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 cotsa 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 disatancia.

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

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 codigose 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.

```
        double cota3(const vector<vector<double>> &graph, const vector<
           int> &solucion,double c_actual) {
            double cota = 0;
            vector<double> v;
            double min = numeric_limits<double>::max();
            for (int i = 1; i < graph.size(); ++i) {
                if ((find(solucion.begin(), solucion.end(), i) ==
                   solucion.end())) {
                    v = graph[i];
                    sort(v.begin(), v.end());
                    cota += (v[1] + v[2]) / 2;
                }
            }
            return cota + c_actual;
        }
```

4.1.3 Algoritmo Backtracking

A continuación se muestra el algoritmo de Backtracking implementado:

```
/**
 * @brief Funcion para resolver el tsp con backtracking .
 * @param solucion vector de ciudades , la posicion de la ciudad indica
     el orden
 * en el que es visitada
 * @param graph graph de adyacencia
 * @param c_mejor mejor coste encontrado
 * @param s_mejor mejor solucion encontrada
 * @param c_actual coste actual
 * @param arco_menorpeso arco de menor peso del grafo (para no tener
    que calcularlo siempre)
 *@param cota cota a utilizar
 */
 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 = 0) {
  c_actual += solucion.size() <= 1? 0: graph[solucion.back()][solucion[
     solucion.size() - 2]];
  if (solucion.size() == graph.size()) {
    c_actual += graph[solucion.back()][solucion[0]];
    if (c_actual < c_mejor) {
      c_mejor = c_actual;
      s_mejor = solucion;
    }
  } else {
    for (int i = 1; i < graph.size(); i++) {
      if (find(solucion.begin(), solucion.end(), i) == solucion.end())
         {
```

```
23          double acotacion;
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_actual,arco_menorpeso, cota);
36              solucion.pop_back();
37            }
38        }
39      }
40    }
41  }
```

El algoritmo recive 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 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

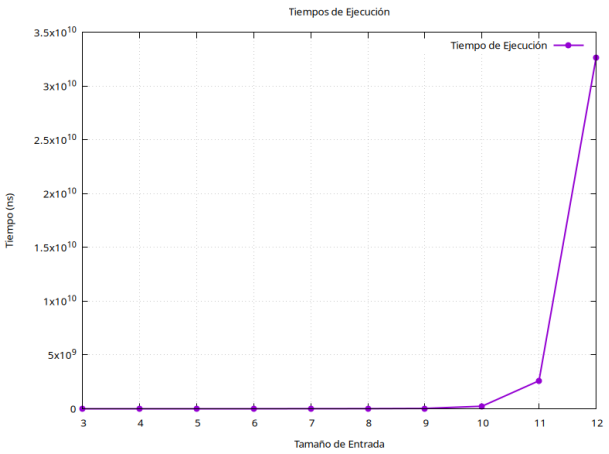

Figura 4.1: Cota 1

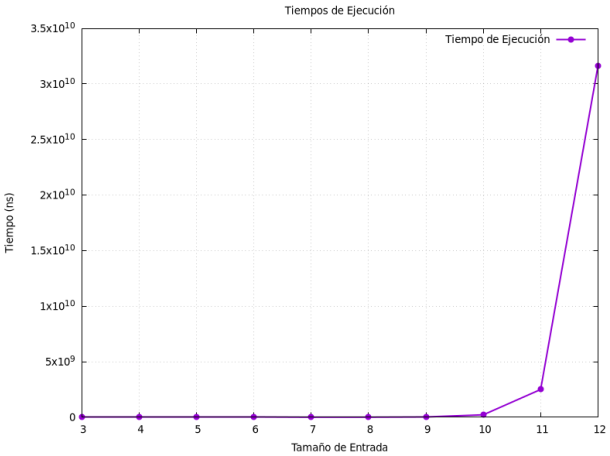

Figura 4.2: Cota 3

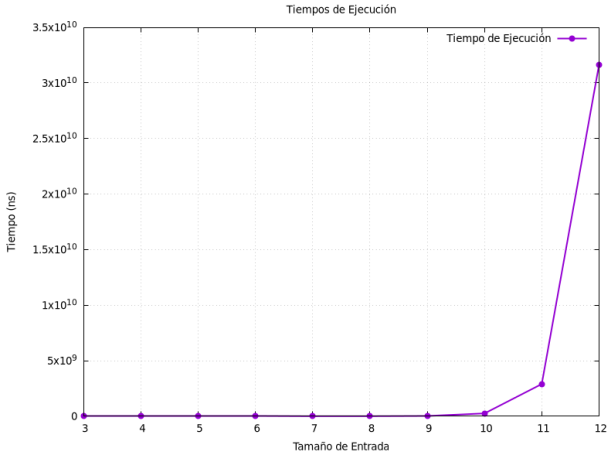

Figura 4.3: Cota 2

4.3.2 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.

```
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 dos funciones de cota local distintas:

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

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

            if (find(indices_a_ignorar.begin(), indices_a_ignorar.end()
                , j) != indices_a_ignorar.end()) {
                continue;
            }
            if (matriz[i][j] > 0 && matriz[i][j] < minimo_fila) {
                minimo_fila = matriz[i][j];
            }
        }

        if (minimo_fila < numeric_limits<double>::max()) {
            suma_minimos += minimo_fila;
        }
    }

    return suma_minimos;
}

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

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

```

```
42              if (find(indices_a_ignorar.begin(), indices_a_ignorar.end()
                    , j) != indices_a_ignorar.end()) {
43                  continue;
44              }
45              if (matriz[i][j] > 0 && matriz[i][j] < minimo_fila) {
46                  minimo_fila = matriz[i][j];
47              }
48          }
49
50          if (minimo_fila < numeric_limits<double>::max()) {
51              if (minimo_fila < min_global) {
52                  min_global = minimo_fila;
53              }
54              ++filas_contadas;
55          }
56      }
57
58      return  (min_global * filas_contadas);
59  }
```

La primera función de cota local consiste en calcular el mínimo de una matriz de distancias, excluyendo obviamente los valores de 0, pasandose como parámetros los índices a ignorar.Destacar que además de obviar dichas filas se obviarán sus respectivas columnas indicadas en el vector de enteros, ya que el objetivo es eliminar las conexiones de ciertos nodos, por lo que eliminamos sus filas y columnas respectivas.

Por otro lado, la segunda función de cota, tiene un funcionamiento distinto. 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, de ahí que se multiplique por el número de nodos restantes. Ignorando, igual que con la cota anterior, los 0 y las filas y las columnas que se pasan como parámetros a ignorar. Función para calcular cota inferior calculando el mínimo de una matriz de distancias, excluyendo el 0 y las filas. Es claro que este segundo método nos proporciona una cota más restrictiva por lo que nos interesa más.

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:

```
vector<int> branch_and_bound_greedy(vector<int>& points, vector< vector
   <double>> &distancias, int inicial){
    priority_queue<Nodo, vector<Nodo>, Comparador> no_visitados;
    vector<int> mejor_camino = {inicial};
    Nodo actual( mejor_camino, 0, cota_inferior(distancias));
    no_visitados.push(actual);

    double costo_minimo = calcularDistanciaTotal(distancias,
       nearest_neighbourTSP(distancias, inicial));
    mejor_camino.clear();

    while (!no_visitados.empty()) {
        actual = no_visitados.top();
        no_visitados.pop();

        if (actual.path.size() == points.size()-1) {
            vector<int> faltantes= numeros_faltantes(actual.path,
               points.size()-1);
            actual.distancia_recorrida += distancias[actual.path.back()
               ][faltantes[0]];
            actual.distancia_recorrida += distancias[faltantes[0]][
               inicial];
            actual.path.push_back(faltantes[0]);
            actual.path.push_back(inicial);
            if (actual.distancia_recorrida <= costo_minimo) {
                costo_minimo = actual.distancia_recorrida;
                mejor_camino = actual.path;
            }
        }
        else {
            if (actual.cota_inferior <= costo_minimo ){
                vector<int> faltantes = numeros_faltantes(actual.path,
                   points.size()-1);
                for (int i = 0; i < faltantes.size(); ++i) {
                    Nodo nuevo= actual;
                    nuevo.path.push_back(faltantes[i]);
                    nuevo.distancia_recorrida += distancias[actual.path
                       .back()][faltantes[i]];
                    nuevo.cota_inferior = nuevo.distancia_recorrida +
                       cota_inferior(distancias, nuevo.path);
                    no_visitados.push(nuevo);
                }
            }
        }
    }

    return mejor_camino;

}
```

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). Mientras la cola no esté vacía; primero, se extrae el nodo con menor costo actual (a partir de la cota inferior y haciendo uso del comparador definido anteriormente); si el camino actual tiene todos los puntos menos unos, sabemos de forma determinada cual queda mediante la función faltantes de antes(dicha fun-

ción encuentra los números faltantes desde 0 hasta n en un vector de enteros), por lo que se calcula la distancia a dicho punto y al inicial y se agrega a la solución si es mejor que la actual; si no, se generan los nodos hijos con los puntos faltantes y se agregan a la cola de prioridad.

A continuación se mostrará la notación mostrada 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 $O(n^3)$.

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 $O(n^4)$, 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.

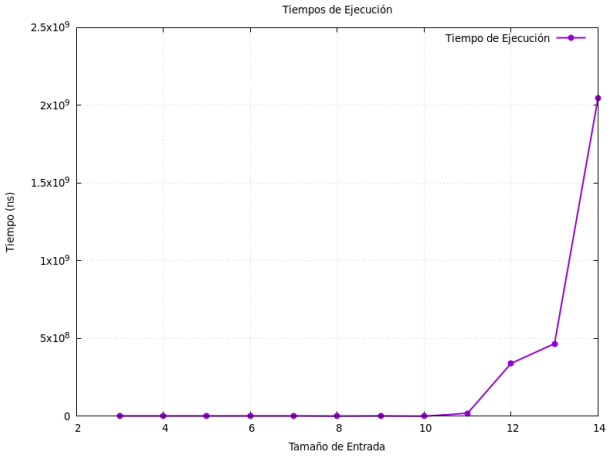

Figura 5.1: Cota 1

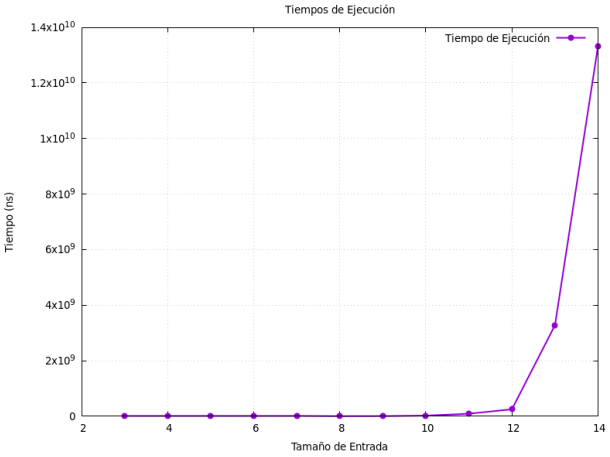

Figura 5.2: Cota 2

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. 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 como hemos mostrado previamente.

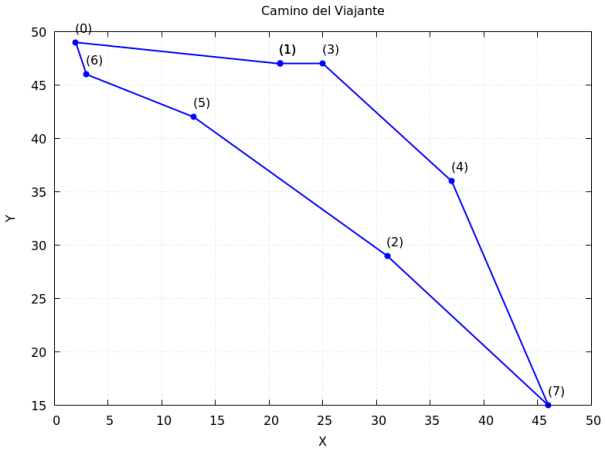

Figura 5.3: Cota 1, 8 puntos

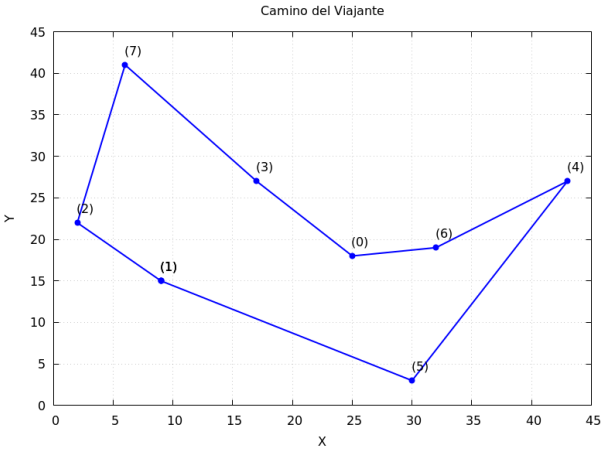

Figura 5.4: Cota 2, 8 puntos

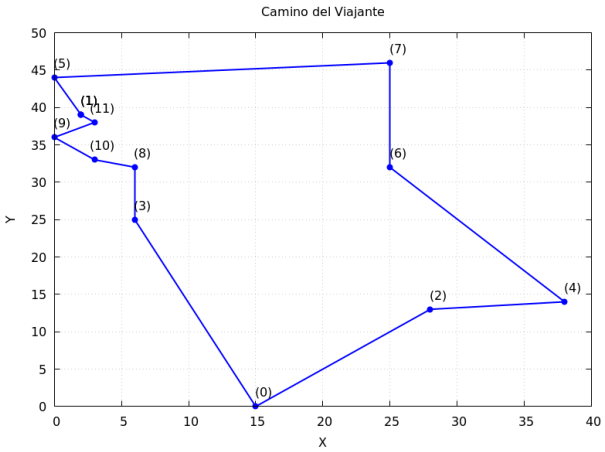

Figura 5.5: Cota 1, 12 puntos

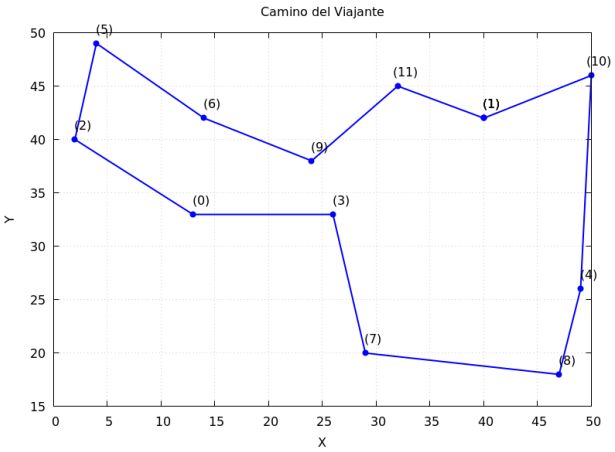

Figura 5.6: Cota 2, 12 puntos

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