j < num_puntos; ++j) {
          if (!visitados[j] && distancias[actual][j] < min_distancia) {
              min_distancia = distancias[actual][j];
              siguiente = j;
          }
      }

      camino.push_back(siguiente);
      visitados[siguiente] = true;
```

```
    }

    camino.push_back(inicial);
    return camino;
}
```

5.1.2 Cota local

Además de esta cota global, hemos implementado tres funciones de cota local distintas:

```
double cota_inferior_1(const std::vector<std::vector<double>>& matriz,
   const std::vector<int>& indices_a_ignorar = {}) {
        double suma_minimos = 0.0;

    for (int i = 0; i < matriz.size(); ++i) {

        if (std::find(indices_a_ignorar.begin(), indices_a_ignorar.end
           (), i) != indices_a_ignorar.end()) {
            continue;
        }

        double minimo_fila = std::numeric_limits<double>::max();
        for (int j = 0; j < matriz[i].size(); ++j) {

            bool condicion_2=find(indices_a_ignorar.begin(),
               indices_a_ignorar.end(), j) != indices_a_ignorar.end();

            if (!condicion_2 && matriz[i][j] > 0 && matriz[i][j] <
               minimo_fila) {
                minimo_fila = matriz[i][j];
            }
        }

        suma_minimos += minimo_fila;
    }

    return suma_minimos;
}
```

La primera función de cota consiste en a partir de los índices a ignorar que hemos pasado como parámetro se ignoran la fila y columna que lo contengan. Luego a partir de las filas restantes obtenemos el mínimo y se lo sumamos a la variable *suma_mnimos*. Repetimos este proceso hasta recorrerlas todas.

```

double cota_inferior_2(const std::vector<std::vector<double>>& matriz,
   const std::vector<int>& indices_a_ignorar = {}) {
    double min_global = std::numeric_limits<double>::max();
    int filas_contadas = 0;

    for (int i = 0; i < matriz.size(); ++i) {
        bool condicion_1=find(indices_a_ignorar.begin(),
           indices_a_ignorar.end(), i) != indices_a_ignorar.end();


```

```
        if (condicion_1) {
            continue;
        }

        double minimo_fila = std::numeric_limits<double>::max();
        for (int j = 0; j < matriz[i].size(); ++j) {
            bool condicion_2=find(indices_a_ignorar.begin(),
               indices_a_ignorar.end(), j) != indices_a_ignorar.end();

            if (!condicion_2 && matriz[i][j] > 0 && matriz[i][j] <
               minimo_fila) {
                minimo_fila = matriz[i][j];
            }
        }

        if (minimo_fila < min_global) {
                min_global = minimo_fila;
        }
        ++filas_contadas;
    }

    return  (min_global * filas_contadas);
}
```

La segunda función de cota consiste principalmente en encontrar la mínima distancia de una matriz de adyacencia y depués multiplicar dicho valor por las filas restantes, suponiendo que ese camino será el mínimo para todos.

```
double cota_inferior_3(const std::vector<std::vector<double>>& matriz,
   const std::vector<int>& indices_a_ignorar = {}) {
    double suma_costos = 0.0;

    for (int i = 0; i < matriz.size(); ++i) {

        if (find(indices_a_ignorar.begin(), indices_a_ignorar.end(),i)
           != indices_a_ignorar.end()) {
            continue;
        }

        std::vector<double> costos_fila;
        for (int j = 0; j < matriz[i].size(); ++j) {
            costos_fila.push_back(matriz[i][j]);
        }


        if (costos_fila.size() < 2) {
            continue;
        }

        std::sort(costos_fila.begin(), costos_fila.end());

        double costo1 = costos_fila[0];
        double costo2 = costos_fila[1];

        suma_costos += (costo1 + costo2) / 2.0;
    }
```

```
27
28      return suma_costos;
29  }
```

Por último, la tercera función de cota consiste en obviar las filas que se encuentran dentro de las filas a ignorar, en este caso, a diferencia de la cota 1 no obviamos las columnas. De las filas disponibles se cogen los dos valores más pequeños a través del sort que las ordena, habiendo comprobado previamente que hay más de dos valores disponibles para que no se produzcan errores. Con esos dos valores hace la media y se la suma a la variable *suma_costos*.

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, int &nodos_desarrollados,
2  double (*funcion_cota)(const vector<vector<double>>& matriz, const std
       ::vector<int>& indices_a_ignorar)){
3      priority_queue<Nodo, vector<Nodo>, Comparador> no_visitados;
4      vector<int> mejor_camino = {inicial};
5      vector <int> auxiliar={};
6      Nodo actual( mejor_camino, 0, funcion_cota(distancias, auxiliar));
7      no_visitados.push(actual);
8
9      double costo_minimo = calcularDistanciaTotal(distancias,
           nearest_neighborTSP(distancias, inicial));
10     mejor_camino.clear();
11
12     while (!no_visitados.empty()) {
13         actual = no_visitados.top();
14         no_visitados.pop();
15
16         if (actual.path.size() == points.size()-1) {
17             nodos_desarrollados++;
18             vector<int> faltantes= numeros_faltantes(actual.path,
                   points.size()-1);
19             actual.distancia_recorrida += distancias[actual.path.back()
                   ][faltantes[0]];
```

```
20              actual.distancia_recorrida += distancias[faltantes[0]][
                    inicial];
21              actual.path.push_back(faltantes[0]);
22              actual.path.push_back(inicial);
23              if (actual.distancia_recorrida <= costo_minimo) {
24                  costo_minimo = actual.distancia_recorrida;
25                  mejor_camino = actual.path;
26              }
27          }
28          else {
29              if (actual.cota_inferior <= costo_minimo ){
30                  nodos_desarrollados++;
31                  vector<int> faltantes = numeros_faltantes(actual.path,
                        points.size()-1);
32                  for (int i = 0; i < faltantes.size(); ++i) {
33                      Nodo nuevo= actual;
34                      nuevo.path.push_back(faltantes[i]);
35                      nuevo.distancia_recorrida += distancias[actual.path
                            .back()][faltantes[i]];
36                      nuevo.cota_inferior = nuevo.distancia_recorrida +
                            funcion_cota(distancias, nuevo.path);
37                      no_visitados.push(nuevo);
38                  }
39              }
40          }
41      }
42
43      return mejor_camino;
44
45  }
```

El funcionamiento de este método se basa en empezar creando un nodo con el punto inicial y agregarlo a la cola de prioridad (cuyo comparador consiste en comparar la distancia recorrida por un nodo a y un nodo b y ver qué distancia es menor).Luego calculamos una cota global inicial que nos ayudará a reducir considerablemente el número de caminos posibles. Mientras la cola no esté vacía, tomamos como nodo actual el menor de todos los disponibles en la cola de prioridad:

·En el caso de que hayamos recorridos todos los puntos menos uno, agregamos la distancia del último nodo que tenemos al nodo faltante y la distancia del nodo faltante al primero del camino, creando así un camino cerrado que recorra todos los puntos. En el caso de que este camino recorrido sea mejor que la cota global actualizamos la variable *mejor_camino* que es lo que devolveremos una vez termina el bucle y la cota global.

·En el caso en que no hayamos recorridos todos los nodos menos uno, comparamos la cota inferior suministrada por el nodo actual con la cota global. En el caso de que dicha cota inferior sea menor podemos seguir ejecutando el código sino, podamos. Una vez visto que la cota inferior es menor que la cota global, añadimos los nodos faltantes a la cola de prioridad y volvemos a realizar los mismos pasos cogiendo el nodo con mejor camino de la cola de prioridad.

Cabe destacar que en nuestra función Branch and Bound hemos decidido pasar como parámetro una función que nos permita elegir entre las distintas cotas disponibles, lo cual nos proporciona un código más funcional.

A continuación se mostrará la notación dada en teoría, aunque sigamos un esquema similar, hay ciertos ámbitos que pueden ser suprimidos.

Notación:

- **Solución parcial:** Vector path del Nodo actual.
- **Función poda:** Momento en que la cota inferior supera al costo minimo,siendo este la cota global.
- **Restricciones explícitas:**Que el siguiete nodo a seleccionar este dentro de los nodos faltantes
- **Restricciones implícitas:**Que el camino del nodo actual sea menor que el costo minimo
- **Árbol de estado:** El espacio solución con el que se trabaja es el Nodo actual, sin embargo, el vector mejor camino es el árbol de estado al encontrar la solución.
- **Estado del problema:** Cada uno de los nodos del árbol
- **Estado solución:** Nodo actual
- **Estado respuesta:**Vector mejor camino
- **Nodo vivo:** Nodos que todavia no hemos podado y pueden darnos una solución en el caso de que el nodo actual con el que estamos trabajando supere el valor del costo mínimo
- **Nodo muerto:**Nodo podado o que hemos recorrido todo su camino
- **e-nodo:**Nodo actual

5.2 Eficiencia teórica y empírica

5.2.1 Eficiencia teórica

En primer lugar, como ya calculamos en la práctica anterior, tenemos que la eficiencai 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 $O(n^3)$.

·Para la tercera cota a diferencia de las dos cotas anteriores tenemos una eficiencia de $O(n^2)$, que es fácil de observar. 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 $O(nlog(n))$ debido a que llama a la funcion numeros faltantes cuya eficiencia se ve claramente, debido a que se ejecuta en un bucle for n veces un if cuya condición es $O(log(n))$.

```
vector<int> numeros_faltantes(const vector<int>& vec, int n) {

      set<int> presentes(vec.begin(), vec.end());

      vector<int> faltantes;
      for (int i : views::iota(0, n + 1)) {
            if (!presentes.contains(i)) {
                faltantes.push_back(i);
            }
      }

      return faltantes;
}
```

Por otro lado, tenemos para el else una eficiencia de $O(n^4)$ o $O(n^3)$, 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 $O(n^3)$ o $O(n^2)$. De aquí obtenemos que la eficiencia del bloque if-else es $O(n^4)$o $O(n^3)$. Sin embargo, calculamos como la condición del bucle while es $O(n!)$, tenemos que la eficiencia del método Branch and Bound es de $O(n!)$.

5.2.2 Eficiencia empírica

En cuanto a la eficiencia empírica, podemos observar como las cotas 1 y 2 desarrollan muchos menos nodos que la cota 3. Sin embargo, debido a que la eficiencía de las cotas 1 y 2 es mayor que la de la cota 3 no se ven afectados los tiempos de ejecución de unas cotas respecto de la otra. A continuación, mostraremos distintas gráficas que lo demuestran.

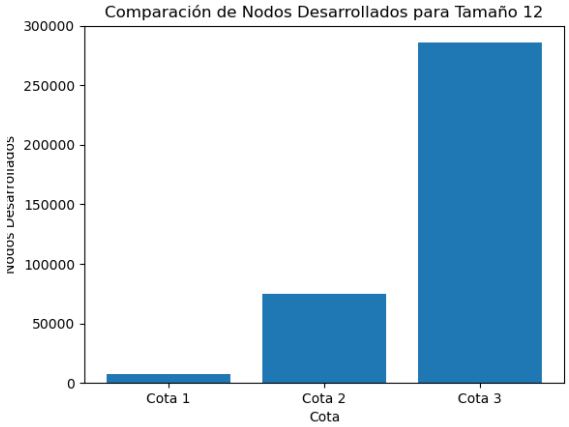

Figura 5.1: Comparación nodos

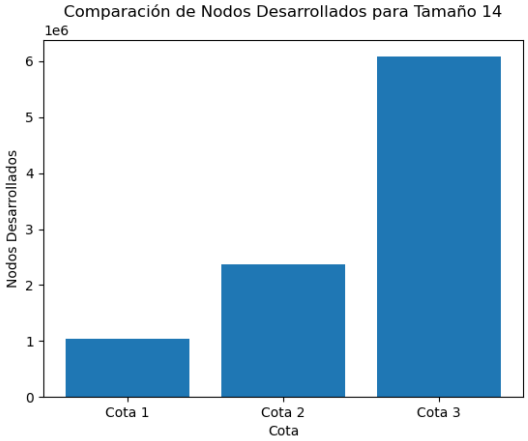

Figura 5.2: Comparación nodos

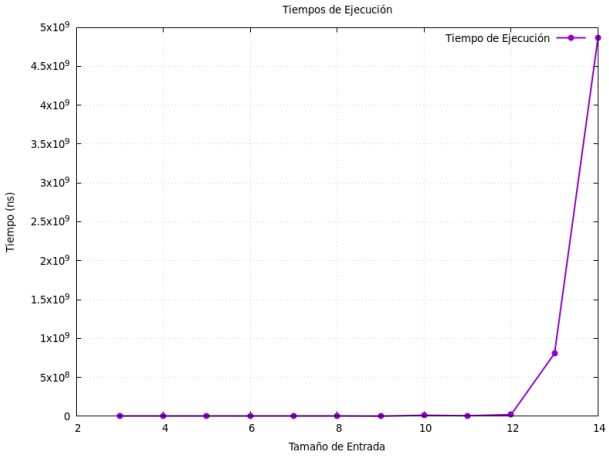

Figura 5.3: Cota 1

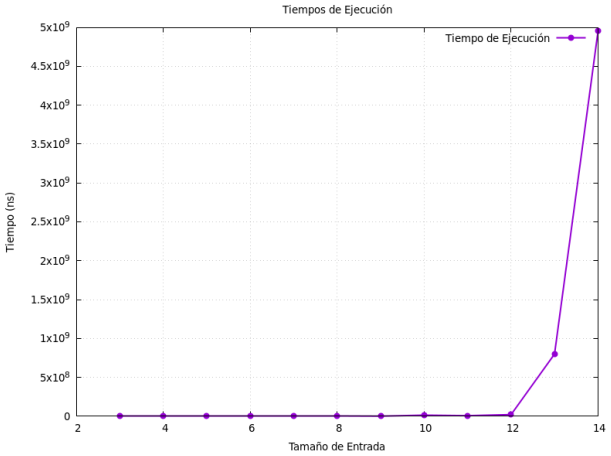

Figura 5.4: Cota 2

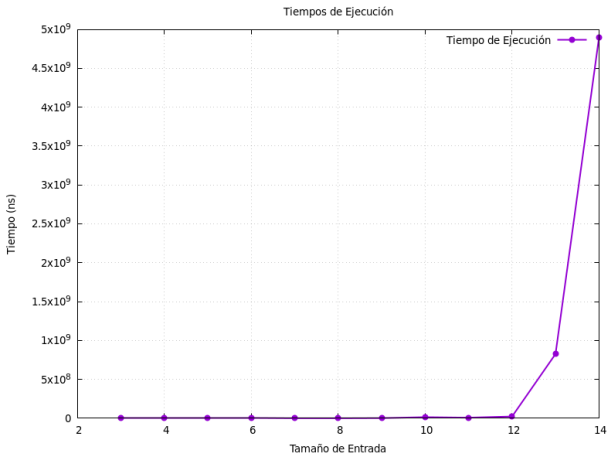

Figura 5.5: Cota 3

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. Llamaremo 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 ultimo desarrollo, es decir, su rama esta podada

Procedemos ahora a realizar la reduccioón 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 prticular, se cumpliría que $\exists N_j \in N$tal que $k_l \leq k_i \ \forall N_i \in N$. 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 desarrollandose al ser menor que la cota global.

Como podemos ver en ambos casos llegamos a contracción 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 y el número de nodos desarrollados,como hemos mostrado previamente.

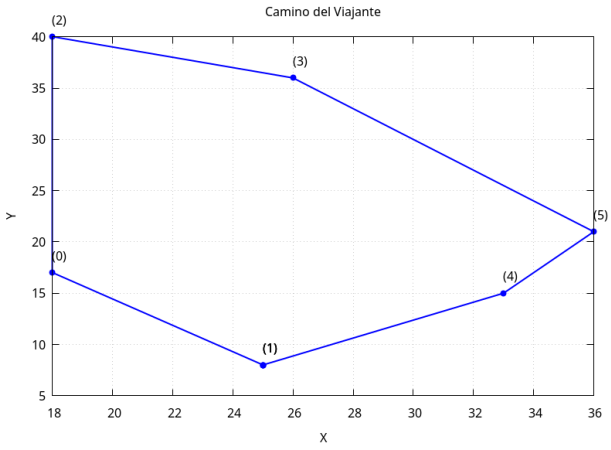

Figura 5.6: Cota 1

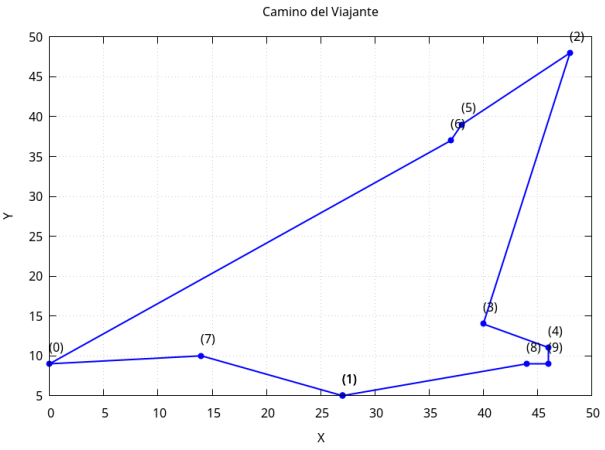

Figura 5.7: Cota 2

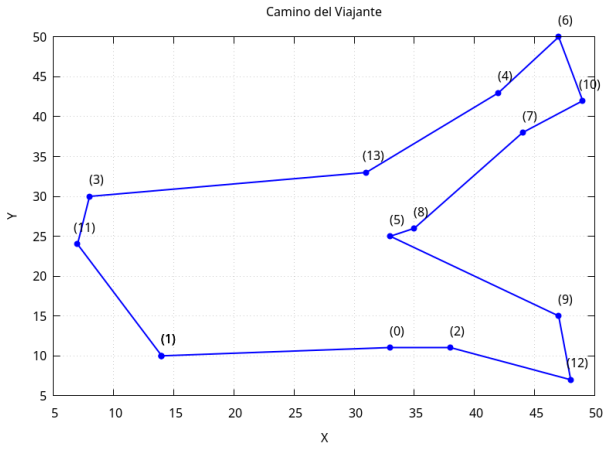

Figura 5.8: Cota 3

A simple vista podemos apreciar que verdaderamenta obtenemos resultados correctos independientemente de la cota utilizada.