ALGORÍTMICA

Práctica 3: Algoritmos voraces

Grupo Petazetas: Sara Bellarabi El Fazazi, Manuel Villatoro Guevara , Arturo Cortés Sánchez, Sergio Vargas Martin

Índice:

1.1 Basado en cercanía:

El esquema del problema corresponde al esquema típico de un algoritmo voraz y que por tanto presenta las 5 condiciones siguientes:

- Podemos considerar al conjunto de nodos (o al conjunto de aristas) como una lista de candidatos. Las ciudades a visitar corresponden con los nodos y el camino entre ciudad y ciudad con las aristas.

- Una solución al problema sería el orden en el que hay que visitar cada nodo o en el que se debe recorrer las aristas.
- Una función de factibilidad en la que cada vez que se escoja a un candidato para incluirlo en la solución se deban cumplir los siguientes criterios:
  - Que no forme un ciclo con las aristas o los nodos ya escogidos excepto para completar el recorrido del viajante.
  -Que no se recorran dos veces la misma arista.

- La función de selección será la que hará que obtengamos distintas soluciones para el mismo problema. En este caso la estrategia está basada en cercanía donde dada una ciudad inicial vo, se agrega como ciudad siguiente aquella vi (no incluída en el circuito) que se encuentre más cercana a vo. Esto se repite hasta que todas las ciudades se hayan visitado.
- Por último, como función objetivo tenemos que la suma de las longitudes de las aristas que constituyan la solución sea mínima.

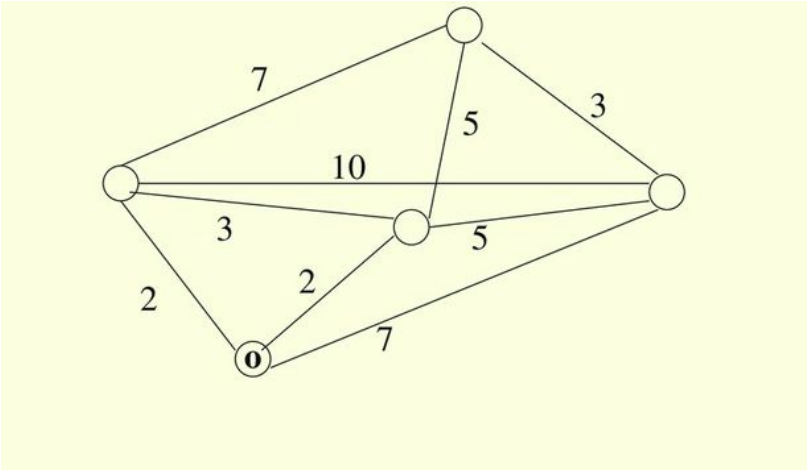
1.1.2 Pseudocódigo:

```
Ciudad { numero_ciudad, x, y }

Ciudades = leo_ciudades(archivo.txt) //Leo una lista de ciudades de un archivo.
Cojo la primera ciudad
Mientras me queden ciudades{
   Elijo la ciudad más cercana
   Introduzco en mi lista de soluciones
   Borro dicha ciudad de mis ciudades
   }
```

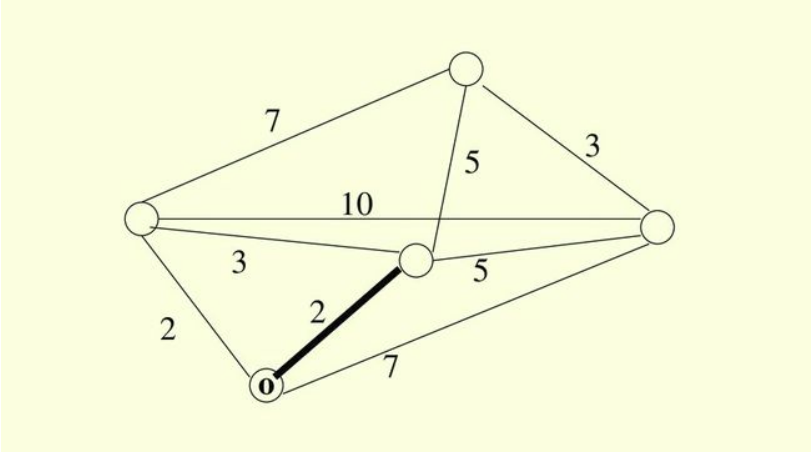
1.1.3 Eficiencia teórica:

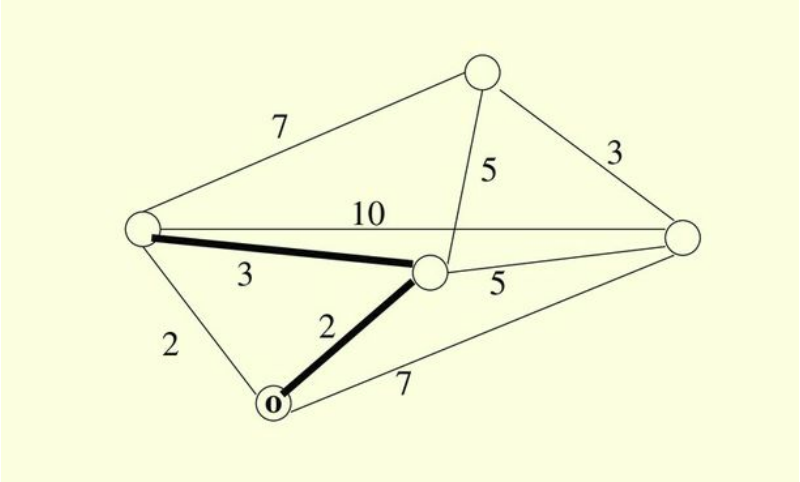
El algoritmo recorre la lista de ciudades desde 0 hasta n. La función sort ordena las ciudades según el criterio que se pase por argumento. Luego la eficiencia del algoritmo es lineal : O(n).

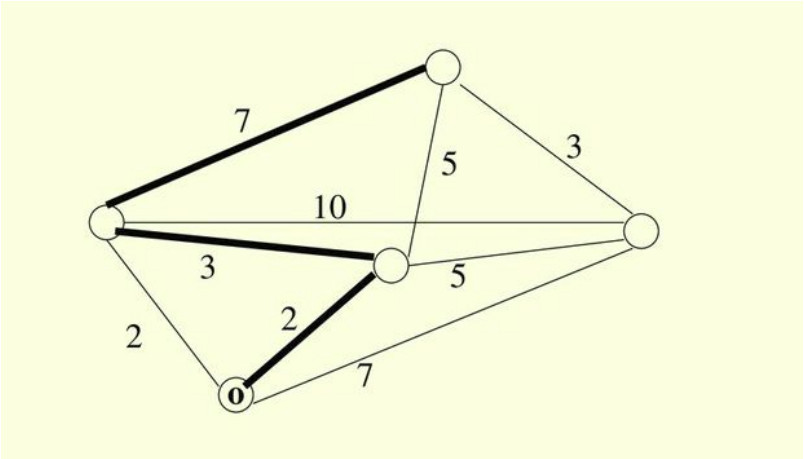
1.1.4 Escenario de ejecución:

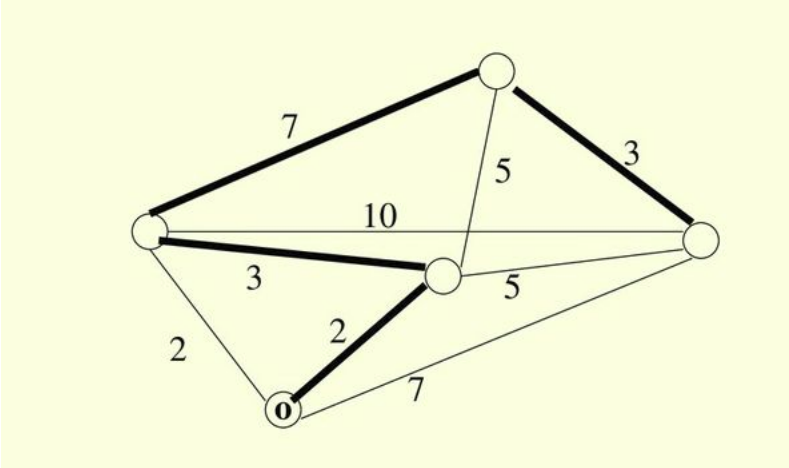


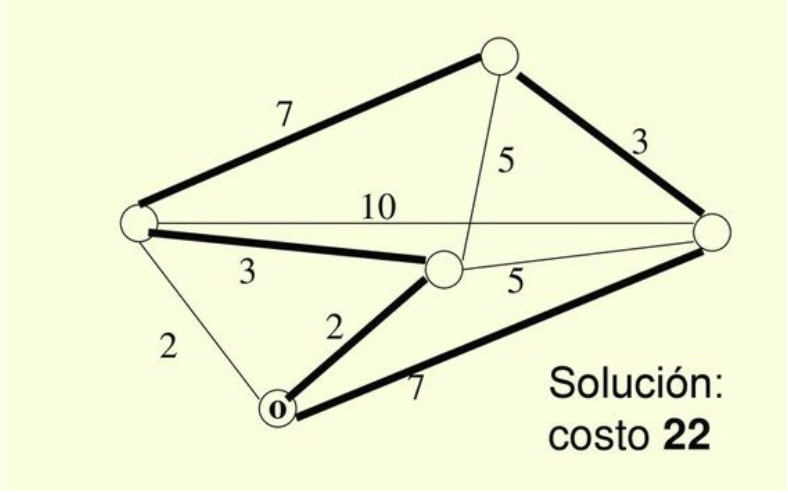
Se elige la ciudad más cercana entre las aún no visitadas:











1.2 Inserción

En este problema se crean al principio tres variables encargadas de almacenar cuales son las ciudades más al norte, más al oeste, y más al este de todas, de esta forma se crea un triángulo donde quedan ubicadas las ciudades.

A lo largo de la ejecución de nuestro algoritmo, se irá comparando cada una de las ciudades que conforman la lista obtenida desde el archivo con las ciudades más alejadas, siendo añadidas posteriormente al vector solución en caso de ser el más cercano a cualquiera de los puntos utilizados como referencia previamente.

- Podemos considerar al conjunto de nodos (o al conjunto de aristas) como una lista de candidatos. Las ciudades a visitar corresponden con los nodos y el camino entre ciudad y ciudad con las aristas.
- Una solución al problema sería el orden en el que hay que visitar cada nodo o en el que se debe recorrer las aristas.
- Una función de factibilidad en la que cada vez que se escoja a un candidato para incluirlo en la solución se deban cumplir los siguientes criterios:
  - Que no forme un ciclo con las aristas o los nodos ya escogidos excepto para completar el recorrido del viajante.
  - Que no se recorran dos veces la misma arista.

- La función de selección será la que hará que obtengamos distintas soluciones para el mismo problema. En este caso se irá eligiendo la ciudad más cercana al norte, este u oeste dependiendo de la iteración en la que se encuentre el bucle.
- Por último, como función objetivo tenemos que la suma de las longitudes de las aristas que constituyan la solución sea mínima.

Pseudocódigo:

```
FUNCIÓN Ciudad : struct
  nciudad, x, y : float

FUNCIÓN distancia(float x1, float y1, float x2, float y2) : double
  RETURN sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}

FUNCIÓN comparador : struct
  x, y : float
  FUNCIÓN operator()(const Ciudad &c1, const Ciudad &c2) : bool
        RETURN distancia(x, y, c1.x, c1.y) < distancia(x, y, c2.x, c2.y);

FUNCIÓN parsear_linea(string s) : vector<float>
  ret : vector<float>
  regex numero("-?[0-9]+\\.?[0-9]*");
  auto inicio_linea = sregex_iterator(s.begin(), s.end(), numero);
  auto fin_linea = sregex_iterator();
  FOR (auto i = inicio_linea; i != fin_linea; i++) {
```

```
        ret.push_back(stof((*i).str()));
  }
  RETURN ret;
}

FUNCIÓN parsear_datos(fstream &archivo) : vector<Ciudad>
  ret : vector<Ciudad>
  letra("[A-Z]") : regex
  vector<float> numeros_linea : vector<float>
  linea : string

  DO // Eliminar las lineas que no son numeros
        getline(archivo, linea);
   WHILE (regex_search(linea, letra));

  WHILE (!archivo.eof())
        IF (linea != "EOF" && linea != " EOF")
        numeros_linea = parsear_linea(linea);
        ret.push_back({numeros_linea[0], numeros_linea[1], numeros_linea[2]});
        getline(archivo, linea);
        ELSE
        break;

   RETURN ret;
}

FUNCIÓN main(int argc, char *argv[]) : int

  IF (argc != 2)
        cerr << "Uso: " << argv[0] << " <archivo>" << endl;
        RETURN 1;

solucion : vector<float>
archivo(argv[1]) : fstream
ciudades = parsear_datos(archivo) : vector<Ciudad>
archivo.close();
  Ciudad norte=ciudades[0];
  Ciudad este=ciudades[0];
  Ciudad oeste=ciudades[0];

 comparador comp;
  turno = 0;
```

```
 while (ciudades.size() > 0) {
        switch (turno % 3) {
        case 0:
        comp = {norte.x, norte.y};
        break;
        case 1:
        comp = {este.x, este.y};
        break;
        case 2:
        comp = {oeste.x, oeste.y};
        break;
        }
        min_element(ciudades.begin(), ciudades.end(), comp);
        Ciudad tmp = *min;
        ciudades.erase(min);
        solucion.push_back(tmp.nciudad);
        turno++;
 }

 FOR (auto i : solucion)
        cout << i << endl;
}
```

1.3 Libre

Como nueva estrategia para resolver el problema del Viajante de Comercio, hemos optado por un algoritmo que recorre todas las ciudades en base a la cercanía a la ciudad de punto de partida. Es decir, ordena las ciudades según la distancia a la que se encuentren de la ciudad de partida de menor a mayor. Para ello hemos tomado dos enfoques, uno usando el sort de la stl, más eficiente pero menos greedy y otro basado en el de cercanía que no actualiza la ciudad con la que comparar la distancia. En el caso de este último enfoque las partes del algoritmo greedy son las mismas que las del cercania así como la eficiencia.

- Podemos considerar al conjunto de nodos (o al conjunto de aristas) como una lista de candidatos. Las ciudades a visitar corresponden con los nodos y el camino entre ciudad y ciudad con las aristas.
- Una solución al problema es ordenar las ciudades por su distancia a la primera utilizando el algoritmo de inserción y recorrerlas de menor a mayor.

- La función selección que hemos diseñado para resolver este problema es la distancia respecto al punto de inicio.
- No hay función de factibilidad. Por último, como función objetivo tenemos que la suma de las longitudes de las aristas que constituyan la solución sea mínima.

1.3.1 Pseudocódigo

PSEUDOCÓDIGO VIAJANTE_SORT.CPP

```
FUNCIÓN Ciudad : struct
  nciudad, x, y : float

FUNCIÓN distancia(float x1, float y1, float x2, float y2) : double
  RETURN sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}

FUNCIÓN comparador : struct
  x, y : float
  FUNCIÓN operator()(const Ciudad &c1, const Ciudad &c2) : bool
    RETURN distancia(x, y, c1.x, c1.y) < distancia(x, y, c2.x, c2.y);

FUNCIÓN parsear_linea(string s) : vector<float>
  ret : vector<float>
  regex numero("-?[0-9]+\\.?[0-9]*");
  auto inicio_linea = sregex_iterator(s.begin(), s.end(), numero);
  auto fin_linea = sregex_iterator();
  FOR (auto i = inicio_linea; i != fin_linea; i++) {
    ret.push_back(stof((*i).str()));
  }
  RETURN ret;
}

FUNCIÓN parsear_datos(fstream &archivo) : vector<Ciudad>
  ret : vector<Ciudad>
  letra("[A-Z]") : regex
  vector<float> numeros_linea : vector<float>
  linea : string

  DO // Eliminar las lineas que no son numeros
    getline(archivo, linea);
   WHILE (regex_search(linea, letra));

  WHILE (!archivo.eof())
    IF (linea != "EOF" && linea != " EOF")
      numeros_linea = parsear_linea(linea);
      ret.push_back({numeros_linea[0], numeros_linea[1],
numeros_linea[2]});
```

```
      getline(archivo, linea);
     ELSE
      break;

   RETURN ret;
}

FUNCIÓN main(int argc, char *argv[]) : int

  IF (argc != 2)
    cerr << "Uso: " << argv[0] << " <archivo>" << endl;
    RETURN 1;

  solucion : vector<float>
  archivo(argv[1]) : fstream
  ciudades = parsear_datos(archivo) : vector<Ciudad>
  archivo.close();

  comp = {ciudades[0].x, ciudades[0].y} : comparador

  sort(ciudades.begin(), ciudades.end(),comp);

  //O también podemos usar este bucle while que es una aproximación más
greedy al problema pero menos eficiente
  //WHILE (ciudades.size() > 0)
    //auto min = min_element(ciudades.begin(), ciudades.end(), comp);
    //Ciudad tmp = *min;
    //ciudades.erase(min);
    //solucion.push_back(tmp.nciudad);


  FOR (auto i : ciudades)
    cout << i.nciudad << endl;
```

1.3.2 Eficiencia teórica.

En este caso hemos tratado dos formas distintas de implementar el algoritmo, una de ellas usando la función sort, y otra que es una aproximación más greedy al problema pero menos eficiente. La primera de ellas es de orden logarítmico y la otra de orden cuadrático.
Aquí podemos ver que el sort de la stl es logarítmico.
https://en.wikipedia.org/wiki/Sort_(C%2B%2B)#Complexity_and_implementations

1.4 Comparación de resultados

Para comparar las distintas aproximaciones diseñado un programa que calcula la distancia de total de una solución, el código de dicho programa se puede encontrar en el archivo distancia.cpp

	Cercanía	Inserción	Ordenación	Solución óptima
a280	3742.35	37859.8	16047.6	3203.12
att48	50446.1	172850	86687.3	43543.6
ulysses16	166.317	225.323	165.657	160.691
ulysses22	151.094	272.085	177.639	162.118

Como vemos en la tabla el enfoque de cercanía es el que mejores resultados obtiene, y el de inserción el que peor. El metodo de ordenacion obtiene unos resultados intermedios pero es el más eficiente en tiempo de ejecución.

2. Problema asignado: Ahorro en gasolina

2.1 Descripción del algoritmo

- Podemos considerar como conjunto de candidatos las gasolineras en las que el camionero debe parar o cada camino que se recorre entre ellas
- Una solución al problema sería el orden en el que hay que para en cada gasolinera.
- La función selección que hemos diseñado para resolver este problema consiste en una matriz de adyacencia donde aparecen todas las ciudades y todas las distancias. A continuación, al seleccionar un punto de la ciudad y otro punto de otra ciudad, calcula el mejor camino teniendo en cuenta los litros de gasolina que le quedan al depósito.
- En la función de factibilidad cada vez que se escoja un candidato para incluir en la solución se debe cumplir que el camino no recorra una gasolinera más de una vez y que no se cierre el ciclo del recorrido
- La función objetivo consiste en obtener el camino más corto posible de forma que se reduzcan al mínimo las paradas en las gasolineras del mapa.

2.2 Pseudocódigo

```
PSEUDOCÓDIGO GASOLINA.CPP

ALGORITMO AHORRO DE GASOLINA;
FUNCIÓN indice_min(vector distancia, vector visitado) : int
       ret,  min : int
       min = INT_MAX
            FOR(int i = 0; i < distancia.size(); i++)
                        IF(visitado[i] == false && distancia[i] <= min)
                  min = distancia[i];
                  ret = i;

            RETURN ret;


FUNCIÓN camino(vector prev, int destino)  : vector
      vector ret;
      WHILE (destino < prev.size())
            ret.push_front(destino);
            destino = prev[destino];

      RETURN ret;


FUNCIÓN dijkstra(matriz adyacencia, int origen, int destino) : int,
vector
      vector distancia = INT_MAX
      vector visitado = false
      vector prev = -1

      distancia[origen] = 0;

      FOR(int i = 0; i < adyacencia.size() - 1; i++)
            int min = indice_min(distancia, visitado);
            visitado[min] = true;
            FOR(int j = 0; j < adyacencia.size(); j++)
            IF(!visitado[j] && adyacencia[min][j] &&
            distancia[min] + adyacencia[min][j] < distancia[j])
                  prev[j] = min;
                  distancia[j] = distancia[min] + adyacencia[min][j];

      return distancia[destino], camino(prev, destino);
```

2.3 Eficiencia:

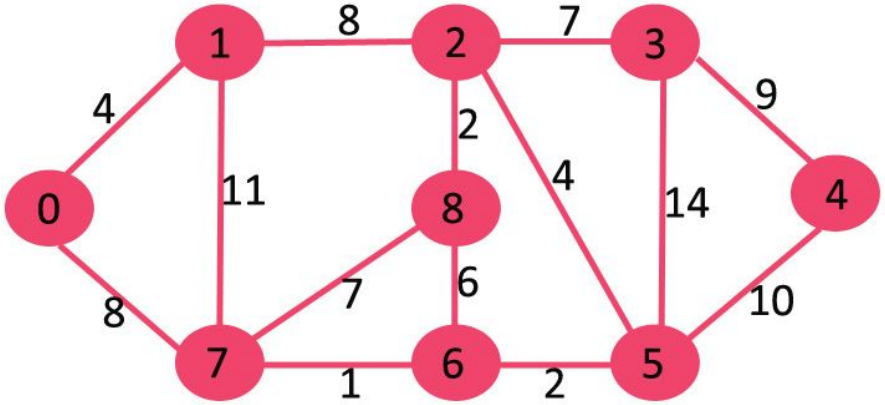
El algoritmo consiste en n-1 iteraciones, como máximo. En cada iteración, se añade un vértice al conjunto y se identifica al vértice con la menor etiqueta. El número de estas operaciones está acotado por n-1. También se realizan una suma y una comparación para actualizar dicha etiqueta de cada uno de los vértices que no están en el conjunto. Por tanto en cada iteración se realizan 2(n-1).
La eficiencia del algoritmo en el peor caso es $O(n^2)$.

2.4 Generación del problema

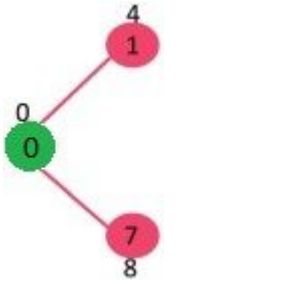
Pseudocódigo:

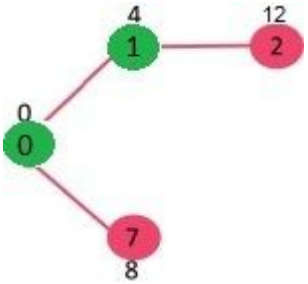
```
Introducir parametros, tam_matriz, max_numero,
probabilidad_ceros, formato

Crear matriz[tam_matriz][tam_matriz]

for(fila = 0 to final){
    for(columna = fila + 1 to final){ //Trianguliza
superiormente
      n = genera_numero_aleatorio(probabilidad_ceros,
max_numero) //Crea numero aleatorio cuyo máximo valor posible
es "max_numero" y con probabilidad "probabilidad_ceros" de ser
0
      matriz[i][j] = n;
      matriz[j][i] = n; //También rellenar en la triangular
inferior
    }
}

if(formato == 0){
    Impresión para pegar en el código directamente
}else{
    Impresión para archivo de texto
}
```

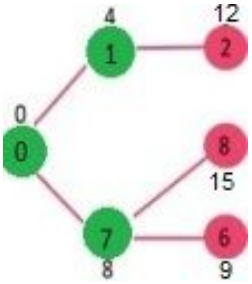
2.5 Escenarios de ejecución

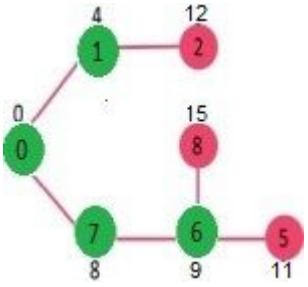


Los puntos verdes son los que se van incluyendo en el set que almacena los puntos del camino mínimo.

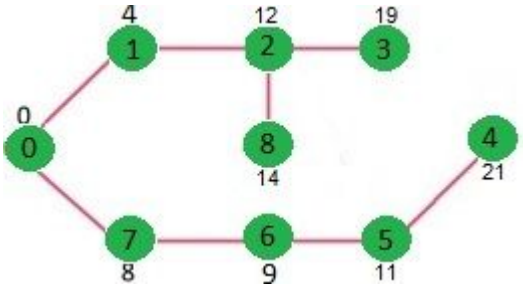








...


4
12
19
1
2
3
0
0
8
4
14
21
7
6
5
8
9
11