UNIVERSIDAD DE GRANADA
E.T.S. DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Asignatura: Algorítmica

Grupo nº: 4

Integrantes: Raúl Rodríguez Pérez, Francisco Javier Gallardo Molina, Inés Nieto Sánchez, Antonio Lorenzo Gavilán Chacón

Grupo: C1

Resolución Práctica 3: Algoritmos Voraces (Greedy)

Curso 2019-2020

Grado en Ingeniería Informática

Índice:

1. Introducción

Esta práctica está relacionada con los algoritmos voraces o algoritmo greedys. Dichos algoritmos se basan en realizar una estrategia de búsqueda por la cual, siguiendo una heurística consistente, se trate de elegir la mejor opción en cada paso para conseguir finalmente, llegar a la solución óptima del problema. En dicha práctica se nos ha pedido realizar la implementación de diversos algoritmos de tipo voraz, que resuelvan el famoso problema del viajante de comercio. Dichas implementaciones tendrán tres enfoques voraces diferentes: 1) Basado en cercanía, 2) Basado en inserción y 3) Un enfoque diseñado por nosotros. Implementaremos los algoritmos basados en dichos enfoques y evaluaremos su rendimiento a la hora de resolver el problema. Finalmente como última tarea se nos ha asignado implementar un algoritmo voraz que, para 'n' trabajadores y 'n' tareas, siendo $c_{i,j} > 0$ el coste de asignarle la tarea 'j' al trabajador 'i', obtengamos una asignación de tareas a trabajadores con coste mínimo.

2. Enfoque basado en cercanía

En primer lugar, hemos realizado la implementación de un algoritmo que resuelva el problema del viajante de comercio mediante un enfoque basado en cercanía. Vamos a proceder a explicar dicho problema del viajante de cara a entender este enfoque y los otros dos restantes que se explicarán a continuación. El problema del viajante de comercio se define como; dado un conjunto de ciudades y una matriz con las distancias entre todas las ciudades (destacando que la distancia entre una ciudad consigo misma no se tiene en cuenta y tiene valor -1), un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma que la distancia recorrida sea mínima.

Una vez tenemos claro el problema, vamos a explicar el primer enfoque que hemos realizado para resolverlo. El enfoque basado en cercanía consiste en que el viajante se desplace siempre a la ciudad vecina más cercana. Es decir, dada una ciudad inicial c0, se agrega como siguiente ciudad aquella que, no se haya añadido antes, y se encuentre más cercana a c0. Dicho procedimiento se repite hasta que todas las ciudades se hayan visitado.

2.1 Análisis del algoritmo y componentes voraces

Procederemos ahora a explicar paso a paso el algoritmo que implementamos para este enfoque, resaltando los componentes del método voraz diseñado y el pseudocódigo del problema.

En primer lugar vamos a ver si podemos afirmar que el problema que abordamos, se trata de un problema con enfoque greedy. Para ello analizaremos las características de un enfoque greedy:

- **Candidatos**: Las ciudades a visitar (nodos del grafo).
- **Usados**: Nodos asignadas hasta el momento.
- **Solución**: El orden en el que hay que visitar los nodos.
- **Criterio de factibilidad**: Cada vez que escojamos un candidato para incorporarlo a la solución que estemos formando se deberá cumplir que:
    - No forme un ciclo con las aristas ya escogidas (excepto para la última arista elegida, que completará el recorrido del viajante).
    - No sea la tercera arista elegida incidente en algún nodo.
- **Criterio de selección**: Dada la primera ciudad, seleccionamos la ciudad más cercana a esta.
- **Objetivo**: La suma de las distancias entre los nodos que constituyan la solución sea mínima.

Como observamos, nuestro problema presenta todas las características propias para la resolución mediante un enfoque greedy. A continuación procederemos a describir nuestra implementación del algoritmo basado en un enfoque de cercanías.

En primer lugar, leemos los datos proporcionados por el profesor y almacenados en un fichero con la función **leer_puntos(fp,m)**, donde fp es un string con el nombre del fichero donde están los datos originales guardados y m es el **map <int, pair <double, double>>** donde se almacena el número de ciudad y sus coordenadas. Tras realizar la lectura, estos datos se pasan a una matriz con la función **calcular_matriz(m,matriz)**, dicha matriz almacena la distancia entre una ciudad ‘i’ con respecto a otra ciudad ‘j’, siendo la distancia entre una ciudad y la misma igual a -1. El cálculo de dicha distancia se realiza mediante la distancia euclídea, que se calcula siguiendo esta fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

A continuación, con la función **recorrido(matriz, distancia)**, creamos y devolvemos el recorrido de ciudades que tendrá que seguir el viajero. Destacando que en dicha función, en primer lugar se elige la primera ciudad, y tras esto se repite el proceso de buscar la ciudad más cercana hasta que se hayan visitado todas y cada una de las ciudades. A su vez, en el momento que vamos eligiendo los candidatos que serán parte de nuestra solución, vamos calculando e incrementando la distancia total del recorrido y almacenando dicha información en la variable ‘distancia’, que se pasa como parámetro.

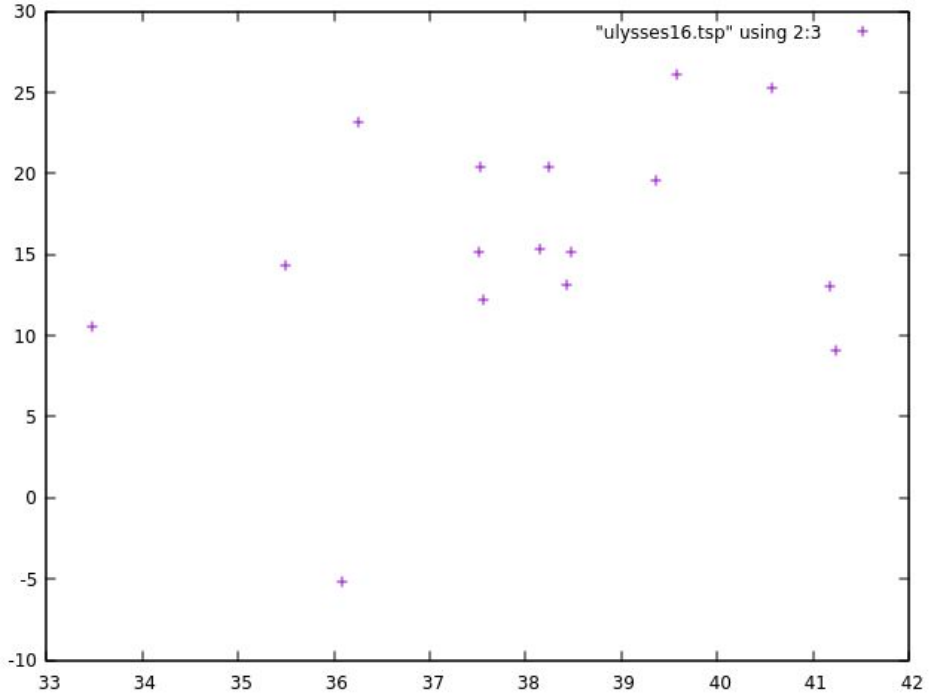
Finalmente, ya tenemos el recorrido del viajero y la distancia de este, por lo que solo quedaría mostrar el resultado.

2.2 Pseudocódigo del problema

```
Require: Conjunto de ciudades C,S
x=0
S=C[0]
for  i = 1 to len(C) do
        if C[i] MenorDistancia then
                x = C[i]
                S.add(x)
                C[i].erase
        end if
end for
        return S
```

2.3 Ejemplo de ejecución

Hemos realizado una ejecución del algoritmo para plasmar los resultados tras haber explicado su funcionamiento. Se ha procedido a ejecutar el algoritmo con el conjunto de datos suministrados en el archivo **ulysses16.tsp**. Dicho archivo posee 16 ciudades junto con sus coordenadas, dichas ciudades quedan plasmadas en la siguiente gráfica:



Como hemos comentado en la explicación del código, primero se crea la matriz de distancias de las ciudades (matriz de 16x16) con la función **calcular_matriz(m,matriz)**, cuyo resultado se puede ver en la siguiente imagen:

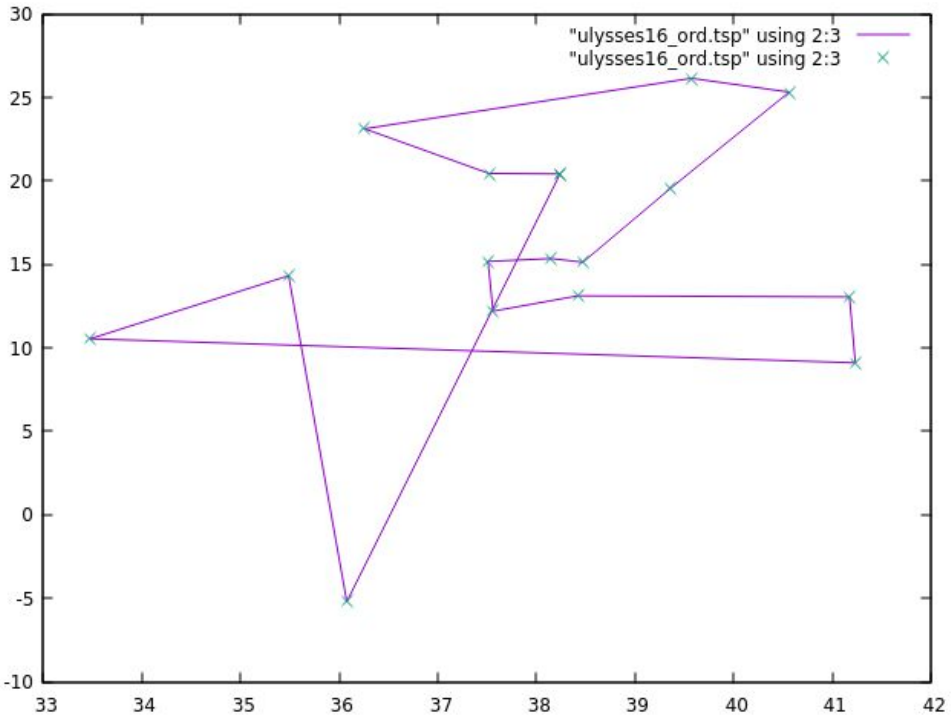
```
La matriz es:
 -1     5     5     3     10     8     7     0     11     7     25     5     5     5     6     1
5     -1     1     4     16     14     13     6     17     13     31     11     10     11     12     6
5     1     -1     4     16     13     12     5     16     12     30     10     10     10     12     5
3     4     4     -1     12     11     10     2     14     11     28     8     7     8     8     4
10     16     16     12     -1     4     5     10     7     8     15     6     6     6     4     10
8     14     13     11     4     -1     1     8     4     3     17     3     3     2     2     7
7     13     12     10     5     1     -1     7     4     2     18     2     2     2     3     6
0     6     5     2     10     8     7     -1     11     8     25     5     5     5     6     2
11     17     16     14     7     4     4     11     -1     3     15     6     6     7     7     10
7     13     12     11     8     3     2     8     3     -1     18     3     3     4     5     6
25     31     30     28     15     17     18     25     15     18     -1     20     20     20     19     24
5     11     10     8     6     3     2     5     6     3     20     -1     0     0     3     4
5     10     10     7     6     3     2     5     6     3     20     0     -1     0     2     4
5     11     10     8     6     2     2     5     7     4     20     0     0     -1     2     4
6     12     12     8     4     2     3     6     7     5     19     3     2     2     -1     6
1     6     5     4     10     7     6     2     10     6     24     4     4     4     6     -1
```

A continuación, tal y como hemos explicado procederemos a seleccionar la primera ciudad y repetir el proceso para encontrar la ciudad vecina hasta que hayamos visitado todas las ciudades:

```
La primera ciudad es: 1 38.24 20.42
La ciudad vecina de la ciudad 1 es: 8 37.52 20.44
```

Al terminar el proceso anterior ya tenemos la solución a nuestro problema, es decir, el recorrido final junto con la distancia total de dicho recorrido. Vamos a representar dicho recorrido en un gráfico, quedando como resultado:

```
Recorrido final:
1 38.24 20.42
8 37.52 20.44
4 36.26 23.12
2 39.57 26.15
3 40.56 25.32
16 39.36 19.56
12 38.47 15.13
13 38.15 15.35
14 37.51 15.17
6 37.56 12.19
7 38.42 13.11
10 41.17 13.05
9 41.23 9.1
5 33.48 10.54
15 35.49 14.32
11 36.08 -5.21
1 38.24 20.42
```



3. Enfoque basado en inserción

En esta ocasión vamos a implementar un algoritmo voraz para la resolución del problema del viajante de comercio, pero esta vez con un enfoque basado en inserción. La idea de este enfoque reside en comenzar con un recorrido parcial, que incluya algunas de las ciudades, y luego extender este recorrido insertando las ciudades restantes mediante algún criterio de tipo voraz. Para esta ocasión, el recorrido parcial se calculará a partir de las tres ciudades que formen un triángulo lo más grande posible, es decir la ciudad más al norte, sur y oeste, por ejemplo. Una vez calculado el recorrido parcial, se procederá a extender el recorrido con el siguiente criterio: se selecciona una ciudad del conjunto que no se ha visitado, dicha ciudad se ubica en el punto del circuito que provoque menor incremento de su longitud total, es decir, para cada ciudad no visitada se busca la posición que resulta en menor incremento de la longitud del circuito, y se selecciona la ciudad (y posición) que genera el menor incremento en cada paso. Por tanto, seleccionaremos aquella ciudad que nos proporcione el mínimo de los mínimos calculados para cada una de las ciudades.

3.1 Análisis del algoritmo y componentes voraces

Procederemos ahora a explicar paso a paso el algoritmo que implementamos para este enfoque, resaltando los componentes del método voraz diseñado y el pseudocódigo del problema.

En primer lugar vamos a ver si podemos afirmar que el problema que abordamos, se trata de un problema con enfoque greedy. Para ello analizaremos las características de un enfoque greedy:

- **Candidatos**: Las ciudades a visitar (nodos del grafo).
- **Usados**: Nodos asignadas hasta el momento.
- **Solución**: El orden en el que hay que visitar los nodos.
- **Criterio de factibilidad**: Cada vez que escojamos un candidato para incorporarlo a la solución que estemos formando se deberá cumplir que:
    - No forme un ciclo con las aristas ya escogidas (excepto para la última arista elegida, que completará el recorrido del viajante).
    - No sea la tercera arista elegida incidente en algún nodo.
- **Criterio de selección**: De entre todas las ciudades no visitadas, elegimos aquella que provoque el menor incremento en la longitud total del circuito
- **Objetivo**: La suma de las distancias entre los nodos que constituyan la solución sea mínima.

Como observamos, nuestro problema presenta todas las características propias para la resolución mediante un enfoque greedy. A continuación procederemos a describir nuestra implementación del algoritmo basado en un enfoque de inserción.

En primer lugar, leemos los datos proporcionados por el profesor y almacenados en un fichero con la función **leer_puntos(fp,m)**, donde fp es un string con el nombre del fichero donde están los datos originales guardados y m es el **map <int, pair <double, double>>** donde se almacena el número de ciudad y sus coordenadas. Tras realizar la lectura, estos datos se pasan a una matriz con la función **calcular_matriz(m,matriz)**, dicha matriz almacena las distancias entre las matrices.

A continuación, con la función **recorrido(matriz, m, distancia)**, donde ‘matriz’ es la matriz de distancias, ‘m’ el mapa con las ciudades y sus coordenadas y ‘distancia’ una variable donde almacenaremos la distancia total del recorrido. En primer lugar calculamos la ciudad más al norte, al sur y al oeste y las añadimos a nuestra solución. A continuación, mientras queden ciudades por visitar vamos a repetir el proceso de elegir el candidato apropiado para añadir a nuestra solución. Este proceso se basa en que, mientras queden ciudades por visitar, elegiremos la próxima ciudad que añadir con la función **elegir_ciudad(result, final).** En dicha función se va comprobando el aumento de distancia que supone insertar una ciudad de entre todas las ciudades que quedan por visitar, para cada par de ciudades del recorrido inicial (en el primer caso se compara con los pares que se pueden formar con las 3 ciudades añadidas por el momento). Una vez tengamos los valores de esas distancias (en el caso de tener solo 3 ciudades en el recorrido final tendríamos 3 valores de distancias diferentes), se calcula la distancia mínima entre dichos

valores y se guarda la posición en la que se alcanza este mínimo. Finalmente dicho proceso se repite para cada una de las ciudades que quedan por visitar, hasta que, para cada una de ellas tengamos un valor de distancia min. Finalmente calculamos el mínimo de todos esos valores mínimos, obteniendo así la ciudad y su posición en la lista de no visitadas.

Tras terminar el proceso anterior, tendremos la solución al problema, es decir, el recorrido de ciudades junto con su distancia total.

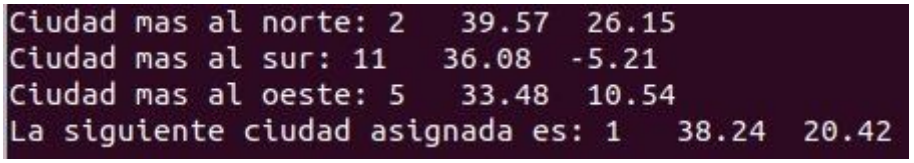
3.2 Pseudocódigo del problema

```
Require: Conjunto de ciudades C,S, distancia
x=0; S=CalcularRecorridoInicial();          // cálculo del triángulo
                                               de ciudades
while len(C)>0 do
        pair<int, list<int>::iterator> p = elegir_ciudad();
        insertar_ciudad(S,p.first, p.second);                  // Insertar
                                                    cuidad no visitada
end while
distancia = distancia_total()                  //Calcular la distancia
                                                    total del recorrido
        return S;
```

3.3 Ejemplo de ejecución

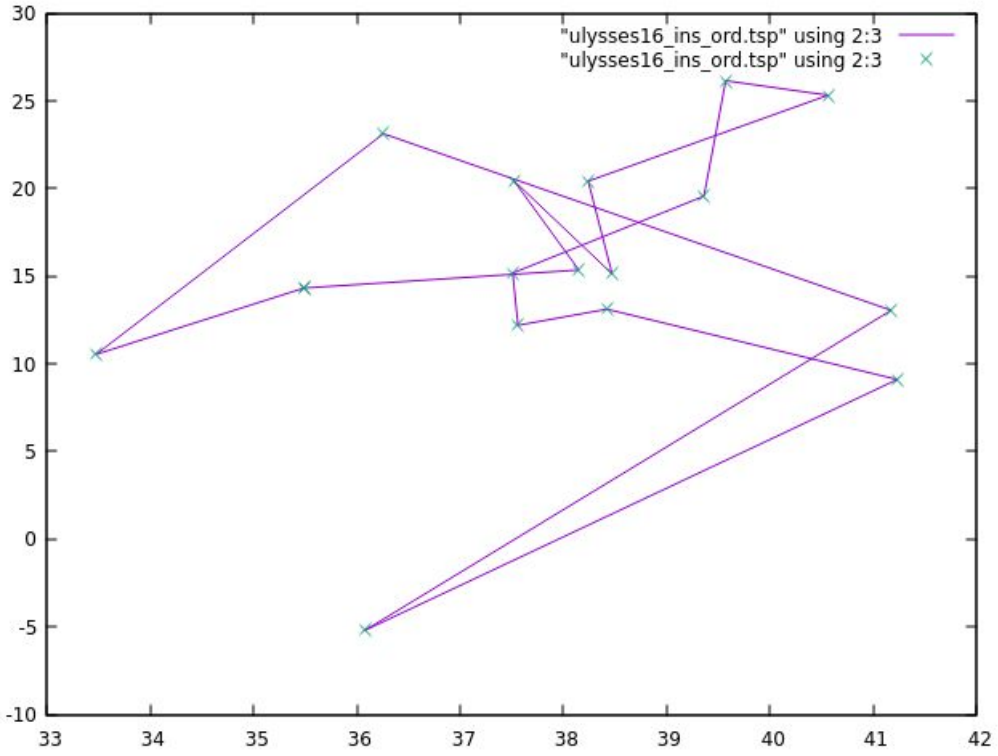
Hemos realizado una ejecución del algoritmo para plasmar los resultados tras haber explicado su funcionamiento. Como comentamos en el anterior caso de ejecución emplearemos el mismo conjunto de datos, en este caso el archivo **ulysses16.tsp**, para realizar los ejemplos de ejecución en los 3 enfoques del problema. La gráfica y la matriz de distancias de este archivo se encuentra en el anterior ejemplo, y por mayor comodidad y no repetir las imágenes las obviaremos en este y en el próximo caso.

A continuación de haber realizados los primeros pasos (lectura y creación de la matriz de distancias), tal y como hemos explicamos, calculamos y añadimos a la solución las ciudades más al norte, sur y oeste. Una vez añadidas, comenzaremos a buscar la próxima ciudad candidata siguiendo el criterio de que provoque el mínimo aumento de la distancia total del circuito:



Finalmente, una vez acabado el proceso anterior, poseemos la solución al problema, el recorrido final y la distancia total de este. Representamos dicho recorrido en la siguiente gráfica:

```
El recorrido final es:
15 35.49 14.32
13 38.15 15.35
8 37.52 20.44
12 38.47 15.13
1 38.24 20.42
3 40.56 25.32
2 39.57 26.15
16 39.36 19.56
14 37.51 15.17
6 37.56 12.19
7 38.42 13.11
9 41.23 9.1
11 36.08 -5.21
10 41.17 13.05
4 36.26 23.12
5 33.48 10.54
15 35.49 14.32
```



4. Enfoque creado por nosotros

La filosofía del algoritmo que hemos desarrollado es la siguiente. Se crea un multimap con el conjunto de todas las aristas del grafo, compuesto cada elemento por la longitud de la arista y las dos ciudades que son unidas por esa arista. Estos

elementos estarán ordenados de menor a mayor longitud de arista y en ese orden se van añadiendo a una lista de pares de ciudades, siempre y cuando sea factible hacerlo, es decir cuando no sea la tercera arista elegida incidente en algún nodo o no forme un ciclo con las aristas ya escogidas. Por último se introduce en la lista el último par de ciudades que tengan una sola arista (es decir las ciudades origen y final) y de esta lista se obtienen las ciudades ordenadas formando el camino mínimo.

4.1 Análisis del algoritmo y componentes voraces

Veamos si se cumplen las 6 características para poder afirmar de que se trata de un problema con enfoque greedy.

- **Candidatos**: Las aristas del grafo.
- **Usados**: Aristas asignadas hasta el momento.
- **Solución**: El orden en el que hay que visitar los nodos.
- **Criterio de factibilidad**: Cada vez que escojamos un candidato para incorporarlo a la solución que estemos formando se deberá cumplir que:
    - No forme un ciclo con las aristas ya escogidas (excepto para la última arista elegida, que completará el recorrido del viajante).
    - No sea la tercera arista elegida incidente en algún nodo.
- **Criterio de selección**: Vamos seleccionando las aristas en orden de menor a mayor longitud.
- **Objetivo**: La suma de las longitudes de las aristas que constituyan la solución sea mínima.

El problema del viajante de comercio presenta las 6 características que buscábamos, por lo tanto es resoluble según un enfoque Greedy. Ahora pasaremos a explicar el código implementado:

En primer lugar se leen los datos existentes en un fichero con la función **leer_puntos(fp,m)**, donde fp es un string con el nombre del fichero donde están los datos originales guardados y m es el **map <int, pair <double, double>>** donde se almacena el número de ciudad y sus coordenadas. Estos datos se pasan a una matriz con la función **calcular_matriz(m,matriz)**, en el que vienen reflejados las distancias entre las ciudades. Este procedimiento inicial es el mismo para todas las versiones realizadas del problema del viajante de comercio.

Seguidamente se calculan todas las aristas del grafo y se guardan de menor a mayor longitud en un **multimap <int , pair <int,int> > aristas**, donde el primer elemento es la longitud de la arista y el segundo la posición del par de ciudades unidas por esta arista. Este procedimiento se hace con la función **calcular_aristas(matriz)**, cuyo código se puede ver abajo.

```
149  // Calcula las posibles aristas y sus distancias
150  multimap<int , pair<int,int> > calcular_aristas(const vector< vector<int> > & m) {
151      multimap<int , pair<int,int> > a;
152      for(int i=0; i<m.size(); i++) {
153          for(int j=0; j<m[i].size(); j++) {
154              if(j>i) {
155                  a.insert(pair<int, pair<int,int> >(m[i][j],pair<int,int>(i+1,j+1)));
156              }
157          }
158      }
159      return a;
160  }
```

Por último con la función **recorrido(aristas, distancia)** se va creando una lista de pares de ciudades, obtenidas desde el primer elemento del map anterior, siempre y cuando sea factible hacerlo (con la función **factible()**), es decir cuando no sea la tercera arista elegida incidente en algún nodo o no forme un ciclo con las aristas ya escogidas (función **hayciclos()**) (ver imagen de abajo).

```
262  //calcula el recorrido
263  vector<int> recorrido(multimap<int, pair<int,int> > & a,int & d) {
264      vector<int>c;
265      list<pair<int,int> > aux;
266      multimap<int , pair<int,int> >::iterator it=a.begin();
267      aux.push_back((*it).second);
268      d+=(*it).first;
269      it++;
270      while(it!=a.end()) {
271          if(factible(aux,(*it).second) and !hayciclos(aux,(*it).second)) {
272              aux.push_back((*it).second);
273              d+=(*it).first;
274          }
275          it++;
276      }
277      cerrar_ciclo(aux,a,d);
278      c=camino(aux);
279      return c;
280  }
```

Dentro de esta función, al final con la función **cerrar_ciclo(aux,a,d)** se introduce en la lista el último par de ciudades que tengan una sola arista (es decir las ciudades origen y final) y de esta lista se obtienen las ciudades ordenadas formando el camino mínimo, con la función **camino(aux)**, que es la solución al problema.

4.2 Pseudocódigo del problema

VIAJANTE_DISTANCIAS.CPP

```
Require: Archivo.tsp con datos de ciudades y coordenadas
    aristas (distancia, <ciudad1, ciudad2>);
    m (ciudad, coordenadas);
    m ⟵ Leer ciudades y coordenadas;
    matriz ⟵ Distancia entre ciudades de m;
    aristas ⟵ Aristas entre ciudades de matriz;
    ciudades ⟵ recorrido sobre aristas;
    return ciudades
```

```
Función recorrido (aristas, distancia)
    C = Ø;
    it ⟵ Iterador sobre aristas;
    aux.add (ciudad1, ciudad2);
    distancia ⟵ Distancia entre ciudad1 y ciudad2;
    it ++;
    while it distinto de it.end do
        if es factible (aux, (*it).second) and no hay ciclos (aux, (*it).second) then
            aux.add ((*it).second));
            distancia.add ((*it).first));
        end if
    end while
    aux ⟵ Añadir las dos ciudades con una arista para cerrar el ciclo;
    actualizar distancia;
    C ⟵ Calcular camino sobre aux;
    return C;
```

4.3 Ejemplo de ejecución

Procedemos a la ejecución nuevamente de algoritmo implementado con el conjunto de datos suministrado en el archivo **ulysses16.tsp**, el cual tiene 16 ciudades junto con sus coordenadas. La gráfica y la matriz de distancias de dicho archivo, además de su obtención se ha explicado en los anteriores ejemplos de ejecución por lo tanto no lo repetiremos en este caso.

Tras hallar la matriz de distancias, se calculan todas las aristas del grafo y se guardan de menor a mayor longitud, lo cual ya hemos explicado anteriormente que se hace con la función **calcular_aristas(matriz)**.

Por último con la función recorrido(aristas, distancia) se va creando una lista de pares de ciudades, que se van añadiendo, siempre y cuando sea factible hacerlo (con la función **factible()**), es decir cuando no sea la tercera arista elegida incidente en algún nodo o no forme un ciclo con las aristas ya escogidas (función **hayciclos()**). Así por ejemplo con las ciudades 12 y 13 se forma un ciclo al añadirlas a esta lista de ciudades, como se ve en la siguiente imagen, por lo que no se añaden.

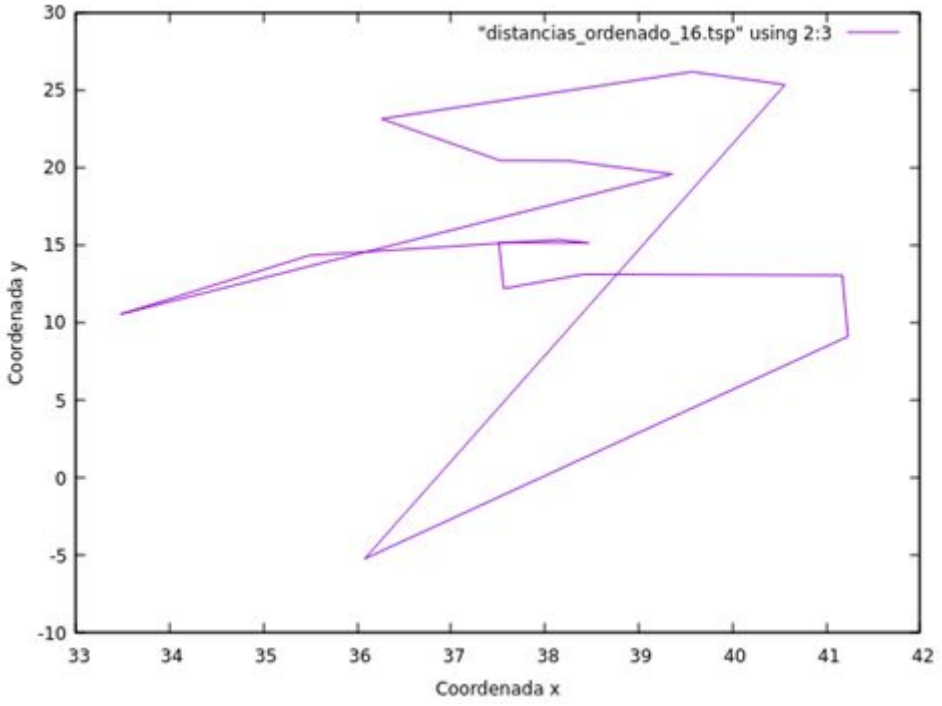
```
La ciudad 12 y la ciudad 13 tienen ciclos
```

Y por último se forma el vector de ciudades una vez cerrado el ciclo con el par de ciudades con una sola arista, gracias a la función **cerrar_ciclo(aux,a,d)** y así se obtiene las ciudades colocadas en orden preparadas para ser unidas consecutivamente en el grafo. Estas ciudades y sus coordenadas se ven en la imagen siguiente.

```
Después de unir el ciclo el resultado final es:

1       38.24   20.42
8       37.52   20.44
4       36.26   23.12
2       39.57   26.15
3       40.56   25.32
11      36.08   -5.21
9       41.23   9.1
10      41.17   13.05
7       38.42   13.11
6       37.56   12.19
14      37.51   15.17
12      38.47   15.13
13      38.15   15.35
15      35.49   14.32
5       33.48   10.54
16      39.36   19.56
1       38.24   20.42
```

De esta forma el grafo obtenido con el recorrido final es el de la siguiente imagen.



5. Trabajadores y tareas

Supongamos que disponemos de n trabajadores y n tareas. Sea bij > 0 el coste de asignarle la tarea j al trabajador i.

Una asignación de tareas puede ser expresada como la siguiente ecuación:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} b_{ij}$$

con una asignación de los valores 0 ó 1 a las variables xij,

xij = 0 significa que al trabajador i no le han asignado la tarea j,

xij = 1 indica que al trabajador i sí le han asignado la tarea j.

Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador. Se debe procurar que la asignación de tareas sea óptima, es decir que la suma de los costes sea mínimo.

5.1 ¿Es greedy nuestro problema?

Veamos si se cumplen las 6 características para poder afirmar de que se trata de un problema con enfoque greedy.

- **Candidatos**: Los costes de la asignación de las tareas a los trabajadores, que son los elementos de la matriz de costos.

- **Usados**: Pares de elementos trabajador tarea, asociados a los costos asignados hasta el momento.
- **Solución**: Conjunto de asignaciones tal que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador.
- **Criterio de factibilidad**: El trabajador asignado no debe haber sido asignado con anterioridad. Y la tarea asignada no debe haber sido seleccionada anteriormente para ningún otro trabajador.
- **Criterio de selección**: Asignar a cada trabajador la mejor tarea posible.
- **Objetivo**: Se debe procurar que la asignación de tareas sea óptima, es decir que la suma de los costes sea mínimo.

Como se puede observar el problema es Greedy. Sin embargo este enfoque no siempre da resultados óptimos. Como ejemplo tenemos esta matriz de costos:

		1	2	3
	1	16	20	18
Trabajador	2	11	15	17
	3	17	1	20

Para este caso, el algoritmo produce una matriz de asignaciones en donde los “unos” están en las posiciones (1,1), (2,2) y (3,3), esto es, asigna la tarea i al trabajador i (i = 1, 2 ,3), con un valor de la asignación de 51 (= 16 + 15 + 20).

Sin embargo la asignación óptima se consigue con los “unos” en posiciones (1,3), (2,1) y (3,2), esto es, asigna la tarea 3 al trabajador 1, la 1 al trabajador 2 y la tarea 2 al trabajador 3, con un valor de la asignación de 30 (= 18 + 11 + 1).

Por lo tanto no vamos a realizar un enfoque Greedy a este problema, sino que lo resolveremos con un método que explicaré a continuación en el siguiente punto.

5.2 Descripción de la solución propuesta

Existe un algoritmo exacto que resuelve este problema llamado **Algoritmo Húngaro**, pero en este caso se utilizará una variante cuyos pasos defino a continuación. En primer lugar se crea la matriz “m” de costes que será cuadrada ya que suponemos que el número de trabajadores es el mismo que el de tareas. Esta matriz se rellena con números aleatoriamente entre el 1 y 50 utilizando para ello la función “**crear(m)**” (Ver imagen de abajo).

```
53  //creamos la matriz de costes de forma aleatoria
54  void crear ( vector< vector<int> > & m) {
55      srand(time(NULL));
56      for(int i=0; i<m.size(); i++) {
57          for(int j=0; j<m[i].size(); j++)
58              m[i][j]=(rand()%50)+1;//aleatorio entre 1-50
59      }
60  }
```

Seguidamente con la función "**seleccionar(m)**" se entra a ejecutar un ciclo en el que se recorre la matriz desde el elemento m[0][0] hasta el último y se selecciona aquél elemento con menor coste (Ver imagen de abajo).

```
74  vector <pair<int,int> > seleccionar(vector< vector<int> > & m) {
75      vector <pair<int,int> > s;
76      pair<int,int> obj;
77      int aux;
78      for(int contador=0; contador<m[0].size(); contador++) {
79          obj.first=0;
80          obj.second=0;
81          aux=m[0][0];
82          for(int i=0; i<m.size(); i++) {
83              for(int j=0; j<m[i].size(); j++) {
84                  if( m[i][j]!=999 && m[i][j]<aux) {
85                      aux=m[i][j];
86                      obj.first=i;
87                      obj.second=j;
88                  }
89              }
90          }
91          s.push_back(obj);
92          modificarfilcol(obj.first,obj.second,m);
93      }
94      return s;
95  }
```

El siguiente paso es añadir a un vector de pares el número de fila y columna que se corresponde con el elemento anterior y, por medio de la función "**modificarfilcol(fila,columna,matriz)**", asignar a los elementos de dicha fila y columna de la matriz de costes el valor 999, que es mayor que cualquier otro valor de la matriz. Con esto se indica que esta fila y columna ya no será utilizada para seleccionar el siguiente valor de coste, lo que nos asegura que ni los trabajadores ni las tareas estarán repetidos en la solución final.

```
62  //ponemos el valor de la fila y la columna a infinito
63  void modificarfilcol(int f,int c,vector< vector<int> > & m) {
64      for(int i=0; i<m.size(); i++) {
65          for(int j=0; j<m[i].size(); j++)
66              if(i==f or j==c) m[i][j]=999;
67      }
68  }
```

Este ciclo se repite tantas veces como trabajadores existan y en cada iteración se selecciona un valor de fila y columna cuya intersección corresponda con el valor mínimo de coste que encuentre en dicho ciclo y que se va añadiendo al vector de pares que será la solución al problema.

5.3 Pseudocódigo del problema.

ASIGNACIÓN_TAREAS.CPP

```
Require: Matriz de costos m
  for cont = 0 to size(m) – 1 do
    aux = m[0][0]; a = 0; b = 0; s = (0,0);
    for i = 0 to size(m) – 1 do
      for j = 0 to size(m) – 1 do
        if m[i][j] distinto a 999 and m[i][j] menor que aux then
          aux = m[i][j]; a = i; b = j;
        end if
      end for
    end for
    s.add(a,b);
    fila a – ésima de m <—— 999;
    columna b – ésima de m <—— 999;
  end for
  rerturn s
```

5.4 Ejemplo de ejecución

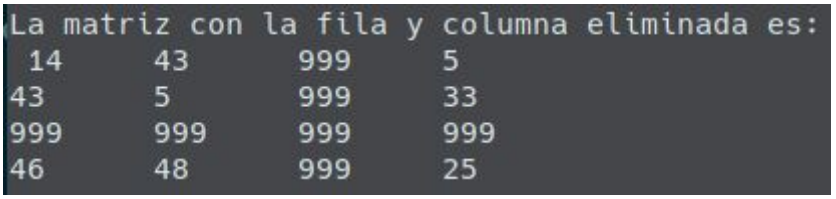
Así por ejemplo, para resolver el problema para 4 trabajadores y 4 tareas se origina una matriz de costos de 4x4 que la función “**crear(m)**” crea, como por ejemplo la de la figura de abajo:

```
 14      43      28      5
43       5       34      33
20       42      2       49
46       48      11      25
```

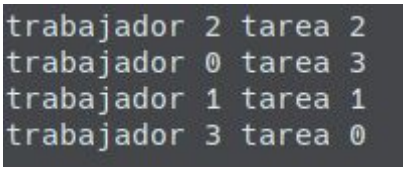
En el primer ciclo se asigna el trabajador 2 con la tarea 2, como se puede ver en la siguiente imagen:

```
El trabajador 2 tiene asignado el trabajo 2
```

De esta forma se debe eliminar la fila 2 y la columna 2 añadiendo el valor 999 a cada uno de sus posiciones:



El valor de la fila y la columna seleccionada se añade al vector de pares “s” y se repetirá el ciclo hasta terminar con las 4 iteraciones, devolviendo el vector “s” como resultado final (ver imagen de abajo).



Por último el coste de la asignación es de 2 + 5 + 5 + 46 = 58. Esta solución **no es óptima** ya que existe otra combinación entre trabajadores y tareas que minimiza aún más el coste total, la cual es la siguiente:

trabajador 2 tarea 0
trabajador 0 tarea 3
trabajador 1 tarea 1
trabajador 3 tarea 2

Esta asignación tiene un coste total de 5 + 5 + 20 + 11 = 41, que es inferior a la anterior calculada por nuestro algoritmo.

6. Conclusión

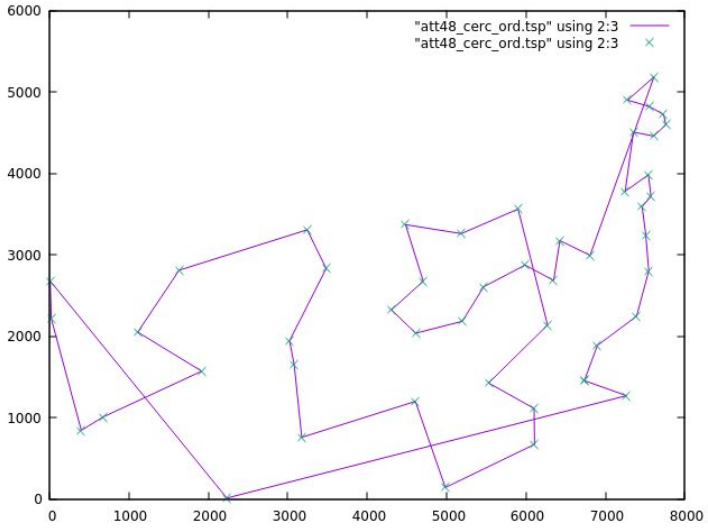
A modo de conclusión, queremos reflejar los rendimientos de los distintos enfoques que hemos utilizado para resolver el problema del viajante de comercio. Para ello hemos realizado una pequeña tabla donde se muestra la distancia total para los recorridos con los distintos enfoques y con los diferentes datos proporcionados por el profesor. Si nos fijamos, nos damos cuenta de que para casi todos los datos, el enfoque creado por nosotros, el de distancias, es el que mejor resultados da. Seguido de este, se encuentra el enfoque de cercanías (destacando que dicho enfoque para el archivo de datos de ulysses 22, da el mejor resultado entre todos) y finalmente se encuentra el de inserción, dando los peores valores para todos los datos.

El resultado más óptimo de las distancias obtenidas se corresponde con los algoritmos más sencillos y simples de implementar, que coincide con el de cercanías y el de distancias. Sin embargo, a pesar de la mayor complejidad del algoritmo de inserción no se refleja como el mejor de todos, sino que queda relegado a la última posición, lo que nos hace pensar que que este factor no es determinante a la hora de diferenciar unos enfoques de otros.

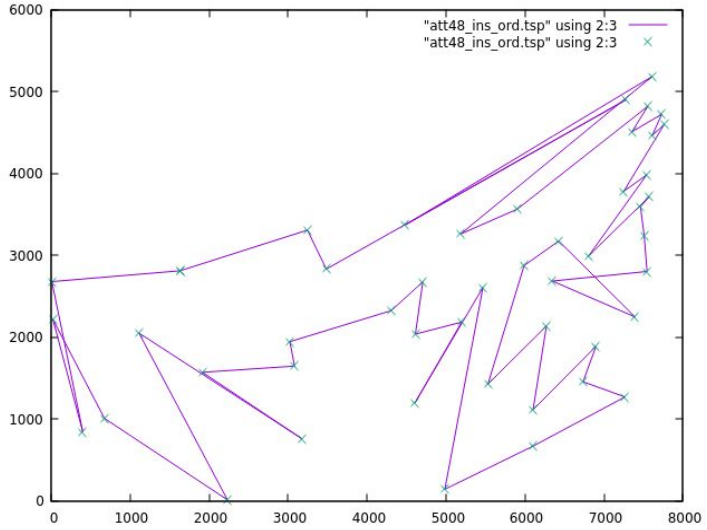
	ulysses 16	**ulysses 22**	**att48**	**a280**
cercanías	79	76	40503	3203
inserción	100	115	61949	6525
distancias	77	79	40138	3096

Queremos destacar que elegimos utilizar el conjunto de datos ulysses16 para explicar los casos de ejecución en los 3 tipos de enfoques, debido a que nos facilitaba la tarea de poder mostrar la matriz de distancia y también para que la primera comparación entre los tres enfoques fuera más homogénea. Aun así, nuestro grupo ha realizado los casos de ejecución para cada uno de los datos proporcionados y sus respectivos algoritmos. Por esto, finalmente hemos querido resaltar el hecho de que el recorrido de los grafos en la solución final refleja los datos obtenidos de las distancias para los distintos algoritmos, en los que se pueden ver claramente la diferencia de complejidades entre cada algoritmo. Por lo cual, como punto final al trabajo queremos mostrar los gráficos de los recorridos finales del archivo att48.tsp a modo de una comparación visual entre los diferentes algoritmos:

a. **Enfoque cercanía:**



b. **Enfoque inserción**



c. **Enfoque distancias**