Universidad de Granada

Algorítmica

Memoria de Prácticas

# Práctica III: Greedy

*Autores:*
Antonio Gámiz Delgado

April 25, 2018

Universidad de Granada

# 1 Problema

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 $i$-ésima tardará $t_i$ minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, 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 sea efectuada).

# 2 Diseño del algoritmo Greedy

A continuación voy a explicar el algoritmo que he diseñado para resolver este problema, primero voy a exponer de las partes (más bien funciones) de las que se compone y luego explicaré como trabaja.

```
vector<node> greedy(graph &g, int begin)
{
  vector<node> nodes=g.get_nodes();
  vector<node> solution;

  int index_nextNode;
  struct node current_node=nodes[begin];
  bool first_time=true;
  solution.push_back(current_node);

  struct node nextNode;
  first_time=true;

    while(solution.size() < nodes.size())
    {
    first_time=true;
      do
      {
        index_nextNode=nodeHeuristic(g, current_node, (first_time)? 0:g.get_weight(
    current_node.label, nextNode.label), solution);
        if( index_nextNode==-1 ) return vector<node>();

        nextNode=nodes[index_nextNode];

        first_time=false;
      } while( cycle( solution, nextNode ) );

    solution.push_back(nextNode);
    current_node=nextNode;
  }

  return solution;
}
```

Listing 1: Función obejtivo

Esta es la función que nos interesa optimizar. Recibe por referencia un grafo $g$ (clase implementada en *graph.c*, representación interna: usando la matriz de adyacencia) y un índice natural *begin* que indica desde que nodo queremos empezar el algoritmo (ya que la solución de este depende del nodo inicial elegido). Lo que hace el algoritmo está más o menos claro, empieza desde el nodo $nodes[begin]$ (*nodes* contiene los nodos del grafo), y lo añadimos a la solución. Luego vamos seleccionando el nodo tal que el camino que forme con el anterior tenga peso mínimo, haciendo uso de las funciones que describo a continuación.

```
bool cycle(vector<node>& v, struct node new_node){ return is_in(new_node.label, v); }
```

Listing 2: Función para validar la solución

Como la nota indica, esta función se encarga de validar la solución, es decir, de comprobar que el nodo que queremos añadir a la solución no forme ciclo. Esto se consigue comprando que el vector a añadir no esté ya en la solución, ya que si estuviera, formaría ciclo. Para ello, uso una función auxiliar *is_in* que comprueba si el nodo $new_node$ está en el vector $v$.

```
int nodeHeuristic(graph &g, struct node n, float min_bound, vector<node> taken){
  float min=LONG_MAX;
  int index_min=-1;
  float current_weight;

  for(int i=0; i<g.size(); i++)
  {
    current_weight=g.get_weight(i, n.label);
    if( current_weight > 0 && current_weight < min && current_weight > min_bound && !is_in
    (i, taken) )
      {
        min=current_weight;
        index_min=i;
      }
    }

    return index_min;
}
```

Listing 3: Función de selección

Esta función se encarga de, dado un grafo $g$, y un nodo $n$, buscar el nodo con índice *index_min* (el índice con el que es representado en la matriz de adyacencia) tal que el camino de $n$ a ese nodo sea el mínimo.

Además, si el nodo con índice *index_min* que encuentra está ya en la solución ($vector < node > taken$) lo desecha y busca otro.

El parámetro *min_bound* es muy importante, se encarga de fijar la cota inferior del mínimo, es decir, la función *nodeHeuristic* busca el arco en la matriz de adyacencia justamente superior al valor de *min_bound*. Esto lo hago debido a que si la función ha sido llamada alguna vez (cosa que compruebo con la variable $bool\ first\_time$ en *greedy*), y la función *cycle* ha detectado que ese nodo forma un ciclo, entonces hay que desecharlo, por lo que para que no encuentre ese nodo otra vez, busco el valor superior. Como vemos,

si esta función no encuentra un nodo que no forme ciclo con distancia mínima, entonces devuelve -1, indicando a la función *greedy* que el algoritmo no puede encontrar una solución.

## 3 Adaptación a n electricistas

Para adaptar el problema a un número $n$ de electricistas he adapto el algoritmo *greedy* (el resto de funciones no es necesario adaptarlas) de la siguiente forma:

```
vector<vector<node> > greedy(graph &g, int begin, int n_electricians)
{
  vector<node> nodes=g.get_nodes();
  vector<vector<node> > solution(n_electricians);
  vector<node> taken;

  int index_nextNode;

  struct node current_node=nodes[begin];
  taken.push_back(current_node);
  bool first_time=true;

  for(int i=0; i<n_electricians; i++)
  solution[i].push_back(current_node);


  struct node nextNode;
  first_time=true;
  int j=0;
    while( taken.size() < nodes.size() )
    {
      if(j==n_electricians) j=0;
      current_node=solution[j].back();

      /*------------searching next node-----------*/
      first_time=true;
      do
      {
        index_nextNode=nodeHeuristic(g, current_node, (first_time)? 0:g.get_weight(
    current_node.label, nextNode.label), taken);
        if( index_nextNode==-1 ) return vector<vector<node> >();
        nextNode=nodes[index_nextNode];

        first_time=false;
      } while( cycle( solution[j], nextNode ) );
      /*------------searching next node-----------*/

      solution[j].push_back(nextNode);
      taken.push_back(nextNode);
      j++;
    }
```

```
41
42   return solution;
43 }
```

Listing 4: Adaptación del algoritmo

Lo que he hecho es "repartir" el trabajo entre los $n$ electricistas. Todos empiezan desde el mismo nodo (esto lo he supuesto yo porque parece evidente que todos los electricistas salgan de la misma empresa), de forma que el primer electricista, va al nodo más cercano, el siguiente electricista va a al segundo nodo más cercano, y así hasta que el $n$-ésimo electricista vaya al $n$-ésimo nodo más cercano.

Todos estos caminos los he guardado en $vector < vector < node >> solution$, y los nodos que ya han sido recorrido los he ido guardando en $vector < node > taken$.

El resto es exactamente igual al algoritmo explicado anteriormente.

[Aclaración]: en ninguno de los algoritmos cierro el camino, ya que el problema busca que los problemas sean resueltos en el menor tiempo posible, por lo que el tiempo que tardaría en volver el electricista después de haber terminado el último trabajo, no influye en el tiempo que ha tardado en realizar los anteriores.