2018

.

TRABAJO PRÁCTICO 4
PROGRAMACIÓN 3 - TUDAI

ALGORITMOS GREEDY – EJERCICIO 5

VÁZQUEZ RODRIGO

RODRIGOVAZQUEZ420@GMAIL.COM

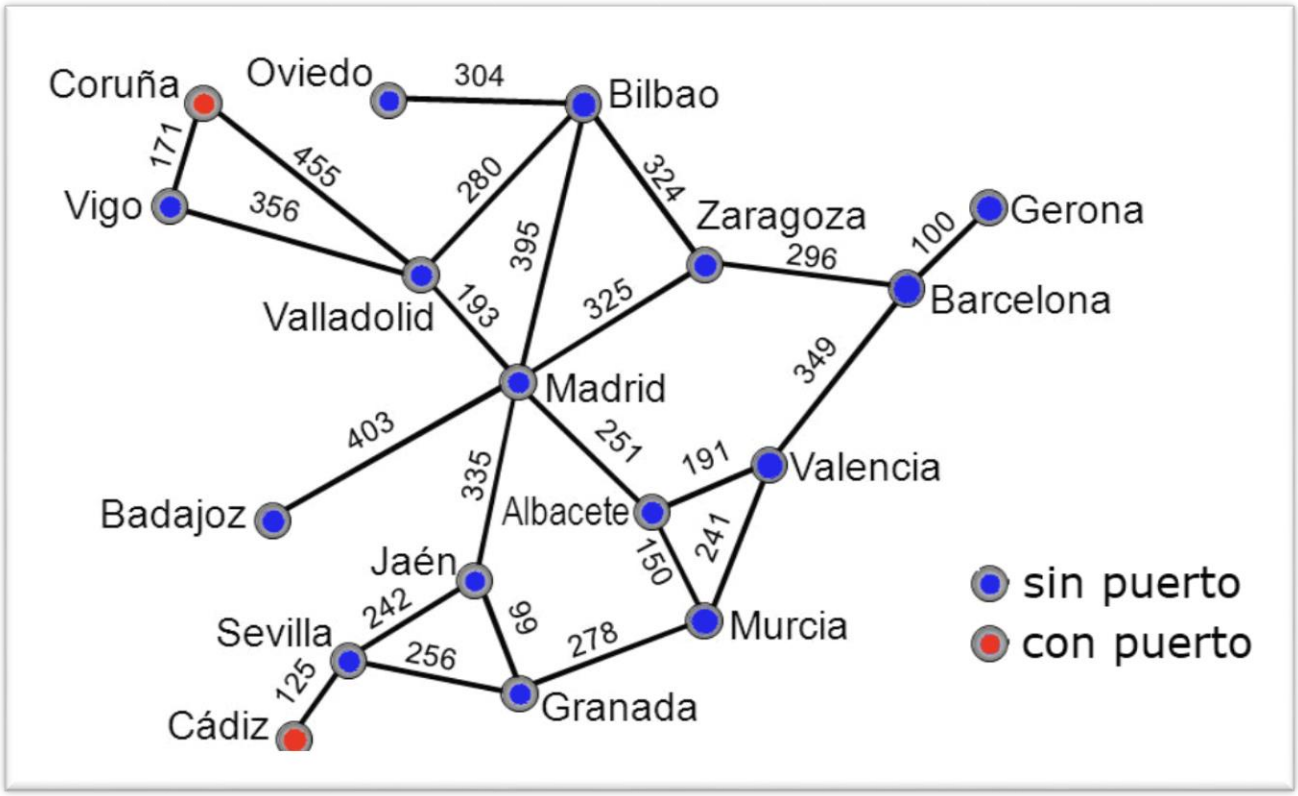
FACULTAD DE CIENCIAS
EXACTAS
UNIVERSIDAD NACIONAL DEL CENTRO
DE LA PROVINCIA DE BUENOS AIRES

Introducción

1 PROBLEMA A RESOLVER

"Desde un cierto conjunto grande de ciudades del interior de una provincia, se desean transportar cereales hasta alguno de los 3 puertos pertenecientes al litoral de la provincia. Se pretende efectuar el transporte total con mínimo costo sabiendo que el flete es más caro cuanta más distancia tiene que recorrer. Dé un algoritmo que resuelva este problema, devolviendo para cada ciudad el camino que debería recorrer hacia el puerto de menor costo."

En esta ocasión se nos pide entregar un pseudocódigo del algoritmo que resuelve el problema y un seguimiento completo del mismo con un grafo de ejemplo:



El pseudocódigo deberá contener el método principal del problema y las funciones ***SOLUCIÓN***, ***FACTIBLE***, ***SELECCIONAR***.

Desarrollo

2 PSEUDOCÓDIGO

```
function dijkstra (Grafo g, nodo_fuente n) {
  HashMap distancias;
  HashMap padres;
  HashMap vistos;
  ColaPrioridad cola;

  for each Vertice v in g.GetVertices {
    distancias[v] = INFINITO;
    padres[v] = NULL;
    vistos[v] = false;
  }
  distancias[n] = 0;
  cola.add(n, distancias[n]);

  While (!cola.isEmpty) {
    u = cola.getMinimo();
    vistos[u] = true;
    for each Vertice vecino in u.getVecinos() {
      if (vistos[vecino] == false) {
        if (distancia[vecino] > distancias[u] + peso (u, vecino)) {
          distancias[vecino] = distancias[u] + peso (u, vecino)
          padres[vecino] = u;
          adicionar(cola,(vecino, distancias[vecino]));
        }
      }
    }
  }
  return padres;
}
```

```
function obtenerPuertosCercanos(Grafo g) {
  puertos[] = g.getPuertos();
  ciudadesNoPortuarias = g.getCiudadesNoPortuarias();
  Hashmap (ciudad, camino) solucion;
  caminosCortosPuertos[];

  for each puertos as puerto {
    caminosCortosPuertos.push(dijkstra(g,puerto));
  }

  for each ciudadesNoPortuarias as ciudad {
    for each caminosCortosPuertos as camino {
      if (camino.getVertices().contains(ciudad)) { // factible
        if (camino.getPesoTo(ciudad)<solucion.get(ciudad).getPesoTo(ciudad)){
          solucion.put(ciudad, camino);
        }
      }
    }
  }
  return solucion; //solucion
}
```

3 SEGUIMIENTO

El programa comienza cuando el usuario invoca la función: “obtenerPuertosCercanos (Grafo g)” pasando como parámetro un grafo como el de la imagen mostrada más arriba:

```
function obtenerPuertosCercanos(Grafo g) {
  puertos[] = g.getPuertos();
  ciudadesNoPortuarias = g.getCiudadesNoPortuarias();
  Hashmap (ciudad, camino) solucion;
  caminosCortosPuertos[];

  for each puertos as puerto {
    caminosCortosPuertos.push(dijkstra(g,puerto));
  }

[…]

}
```

Este comienza por obtener todas las ciudades que son puertos, con ayuda de algún método (en este caso (`g.getPuertos()`) y obtener de cada uno de estos, con ayuda de la función `dijkstra(g,puerto)`, el caminos más cortos hacia cada nodo (El seguimiento de esta función se realizara más adelante).

```
for each ciudadesNoPortuarias as ciudad {
    for each caminosCortosPuertos as camino {
      if (camino.getVertices().contains(ciudad)) {
        if (camino.getPesoTo(ciudad)<solucion.get(ciudad).getPesoTo(ciudad)){
          solucion.put(ciudad, camino);
        }
      }
    }
  }
  return solucion;
}
```

Luego a cada ciudad que no sea un puerto, se procederá a buscarla dentro de cada uno de los caminos obtenidos. Si la ciudad no se encuentra quiere decir que no se puede llegar al puerto por ese camino, caso contrario, si la ciudad es encontrada se comparará el peso de este camino con el mejor hasta ese momento, y si llega a ser mejor, remplazarlo.

SEGUIMIENTO DE DIJKSTRA:

```
function dijkstra (Grafo g, nodo_fuente n) {
  HashMap distancias;
  HashMap padres;
  HashMap vistos;
  ColaPrioridad cola;

  for each Vertice v in g.GetVertices {
    distancias[v] = INFINITO;
    padres[v] = NULL;
    vistos[v] = false;
  }
[…]
}
```

Se comienza por setear cada vértice como no visitado, sin padre y con distancias infinitas:

DISTANCIAS	
Vértice	Distancia
Coruña	∞
Vigo	∞
Valladolid	∞

PADRES	
Vértice	Padre
Coruña	NULL
Vigo	NULL
Valladolid	NULL

VISTOS	
Vértice	Visitado
Coruña	false
Vigo	false
Valladolid	false

```
distancias[n] = 0;
  cola.add(n, distancias[n]);
```

Luego para el nodo inicial, (en este caso el puerto de **Coruña**) actualizamos su distancia a 0 y lo agregamos a una cola de prioridad (hasta este momento vacía).

DISTANCIAS	
Vértice	Distancia
Coruña	**0**
Vigo	∞
Valladolid	∞

PADRES	
Vértice	Padre
Coruña	NULL
Vigo	NULL
Valladolid	NULL

VISTOS	
Vértice	Visitado
Coruña	false
Vigo	false
Valladolid	false

COLA DE PRIORIDAD
Coruña

```
While (!cola.isEmpty) {
    u = cola.getMinimo();
    vistos[u] = true;
```

Ahora, mientras la cola no este vacía procederemos a obtener el valor mínimo de esta, marcarlo como visitado y guardarlo en la variable u, como el único valor de la cola es Coruña, este es vértice que se obtendrá :

DISTANCIAS	
Vértice	Distancia
Coruña	0
Vigo	∞
Valladolid	∞

PADRES	
Vértice	Padre
Coruña	NULL
Vigo	NULL
Valladolid	NULL

VISTOS	
Vértice	Visitado
Coruña	true
Vigo	false
Valladolid	false

COLA DE PRIORIDAD

```
    for each Vertice vecino in u.getVecinos() {
      if (vistos[vecino] == false) {
        if (distancia[vecino] > distancias[u] + peso (u, vecino)) {
          distancias[vecino] = distancias[u] + peso (u, vecino)
          padres[vecino] = u;
          adicionar(cola,(vecino, distancias[vecino]));
        }
      }
    }
```

En este momento se procederá a visitar los vecinos no visitados de u (Coruña) Vecinos: [Vigo, Valladolid] (supongamos que se eligió Vigo), y si la distancia de este vecino es mayor a la distancia de u + la distancia que hay entre ellos (∞ > 0 + 171) se procederá a actualizar la distancia del vecino por la nueva que es más corta, se le actualizará el padre y lo adicionaremos a la cola de prioridad:

DISTANCIAS	
Vértice	Distancia
Coruña	0
Vigo	171
Valladolid	∞

PADRES	
Vértice	Padre
Coruña	NULL
Vigo	Coruña
Valladolid	NULL

VISTOS	
Vértice	Visitado
Coruña	true
Vigo	false
Valladolid	false

COLA DE PRIORIDAD
Vigo

Luego, para el siguiente vecino Valladolid lo mismo:

DISTANCIAS	
Vértice	Distancia
Coruña	0
Vigo	171
Valladolid	455

PADRES	
Vértice	Padre
Coruña	NULL
Vigo	Coruña
Valladolid	Coruña

VISTOS	
Vértice	Visitado
Coruña	true
Vigo	false
Valladolid	false

COLA DE PRIORIDAD	
Vigo	Valladolid

Luego de recorrer todos los vecinos, la cola de prioridad tiene nuevos valores por lo que el ciclo vuelve a comenzar:

```
While (!cola.isEmpty) {
    u = cola.getMinimo();
    vistos[u] = true;
```

el nuevo mínimo u será en este caso será Vigo, se lo marca como visitado y así continuará el ciclo hasta haber visitado todos los Vértices:

DISTANCIAS	
Vértice	Distancia
Coruña	0
Vigo	171
Valladolid	455

PADRES	
Vértice	Padre
Coruña	NULL
Vigo	Coruña
Valladolid	Coruña

VISTOS	
Vértice	Visitado
Coruña	true
Vigo	true
Valladolid	false

COLA DE PRIORIDAD
Valladolid

Al finalizar el ciclo, se procederá a retornar el HashMap de padres

```
    return padres;
}
```