Universidad de Granada

Algorítmica

Práctica 2: Algoritmo Divide y Vencerás

Autora:
Elena Merelo Molina

Abril de 2018

UNIVERSIDAD
DE GRANADA

1 Problema a resolver

Un electricista necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas, en la tarea iésima tardará t minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente y ésta es inversamente proporcional al tiempo que tardan en atenderles, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de atención de los clientes (desde el inicio hasta que su reparación es efectuada). Tareas a realizar: Diseñar un algoritmo greedy para resolver lo anterior. Demostrar que el algoritmo obtiene la solución óptima. Modificar el algoritmo anterior para el caso de una empresa en la que se disponga de los servicios de más de un electricista.

2 Explicación de mi solución

Para resolver el problema he creado la clase $matriz_de_adyacencia$, cuyo fichero de cabecera es el siguiente:

```
#ifndef __MATRIZ_DE_ADYACENCIA__
#define __MATRIZ_DE_ADYACENCIA__

#include <vector>
#include <utility>
#include <fstream>
#include <assert.h>
#include <algorithm>
#include <math.h>
#include <set>
#include <iostream>

using namespace std;

typedef pair<double, double> ciudad;

class matriz_de_adyacencia{
private:
  vector<vector<double> > m;
  vector<ciudad > ciudades;
  vector<bool> visitadas;

  double distancia_euclidea(ciudad c1, ciudad c2);
  void rellenar_matriz(const vector<ciudad> &v);
  bool recorrido_terminado();
  bool forma_ciclo(vector<int> recorrido, int nodo);

```

```
28
29 public:
30   //Crea la matriz de adyacencia a partir del fichero pasado como parametro
31   matriz_de_adyacencia(const char* fichero);
32
33   //Obtiene el camino minimo desde la fila i columna j de la matriz
34   vector<int> min_path(int i, double &longitud);
35 };
36
37 #endif
```

Veamos uno a uno los métodos y atributos de esta clase.

Atributos privados

• m es la matriz de adyacencia, que relleno de manera que sea triangular superior y sin tener en cuenta la diagonal principal.

• *ciudades* es un vector con las coordenadas x e y de cada ciudad (es de tipo `vector<ciudad>`, donde ciudad es un par formado por la abscisa y ordenada de la ciudad correspondiente (`$pair<double, doubl` definido así simplemente para que quede el código más claro).

• *visitadas* es otro vector en el cual por cada ciudad hay una casilla que será `true` si dicha ciudad ha sido recorrida por el electricista y `false` en caso contrario.

Métodos privados

• `double distancia_euclidea(ciudad c1, ciudad c2);` halla la distancia euclídea entre dos ciudades dadas:

```
1 double matriz_de_adyacencia::distancia_euclidea(ciudad c1, ciudad c2){
2   return sqrt(pow(c2.first - c1.first, 2) + pow(c2.second - c1.second, 2));
3 }
```

• `void rellenar_matriz(const vector<ciudad> &v);` dado el vector de ciudades rellena la matriz con la distancia entre ellas:

```
1  void matriz_de_adyacencia::rellenar_matriz(const vector<ciudad> &v){
2    int n= v.size();
3
4    //Inicializamos la matriz triangular superior
5    m.resize(n);
6
7    for(unsigned int i= 0; i< m.size(); i++)
8      m[i].resize(n);
9
10   for(int i= 0; i< n; i++)
11     for(int j= i+1; j< n; j++)
12       m[i][j]= distancia_euclidea(v[i], v[j]);
13 }
```

El método es simple: la matriz es realmente un vector de vectores, luego creamos el vector del que van a "colgar" el resto de vectores con un tamaño igual al número de ciudades, y luego cada componente de éste le reservamos también ese espacio, evitando así que se den violaciones de segmento cuando queremos acceder a las componentes de la matriz. Luego rellenamos el triángulo superior con la distancia euclídea entre las ciudades.

- `bool recorrido_terminado();` Chequea si se todas las ciudades han sido visitadas:

```
bool matriz_de_adyacencia::recorrido_terminado(){
  return count(visitadas.begin(), visitadas.end(), false) == 0;
}
```

Como habíamos inicializado todas las componentes del vector de ciudades visitadas a false, el recorrido terminará cuando todas sean true; contamos pues el número de "falses" y si es 0 es que el electricista ha terminado.

- `bool forma_ciclo(vector<int> recorrido, int nodo);` Comprueba si se forma un ciclo en el grafo, mirando si el nodo pasado como parámetro ha sido recorrido ya o no:

```
bool matriz_de_adyacencia::forma_ciclo(vector<int> recorrido, int nodo){
  for(unsigned int i= 0; i< recorrido.size(); i++)
    if(recorrido[i] == nodo)
      return true;

  return false;
}
```

Métodos públicos

- Constructor de la matriz de adyacencia a partir del fichero pasado como argumento:

```
matriz_de_adyacencia::matriz_de_adyacencia(const char *fichero){
  int num_cities, n;
  double x, y;
  string cabecera;
  ifstream flujo;

  flujo.open(fichero);

  if(!flujo){
    cout << "No se pudo abrir el fichero";
    exit(-1);
  }

  //Leemos hasta el primer espacio en blanco, lo correspondiente a "
    DIMENSION:"
  flujo >> cabecera;

  //Lo siguiente son el numero de ciudades:
```

```
  flujo >> num_cities;

  //Creamos la lista con las ciudades y sus coordenadas en el mapa
  for(int i= 0; i< num_cities; i++){
    flujo >> n;
    flujo >> x;
    flujo >> y;
    ciudades.push_back(make_pair(x, y));
    visitadas.push_back(false);
  }

  rellenar_matriz(ciudades);
  flujo.close();
}
```

Este método abre el fichero y lee la dimensión con `flujo >> cabecera`, al leer `>>` hasta el primer `whitespace`, en este caso un espacio en blanco, y lo consume, de manera que lo próximo a leer es el número de ciudades, dato que guardamos en una variable. Posteriormente leemos el número de ciudad, su coordenada x y su coordenada y, y con estas dos últimos hacemos un par que insertamos en el vector de ciudades, a la vez que inicializamos las componente de ciudades visitadas a false. Una vez se ha hecho esto para todas las ciudades rellenamos la matriz y cerramos el flujo.

- Algoritmo greedy propiamente dicho que calcula el camino mínimo a partir de una ciudad i:

```
vector<int> matriz_de_adyacencia::min_path(int i, double &longitud){
  int n= ciudades.size(), j;
  assert( i >= 0 && i < n);

  set<pair<double, int> > posibilidades;
  set<pair<double, int> >::iterator it;
  double min_dist, destino;
  vector<int> r;
  r.push_back(i);
  visitadas[i]= true;

  //Mientras haya ciudades por recorrer
  while(!recorrido_terminado()){
    for(j= 0; j< n; j++){
      //Si estamos en el triangulo superior de la matriz de adyacencia
      if(!forma_ciclo(r, j)){
        if( i > j)
        posibilidades.insert(make_pair(m[j][i], j));  //insertamos la
    distancia entre las ciudades y a que ciudad va

        //Si no esta en el triangulo superior obtenemos la coordenada
    simetrica
```

```
21          else if( i< j)
22          posibilidades.insert(make_pair(m[i][j], j));
23
24          //Si i == j no se hace nada.
25        }
26      }
27
28      //Como el set ordena automaticamente sus componentes, en la primera
        posicion estara la minima distancia
29      it= posibilidades.begin();
30      min_dist= it->first;
31      destino= it->second;
32
33      //Sumamos la distancia a la cantidad de camino recorrido
34      longitud += min_dist;
35
36      //Aniadimos la ciudad destino a la lista de ciudades recorridas
37      r.push_back(destino);
38      visitadas[destino]= true;
39      //Nos situamos en la ciudad destino y buscamos desde ahi el minimo
        camino a la siguiente que no haya sido recorrida
40      i= destino;
41
42      posibilidades.clear();
43    }
44    return r;
45  }
```

Primeramente nos aseguramos de que la ciudad introducida es válida, y creo un conjunto de `pair<double, int>`, donde el double será una distancia euclídea, la que hay entre la ciudad i y otra j, y el int se corresponde con el índice de la ciudad j, para poder luego más fácilmente saber qué distancia corresponde a quién. Hice la elección de este contenedor de la STL ya que al insertar elementos en un `set` éstos se ordenan de menor a mayor automáticamente, teniendo un set de pair los ordena fijándose en la primera coordenada y si hay dos primeras coordenadas iguales se fija en la segunda. Así, solo consultando el primer elemento del conjunto tendremos la distancia mínima, sin tener que usar for ni nada.

Introducimos el nodo origen i a la lista de ciudades recorridas, lo ponemos a true en el vector de ciudades visitadas, y mientras haya ciudades por visitar recorremos la columna asociada a la ciudad i; se comprueba que no forme ciclo y si $i > j$ le damos la vuelta a los índices por la

representación de la matriz que hemos escogido, veámoslo en un ejemplo sencillo, si fuera 4x4:

$$M = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Entonces si queremos obtener la componente $m[0][j]$ no hay problema para ningún j, al estar rellena esa fila, mas sin embargo si tomamos $m[2][j]$ para $j = 0$ tendríamos un 0, pero hay distancia entre esas ciudades, únicamente que la hemos puesto en la posición $m[j][2]$, es por esto que cuando $i > j$ hemos de hacer el cambio. Después inserta esa distancia junto con la ciudad destino en el set, sumamos a la longitud la primera componente del primer elemento del set que, como hemos dicho, contiene la mínima distancia, y añadimos la ciudad j a la lista de ciudades recorridas. Por último nos posicionamos en dicha ciudad, haciendo $i = destino$, vaciamos el set con las posibilidades y se vuelve a empezar, devolviendo cuando se termina el vector con el recorrido y en la variable longitud, al haber sido pasada por referencia estará almacenado el peso total del camino.