

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	6
4.1.3 Algoritmo Backtracking	7
4.2 Justificación	7
4.3 Eficiencia teórica y empírica	7
4.3.1 Eficiencia teórica	7
5 Branch and bound	8
5.1 Diseño	8
5.1.1 Cota global	8
5.1.2 Cota local	9
5.1.3 Algoritmo Branch and Bound	10
5.2 Eficiencia teórica y empírica	13
5.2.1 Eficiencia teórica	13
5.2.2 Eficiencia empírica	13
5.3 Justificación	15
5.4 Gráficas	16

Autores

- **Miguel Ángel De la Vega Rodríguez: 20%**
 - Algoritmos Greedy del Viajante
 - Redacción memoria sección Viajante
- **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%**
 - Programación hijo predilecto
 - Redacción hijo predilecto
 - Conclusión
- **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 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

4.1.2 Cota local

Primera función de cota:

La primera función de cota implementada consiste en seleccionar, de todos los posibles puntos que no hayan sido recorridos con aterioridad, el que esté a menor distancia, y tomarlo como si el resto tuvieran esa misma distancia, es decir, si hemos seleccionado ya 3 puntos y quedan 2 mas por recorrer, la función de cota interpretaría que la distancia total que nos quedaría por recorrer será dos veces, ya que sólo quedan dos puntos por seleccionar, la menor de esos dos puntos al último seleccionado. De esta forma si la solución que se propone mediante esta función de cota no es mejor que la que tengamos seleccionada en ese momento como la mejor, entonces se descartará el camino y se comenzará a explorar otro.

4.1.3 Algoritmo Backtracking

4.2 Justificación

4.3 Eficiencia teórica y empírica

4.3.1 Eficiencia teórica

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

Primero empezaremos mostrando la función de cota global:

```
1  vector<int> nearest_neighborTSP(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 función es obtenida de la práctica anterior de algoritmos Greedy. La idea es sencilla, nos dan un conjunto de puntos y 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. Este algoritmo Greedy junto con la siguiente función, que calcula la distancia total de un camino dado a partir de su matriz de adyacencia, nos permite obtener una cota superior inicial.

```

1 double calcularDistanciaTotal(const vector<vector<double>>& distancias,
2   const vector<int>& camino) {
3   double total = 0.0;
4   for (int i = 0; i < camino.size() - 1; ++i) {
5       total += distancias[camino[i]][camino[i + 1]];
6   }
7   total += distancias[camino.back()][camino.front()];
8   return total;
9 }

```

5.1.2 Cota local

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

```

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

```

```

29 double cota_inferior_2(const std::vector<std::vector<double>>& matriz,
30   const std::vector<int>& indices_a_ignorar = {}) {
31   double min_global = std::numeric_limits<double>::max();
32   int filas_contadas = 0;
33
34   for (int i = 0; i < matriz.size(); ++i) {
35       if (std::find(indices_a_ignorar.begin(), indices_a_ignorar.end(), i) != indices_a_ignorar.end()) {
36           continue;
37       }
38
39       double minimo_fila = std::numeric_limits<double>::max();
40       for (int j = 0; j < matriz[i].size(); ++j) {
41
42           if (std::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 < std::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 funci3n de cota local consiste en calcular el m3nimo de una matriz de distancias, excluyendo obviamente los valores de 0, pasandose como par3metros los 3ndices a ignorar. Destacar que adem3s de obviar dichas filas se obviar3n 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 funci3n de cota, tiene un funcionamiento distinto. Consiste principalmente en encontrar la m3nima distancia de una matriz de adyacencia y despu3s multiplicar dicho valor por las filas restantes, suponiendo que ese camino ser3 el m3nimo para todos, de ah3 que se multiplique por el n3mero de nodos restantes. Ignorando, igual que con la cota anterior, los 0 y las filas y las columnas que se pasan como par3metros a ignorar. Funci3n para calcular cota inferior calculando el m3nimo de una matriz de distancias, excluyendo el 0 y las filas. Es claro que este segundo m3todo nos proporciona una cota m3s restrictiva por lo que nos interesa m3s.

M3s 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 informaci3n sobre la cota m3nima, el camino y la distancia recorrida ser3:

```

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              ) {
8              path = inicial;
9              distancia_recorrida = dist_rec;
10             cota_inferior = cota_inf;
11         }
12     };

```

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

```

1  vector<int> branch_and_bound_greedy(vector<int>& points, vector< vector
2  <double>> &distancias, int inicial){
3      priority_queue<Nodo, vector<Nodo>, Comparador> no_visitados;
4      vector<int> mejor_camino = {inicial};
5      Nodo actual( mejor_camino, 0, cota_inferior(distancias));
6      no_visitados.push(actual);
7
8      double costo_minimo = calcularDistanciaTotal(distancias,
9          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             vector<int> faltantes= numeros_faltantes(actual.path,
18                 points.size()-1);
19             actual.distancia_recorrida += distancias[actual.path.back()
20                 ][faltantes[0]];
21             actual.distancia_recorrida += distancias[faltantes[0]][
22                 inicial];
23             actual.path.push_back(faltantes[0]);
24             actual.path.push_back(inicial);
25             if (actual.distancia_recorrida <= costo_minimo) {
26                 costo_minimo = actual.distancia_recorrida;
27                 mejor_camino = actual.path;
28             }
29         }
30         else {
31             if (actual.cota_inferior <= costo_minimo ){
32                 vector<int> faltantes = numeros_faltantes(actual.path,
33                     points.size()-1);
34                 for (int i = 0; i < faltantes.size(); ++i) {
35                     Nodo nuevo= actual;
36                     nuevo.path.push_back(faltantes[i]);
37                     nuevo.distancia_recorrida += distancias[actual.path
38                         .back()][faltantes[i]];
39                     nuevo.cota_inferior = nuevo.distancia_recorrida +
40                         cota_inferior(distancias, nuevo.path);
41                     no_visitados.push(nuevo);
42                 }
43             }
44         }
45     }
46     return mejor_camino;
47 }

```

```

34         }
35     }
36 }
37 }
38
39     return mejor_camino;
40
41 }
```

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 función 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 mínimo, siendo este la cota global.
- **Restricciones explícitas:** Que el siguiente nodo a seleccionar este dentro de los nodos faltantes
- **Restricciones implícitas:** Que el camino del nodo actual sea menor que el costo mínimo
- **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 todavía 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 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  std::vector<int> numeros_faltantes(const std::vector<int>& vec, int n)
2  {
3      std::set<int> presentes(vec.begin(), vec.end());
4
5      std::vector<int> faltantes;
6      for (int i : std::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.

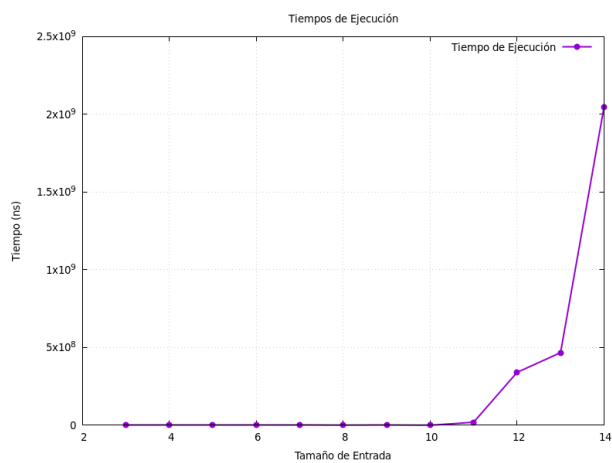


Figura 5.1: Cota 1

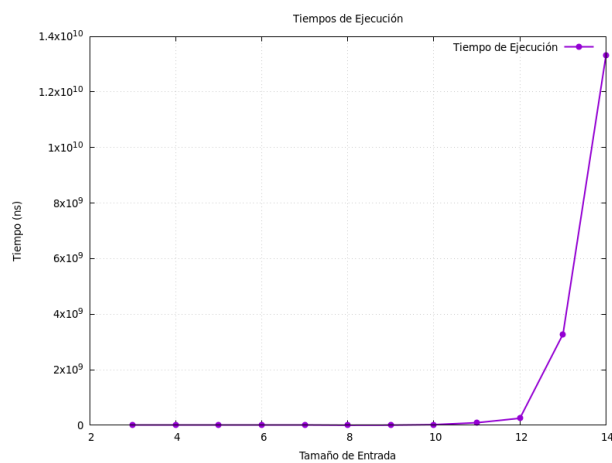


Figura 5.2: Cota 2

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.

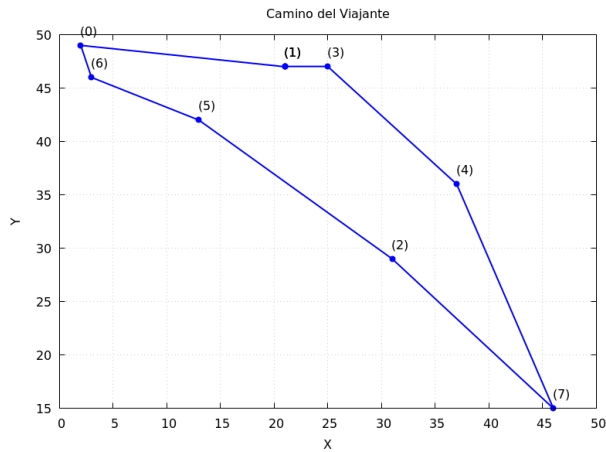


Figura 5.3: Cota 1, 8 puntos

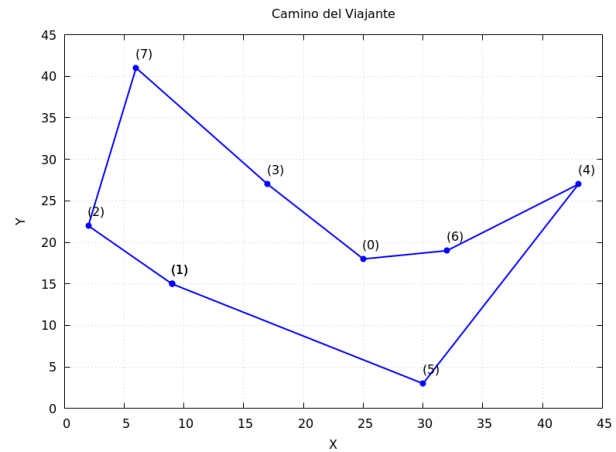


Figura 5.4: Cota 2, 8 puntos

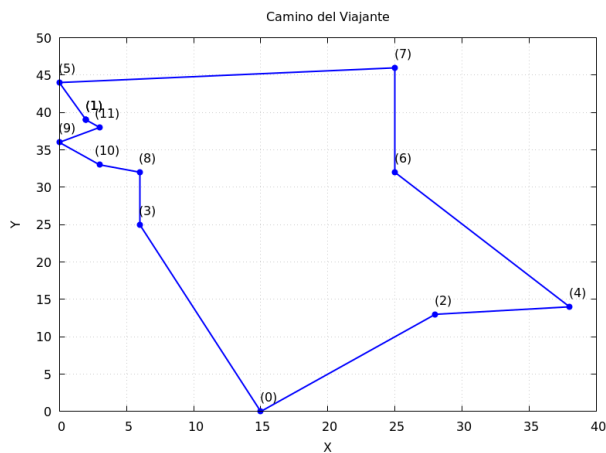


Figura 5.5: Cota 1, 12 puntos

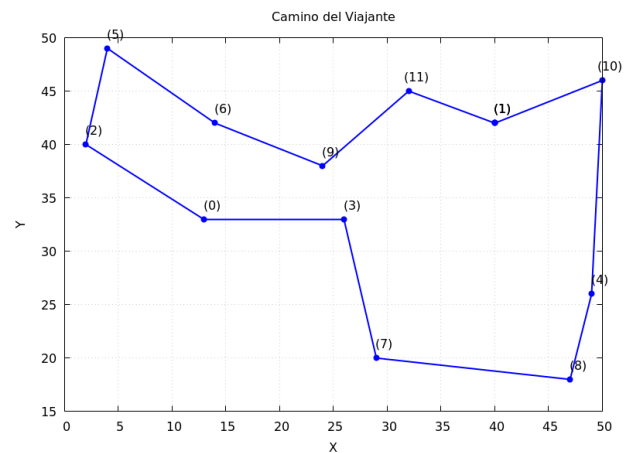


Figura 5.6: Cota 2, 12 puntos

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.