Universidad de Granada

Algorítmica

Memoria de Prácticas

Práctica III: Algoritmos greedy

Autores:
Elena Merelo Molina
Antonio Gámiz Delgado

May 1, 2018

Universidad de Granada

1 Problema

Dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

2 Problema del Viajante de Comercio (solución 1)

Para esta solución hemos precisado de un método aparte:

```
int graph::nearest_city(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(!visited[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 quién. Hicimos 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. Así, solo consultando el primer elemento del conjunto tendremos la distancia mínima, sin tener que usar 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 y si $i > j$ le damos la vuelta a los índices por la representación de la matriz que hemos 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. Visto esto, esta solución primera sería:

```
vector<int> graph::min_path1(int i, double &l){
  int n= cities.size(), j;
  l= 0;
  assert( i >= 0 && i < n);

  double min_dist;
  vector<int> r;

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

    j= nearest_city(i, min_dist);
    //Sumamos la distancia a la cantidad de camino recorrido
    l += 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;
  }
  close_path(r, l);

  return r;
}
```

Su funcionamiento es sencillo; en una primera iteración se pone el índice de la ciudad i a `true` en el vector `visited`, 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 recorrido seguido y en la variable `l`, al haber sido pasada por

referencia, se habrá almacenado el peso total del camino. El método `finished_path()` cuenta el número de "falses" que hay en `visited`, y devuelve `true` si hay 0, lo que significará que ya no hay más ciudades por visitar. (`bool graph::finished_path(){ return count(visited.begin(), visited.end(), false) == 0; }`.

3 Problema del Viajante de Comercio (solución 2)

A continuación vamos a explicar como hemos diseñado la solución 2 del problema del viajante de comercio, viendo primeramente algunas funciones auxiliares que hemos usado para facilitar la resolución del problema.

3.1 Funciones auxiliares

```
int graph::westernmost_city(){
  double x=LONG_MAX;
  int index;
    for(int i=0; i<cities.size(); i++)
    if( cities[i].first < x ){
      x=cities[i].first;
      index=i;
    }
  return index;
}
```

Listing 1: Encontrar la ciudad más al oeste

```
int graph::northernmost_city(){
  double x=-LONG_MAX;
  int index;
  for(int i=0; i<cities.size(); i++)
    if( cities[i].second > x ){
      x=cities[i].second;
      index=i;
    }
  return index;
}
```

Listing 2: Encontrar la ciudad más al norte

```
int graph::easternmost_city(){
  double x=-LONG_MAX;
  int index;
  for(int i=0; i<cities.size(); i++)
    if( cities[i].first > x ){
      x=cities[i].first;
      index=i;
    }
```

```
  return index;
}
```

Listing 3: Encontrar la ciudad más al este

Estas funciones no tienen mucho que explicar, únicamente recorren un vector llamado *cities* en el que están almacenadas las ciudades y hayan el mínimo (ciudad más al oeste) y el máximo (ciudad más al este) de las coordenadas x, además del máximo de las coordenadas y (más al norte).

```
double graph::total_weight(vector<int> path){
  double l=0;
  for(int i=0; i<path.size()-1; i++){
    l+=get_weight(path[i], path[i+1]);
  }
  //Para cerrar el camino
  close_path(path, l);
  return l;
}
```

Listing 4: Hallar el peso de un camino cerrado

Esta función se encarga de hallar el peso de un camino cerrado en el grafo, que es suma de las distancias euclídeas entre las ciudades por las que pasa el mismo,añadiendo al final la distancia de la primera ciudad a la última para cerrar el camino.

3.2 Implementación

Para implementar el algoritmo, hemos dividido el problema en 3 funciones, *particular_min*, *general_min* y *min_path*2.

Veamos qué hace cada una de ellas:

```
pair<int, double> graph::particular_min(vector<int>& path, int x){
  double aux=0, min=LONG_MAX;
  int index;

  for(int i=0; i<=path.size(); i++){
    path.insert(path.begin()+i, x);

    aux=total_weight(path);
    if( aux<min ){
      index=i;
      min=aux;
    }

    path.erase(path.begin()+i);
  }

  return make_pair(index, min);
```

```
}
```

Listing 5: particular_min

El objetivo de esta función es, dado un recorrido de ciudades ($vector < int > path$), encontrar el índice $index$ tal que al insertar la ciudad x, el peso de ese nuevo camino sea mínimo. Para ello, insertamos la ciudad x en cada posición posible, calculamos el peso de ese camino, y nos quedamos con el mínimo. Devolvemos un par formado por ese índice y el peso del camino mínimo, que serán usados por la siguiente función que hemos implementado.

```
pair<int, int> graph::general_min(vector<int> &r)
{
  set<pair<double,pair<int, int> > >posibilities;
  pair<int, double> aux;

  for(int i=0; i<cities.size(); i++){
    if( !visited[i] ){
      aux=particular_min(r, i);
      posibilities.insert( pair<double,pair<int, int> >( aux.second, make_pair(i, aux
    .first) ) );
    }
  }

  return make_pair( (*posibilities.begin()).second.first, (*posibilities.begin()).
    second.second );
}
```

Listing 6: general_min

El objetivo de esta función es, dado un camino r, encontrar la siguiente ciudad a insertar de forma que el incremento del camino sea mínimo. Para ello, crea un *set* formado por:

- pair<int, int >: en el primer miembro contiene el indice, $i\_$ de la posible ciudad a insertar, y en el segundo miembro, la posición de r, $j\_$ donde debería ser insertada para que el peso fuera mínimo.
- double: peso del camino resultante de insertar la ciudad $i\_$ en la posición $j\_$.

El contenedor *set* ordena de menor a mayor automáticamente los elementos (según el valor del primer miembro del par), por lo que una vez relleno con todas las posibilidades, solo hay que devolver un par formado por los índices $i\_$ y $j\_$.

[Aclaración] *visited* es un vector de *bool* que forma parte de la clase *graph*, y contiene *true* en la posición i-ésima si la ciudad i ya forma parte de la solución.

```
vector<int> graph::min_path2(double &l){
  clear();
  vector<int> r;

  r.push_back( westernmost_city() );
```

```
  r.push_back( easternmost_city() );
  r.push_back( northernmost_city() );

  visited[r[0]]=true; visited[r[1]]=true; visited[r[2]]=true;

  pair<int, int> new_node;
  while( r.size() < cities.size() ){
    new_node = general_min(r);
    visited[new_node.first]=true;
    r.insert(r.begin()+new_node.second, new_node.first);
  }

  l+=total_weight(r);
  return r;
}
```

Listing 7: min_path2

Esta función es la encargada de crear el camino mínimo, ejecuta la función anteriormente descrita (*general_min*) hasta que el camino está completo, es decir, pasa por todos los vértices.

Primero llama a *clear* para asegurarse de que el vector *visited* antes comentado esté completamente inicializado a $false$.

Luego inicializa r con las tres ciudades que hemos mencionado antes y ajusta *visited* adecuadamente.

Por último, llama a la función anterior descrita hasta que la solución esté completa.

[Aclaración] No hace falta comprobar si se forman ciclos ya que, al aparecer cada vértice una, y sólo una vez, no se puede dar esa situación.

4 Problema del Viajante de Comercio (solución 3)

```
vector<int> graph::min_path3(int i, double &l){
  clear();
  vector<int> r;

  set<pair<double, int> > posibilities;
  for(int j=0; j<cities.size(); j++)
    if(i!=j) posibilities.insert(make_pair( get_weight(i,j) , j) );

  r.push_back(i);
  r.push_back( (*posibilities.begin()).second );
  r.push_back( (*(++posibilities.begin())).second );

  visited[r[0]]=true; visited[r[1]]=true; visited[r[2]]=true;

  pair<int, int> new_node;
  while( r.size() < cities.size() ){
    new_node = general_min(r);
```

```
18      visited[new_node.first]=true;
19      r.insert(r.begin()+new_node.second, new_node.first);
20    }
21
22    l=total_weight(r);
23    return r;
24  }
```

Listing 8: min_path3

La tercera solución es una hecha por nosotros. Lo que hemos hecho es combinar los dos algoritmos anteriores, es decir, hemos elegido un nodo arbitrario i por el que empezar, y aplicando la heurística del camino más cercano (la primera solución) hemos elegido los dos nodos más cercanos a i.

Una vez con que tenemos ese triángulo, hemos aplicado la heurística del triángulo (la segunda solución).