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 <limits.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();
  void clear();
```

```
28
29
30 public:
31   //Crea la matriz de adyacencia a partir del fichero pasado como parametro
32   matriz_de_adyacencia(const char* fichero);
33
34   //Muestra la matriz de adyacencia
35   void show_matrix();
36
37   //Muestra el camino y su longitud
38   void show_path(vector<int> path, double longitud);
39
40   //Busca la ciudad mas cercana a una dada
41   int ciudad_mas_cercana(int city, double &min_dist);
42
43   //Obtiene el camino minimo desde la fila i columna j de la matriz
44   vector<int> min_path(int i, double &longitud);
45
46   //Aniade al recorrido total la vuelta al punto de partida, con el peso adicional
       que eso conlleve
47   vector<int> cierra_camino(vector<int> recorrido, double &longitud);
48
49   //Obtiene el camino minimo de todos los posibles, comparando entre todos los
       recorridos que hay (tantos como ciudades)
50   vector<int> recorrido_optimo(double &longitud);
51
52   //Reparte el recorrido entre varios electricistas
53   vector<vector<int> > reparto_multiple(int origen, int n_electricians, double &
       longitud);
54
55 };
56
57 #endif
```

Veamos uno a uno todos 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:

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

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

```
void matriz_de_adyacencia::rellenar_matriz(const vector<ciudad> &v){
  int n= v.size();
  //Inicializamos la matriz triangular superior
  m.resize(n);
  for(unsigned int i= 0; i< m.size(); i++)
    m[i].resize(n);
  for(int i= 0; i< n; i++)
    for(int j= i+1; j< n; j++)
      m[i][j]= distancia_euclidea(v[i], v[j]);
}
```

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 ese mismo espacio, evitando así que se den violaciones de segmento cuando queramos acceder a las componentes de la matriz. Por último rellenamos el triángulo superior con la distancia euclídea entre las ciudades.

• `bool recorrido_terminado();` Chequea si todas las ciudades han sido visitadas (cuando una ciudad se recorrer se pone su componente del vector `visitadas` a `true`, por ello si hay 0 "falses" ya habremos finalizado):

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

• `void clear();` rellena el vector de ciudades visitadas con false, para poder empezar un nuevo recorrido:

```
void matriz_de_adyacencia::clear(){
  visitadas.clear();
  visitadas.resize(m.size(), 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 nÃºmero 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.resize(num_cities, false);
  rellenar_matriz(ciudades);
  flujo.close();
}
}
```

Este método abre el fichero y obtiene la dimensión con `flujo >> cabecera`, al leer `>>` hasta el primer `whitespace`, en este caso un espacio en blanco, y consumirlo. Así, 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 (`pair <double, double>` que insertamos en el vector de ciudades. Una vez fuera del bucle, inicializamos las componentes de ciudades visitadas a false, rellenamos la matriz con los datos sobre las ciudades recogidos y cerramos el flujo.

- Método que muestra la matriz de adyacencia, recorriendo sus filas y columnas:

```
void matriz_de_adyacencia::show_matrix(){
  for(int i= 0; i< m.size(); i++){
    for(int j= 0; j< m.size(); j++)
      cout << m[i][j] << " ";
    cout << "\n";
  }
```

```
}
```

- `void show_path(vector<int> path, double longitud);` Muestrael recorrido y la longitud del mismo del camino pasado como parámetro:

```
void matriz_de_adyacencia::show_path(vector<int> v, double longitud){
  cout << "Peso: " << longitud << ", recorrido: ";
  for(unsigned int i= 0; i< v.size(); i++)
    cout << v[i] << " ";
}
```

- `int ciudad_mas_cercana(int city, double &min_dist);` Obtiene la ciudad más cercana a una dada, guardando la distancia en la variable correspondiente:

```
int matriz_de_adyacencia::ciudad_mas_cercana(int i, double &min_dist){
  int j, n= ciudades.size();
  set<pair<double, int> > posibilidades;
  set<pair<double, int> >::iterator it;

  min_dist= 0;

  for(j= 0; j< n; j++){
    //Si estamos en el triangulo superior de la matriz de adyacencia
    if(!visitadas[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
      else if( i< j)
      posibilidades.insert(make_pair(m[i][j], j));

      //Si i == j no se hace nada.
    }
  }

  //Como el set ordena automaticamente sus componentes, en la primera posicion estara la minima distancia
  it= posibilidades.begin();
  min_dist= it->first;
  return it->second;
}
```

Como vemos, en primer lugar se crea 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 qué. 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, en este caso al ser un set de pair los ordena fijándose en la primera coordenada y si las dos primeras coordenadas son iguales se fija en la segunda. Consecuentemente, solo consultando el primer elemento del conjunto tendremos la distancia mínima, sin tener que usar un bucle `for` ni nada.

Posteriormente recorremos la fila asociada a la ciudad i, añadiendo las distancias de i a j al set, comprobando que no se forme ciclo (con `if(!visitadas[j])`, mirando si ha sido recorrida ya la ciudad o no) y si $i > j$ le damos la vuelta a los índices por la representación de la matriz que he escogido (es triangular superior), 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 insertamos 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 devolvemos la ciudad j.

- 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;
  longitud= 0;

  double min_dist;
  vector<int> r;

  //Mientras haya ciudades por recorrer
  while(!recorrido_terminado()){
    //Aniadimos la ciudad destino a la lista de ciudades recorridas
    r.push_back(i);
    visitadas[i]= true;

    j= ciudad_mas_cercana(i, min_dist);
    //Sumamos la distancia a la cantidad de camino recorrido
    longitud += min_dist;

    //Nos situamos en la ciudad destino y buscamos desde ahi el minimo camino a la
   siguiente que no haya sido recorrida
    i= j;
  }
```

```
21   return r;
22 }
```

Su funcionamiento es sencillo; en una primera iteración se pone el índice de la ciudad i a `true` en el vector `visitadas`, que es donde nos encontramos, y se añade a la lista de ciudades recorridas. Posteriormente se busca la ciudad más cercana a i, j, se le suma a la longitud total la distancia entre ambas y nos "movemos" a esta última, repetiendo el procedimiento mientras haya ciudades que no han sido visitadas. Una vez se sale del `while` se devuelve el vector que contiene el camino seguido y en la variable `longitud`, al haber sido pasada por referencia, se habrá almacenado el peso total del camino.

- `vector<int> cierra_camino(vector<int> recorrido, double &longitud);` Añade al vector pasado como parámetro la ciudad origen, lo guardaa en otro vector que se devuelve y sumando esa distancia adicional a longitud.

```
vector<int> matriz_de_adyacencia::cierra_camino(vector<int> recorrido, double &
    longitud){
  vector<int> f= recorrido;
  int j= recorrido.back(), i= recorrido.front();

  if(i > j)
    longitud += m[j][i];
  else
    longitud += m[i][j];

  f.push_back(i);
  return f;
}
```

- `vector<int> recorrido_optimo(double &longitud);` busca, de entre todos los recorridos posibles, el mínimo. Para ello usa el método `min_path` y desde la ciudad 0 a la dimension-1 va calculando el camino que parte de ella y su peso, comparádolas de manera que se queda con el que tiene menor longitud.

```
vector<int> matriz_de_adyacencia::recorrido_optimo(double &longitud_min){
  double longitud;
  longitud_min= LONG_MAX;
  vector<int> actual, min;

  for(unsigned int i= 0; i< m.size(); i++){
    longitud= 0;
    clear();
    actual= min_path(i, longitud);
    if(longitud< longitud_min){
      min= actual;
      longitud_min= longitud;
```

```
    }
  }
  return min;
}
```

- `vector<vector<int> > reparto_multiple(int i, double &longitud);` Reparte el recorrido entre varios electricistas.

```
vector<vector<int> > matriz_de_adyacencia::reparto_multiple(int city, int n, double &longitud){
  assert(n > 0);
  assert(city >= 0 && city < ciudades.size());
  clear();

  int i;
  double dist= 0;
  vector<vector<int> > repartos;
  repartos.resize(n);

  for(i=0; i< repartos.size(); i++)
    repartos[i].resize(1);

  longitud= 0;
  if(n == 1){
    vector<int> r= min_path(city, longitud);
    repartos[0].erase(repartos[0].begin()); //Para ajustar el resize a 1.

    for(i= 0; i< r.size(); i++)
      repartos[0].push_back(r[i]);
  }else{
    /*Todos los electricistas parten de la ciudad city, por lo que la primera
    componente del vector que contiene el recorrido que hace cada uno sera city.*/
    for(i= 0; i< n; i++)
      repartos[i][0]=city;

    //Como ya ha sido visitada ponemos su componente a true
    visitadas[city]= true;
    i= 0;

    while(!recorrido_terminado()){
      //Movemos a cada electricista de la ciudad en la que se encuentra a la mas cercana
      if(i < n){
        repartos[i].push_back(ciudad_mas_cercana(repartos[i].back(), dist));
        visitadas[repartos[i].back()]=true;
        longitud += dist;
        i++;
```

```
38          }
39          else if(i == n)
40            i= 0;
41        }
42    }
43    return repartos;
44  }
```

El algoritmo es sencillo: creamos un vector con un vector por electricista, éste segundo contendrá el recorrido que ha hecho cada uno, y como primera componente de todos ponemos la ciudad origen `city`. Luego, mientras haya ciudades que no han sido visitadas se añade de uno en uno a cada vector `repartos[i]` la ciudad más cercana a la ciudad $i-1$, la última que haya. Cuando i sea igual al número de electricistas se resetea y pone a 0, para que cada electricista tenga un número equitativo de ciudades que recorre. Finalmente se devuelve la matriz con los repartos efectuados.