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%
  - Algoritmos Greedy del Viajante
  - Redacción memoria sección Viajante
- **Joaquín Avilés De la Fuente:** 20%
  - Programacion Dijkstra
  - Programación creación de grafos (casos de prueba)
  - Redacción memoria sección Dijkstra
  - Creación de gráficas y estudio de eficiencia Dijkstra
- **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 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

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:

```
vector<int> nearest_neighborTSP(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;
}
```

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 Greey 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.

```
double calcularDistanciaTotal(const vector<vector<double>>& distancias,
    const vector<int>& camino) {
    double total = 0.0;
    for (int i = 0; i < camino.size() - 1; ++i) {
        total += distancias[camino[i]][camino[i + 1]];
    }
    total += distancias[camino.back()][camino.front()];
    return total;
}
```

5.1.2 Cota local

Además de esta cota global, hemos implementado do 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) {

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

    return suma_minimos;
}

```

```
29 double cota_inferior_2(const std::vector<std::vector<double>>& matriz,
      const std::vector<int>& indices_a_ignorar = {}) {
30     double min_global = std::numeric_limits<double>::max();
31     int filas_contadas = 0;
32 
33     for (int i = 0; i < matriz.size(); ++i) {
34 
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 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:

```
struct Nodo {
      vector<int> path;
      double distancia_recorrida;
      double cota_inferior;

      Nodo(vector<int>& inicial, double dist_rec, double cota_inf) {
          path = inicial;
          distancia_recorrida = dist_rec;
          cota_inferior = cota_inf;
      }
};
```

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_neighborTSP(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);
```

```
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 minimo,siendo este la cota global.
- **Restricciones explícitas:**
- **Restricciones implícitas:**
- **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:**
- **Nodo muerto:**
- **e-nodo:**Nodo actual

5.2 Justificación

5.3 Eficiencia teórica y empírica

5.3.1 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 $O(log(n))$.

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

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

       std::vector<int> faltantes;
       for (int i : std::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 $O(n! \cdot n^4)$.

5.3.2 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.

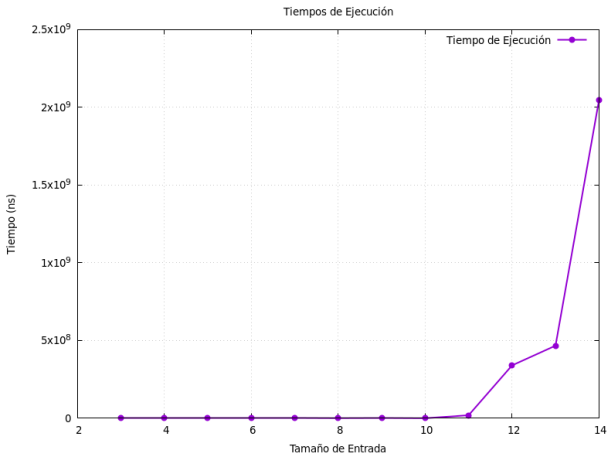

Figura 5.1: Cota 1

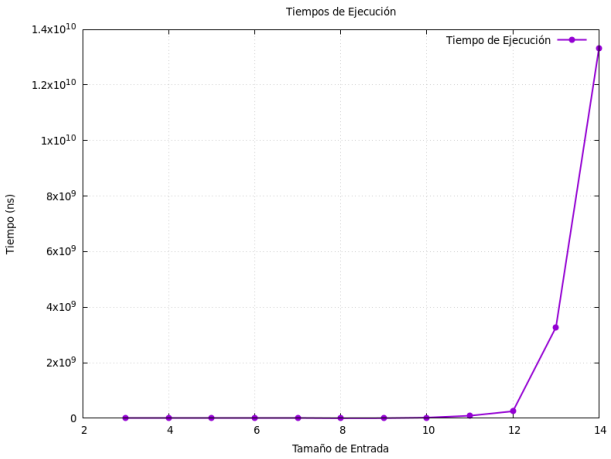

Figura 5.2: Cota 2

5.4 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.

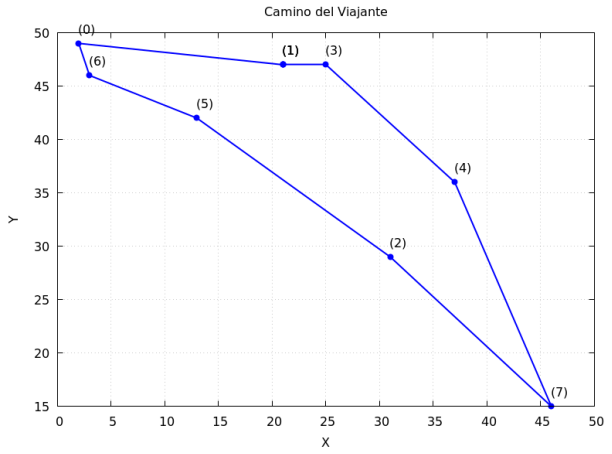

Figura 5.3: Cota 1, 8 puntos

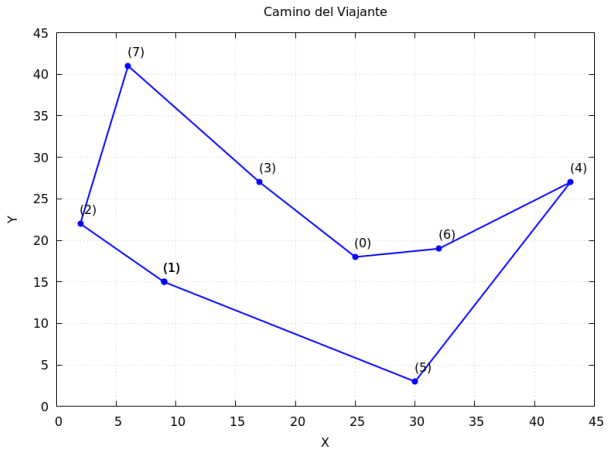

Figura 5.4: Cota 2, 8 puntos

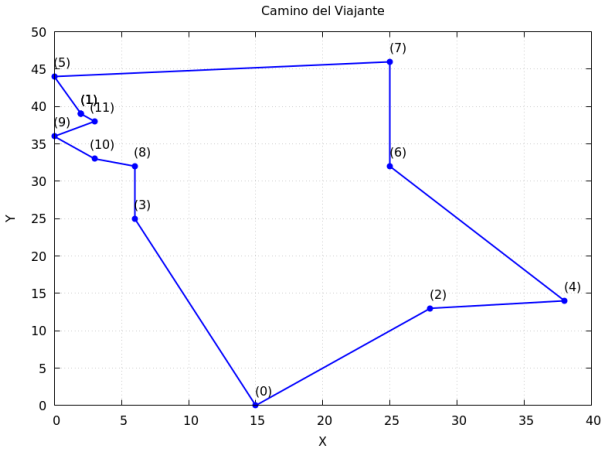

Figura 5.5: Cota 1, 12 puntos

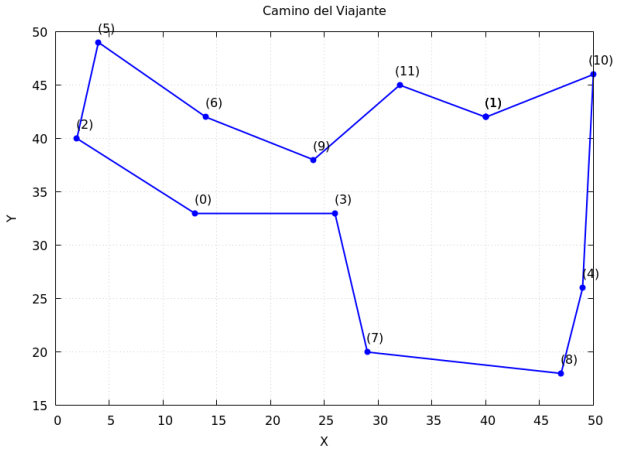

Figura 5.6: Cota 2, 12 puntos

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.