Programación 3: Entregable Nro 3

Algoritmos Greedy

Carton Nelson (nelsoncarton@gmail.com**)**

Problema a resolver

Se pide realizar en pseudocódigo el ejercicio 5 del práctico 4 de algoritmos greedy, el cual pide que se encuentre el camino de menor costo(menor cantidad de KM) desde una ciudad dada hasta una que posea puerto, una vez obtenido el resultado hacia cada puerto se debe calcular a cuál de esos puertos se encuentra a menor distancia de la ciudad origen. Se utilizó un grafo provisto por la cátedra para llevar a cabo el ejercicio. Para esto se implementó el algoritmo de Dijkstra, el cual recorriendo el grafo se va quedando con el camino más corto analizando lo adyacentes de cada nodo, si bien esto no asegura obtener el camino mas optimo, se puede llegar a un resultado cercano al esperado.

Como inicio se implementa el procedimiento **Dijkstra** el cual va a devolver un conjunto de vértices que va a permitir trazar el recorrido que dé como resultado.

Dicho procedimiento va a recibir como parámetros:
Vértice **inicio** (ciudad inicial)
Matriz **M** (matriz de adyacencia)
Conjunto <Vértices> **Q** (conjunto de candidatos/vértices)
Conjunto <Vértices> finales(conjunto de puertos)

La matriz **M** (matriz de adyacencia) posee los pesos de cada Vértice hacia sus adyacentes como por ejemplo

	vértice A	vérticeB	vérticeC	…...	vérticeN
vérticeA		4	3	…..	-
vérticeB	1	-	-	…..	6
vérticeC	4	1	-	…..	-
	….	….	…..	…...	-
vérticeN	5	1	-	…...	-

```
Conjunto <Vértices> Dijkstra (Conjunto <Vértices> final, Conjunto <Vértices> Q, Vértice inicio, Matriz M){

    arr temp[ ]                    // arreglo de pesos temporales de cada vértice
    arr sol [ ]                        //conjunto solución inicializado en vacío
    arr padres [ ]                 //arreglo de padres el cual va a retornar este método
        para (cada Vértice V){
            temp[V]=∞                  //inicializo arr de temporales en infinito
            padres[V]=null             //inicializo arr de padres en nulo
        }

    temp [inicio]=0                //al ser el nodo inicial seteo el costo temporal de inicio en 0

    mientras (Q!= vacio && !solución(sol,final)){

        Vértice mínimo = seleccionar (temp, q) //elijo el más pequeño de los temporales
        q.borrar(mínimo)           //lo elimino del conjunto de candidatos

        si (factible(sol,min)){    //chequeo posible solución y si lo es lo agrego al arr sol
            sol.add[min]           //lo agrego al conjunto solución
            para (cada Vértice V de adyacentes a mínimo){
                si(V ∉ sol){
                    si (temp[V]> temp[mínimo]+M [mínimo][V]){ //seteo temp siempre que el temp de
                    temp[V]= temp[mínimo]+M [mínimo][V] //V sea mayor que el temp de V sumado
                    padres[V]=mínimo //al temp del mínimo mas el costo de mínimo a ese V
                                     // usando la matriz de ady, luego lo agrego a mínimo como
                                     //padre

                    }
                }
            }
        }
    }

    si(solución (sol,final)){
        retornar (padres[]); //retorna el arr de padres para luego poder recrear
    }sino{                                  //el camino y su costo hacia cada

        return padres[null]        //no existe camino
        }


}//final de dikjstra
```

Implementacion del metodo **seleccionar,** el cual recibe como parámetro el conjunto de pesos temporales (para elegir el vértice con menor peso temp), el conjunto de candidatos y retorna un vértice.

```
Vértice seleccionar (Conjunto <Vértices> temp, Conjunto<Vértices> Q){
        Vértice mínimo= ∞
        for(i=o, i<temp.size){
                si(temp.get[i]<minimo && temp.get[i]  ∈ Q){ //chequeo que el vértice temporal sea menor a
                                                                //mínimo y este en el conjunto de candidatos
                        minimo=temp.get[i]
                }
        }
        retorno mínimo
  }
```

implementación del método **factible**, el cual recibe como parámetro el conjunto solución y el vértice a chequear. Este determina si el vértice a analizar cumple con los requisitos para su utilización.

```
boolean factible (conjunto<Vértices> sol, Vértice V){

   retorno (V.getVertices().size()>0 && !sol.contains(V)) // chequeo que el vértice posea adyacentes y que,
                                                          // a su vez, no esté incluido en solución

}
```

implementación del método **solución**, el cual recibe como parámetros el conjunto solución y el conj de vértices final para determinar si dichos puertos están en solución.

```
boolean solucion (Conjunto <Vértices> sol,  Conjunto <Vértices> final){
result=false
        para (cada V del conj final){
                si sol.contains(v){
                result=true
                }sino{
                return false
                  }

        return (result)
}
```

Una vez implementado **dijkstra** y obtenido el arr de padres, agrego el método **seleccionarPuerto** para encontrar el mejor puerto, es decir el de menor distancia.

```
Vértice seleccionarPuerto (Conjunto<vértices> puertos, Conjunto <vértices> padres,Vértice inicio){

        Vértice mejorPuerto=null
        int aux= ∞

        por( cada puerto p){
                si (getKilometros(puertos[p],inicio,padres)<aux){
                        aux=(getKilometros(puertos[p],inicio,padres))//analizo los km hacia cada
                                                                     //puerto y voy guardando el menor
                        mejorPuerto=puertos[p]
                }
        }
 return mejorPuerto
 }
```

En los siguientes métodos, **getCamino** y **getKilometros,** implemento el trazado de la ruta y cuanto cuesta la misma.

```
int getKilometros (Vértice final,Vértice inicio,padres[]){
        pila<vértices> p //pila que va a poseer el recorrido
        int aux=0
        Vértice V=null
        p=getCamino(final,padres,inicio)       //trazo el recorrida
        mientras(p!=vacío){
                 v=p.desapilar();  //tomo el tope de la pila
                aux=v.getPeso(p.tope)          //sumo el peso del vértice desapilado con el
                                               //siguiente (tope de la pila)
        }

return aux //retorna el kilometraje total del camino
}
```

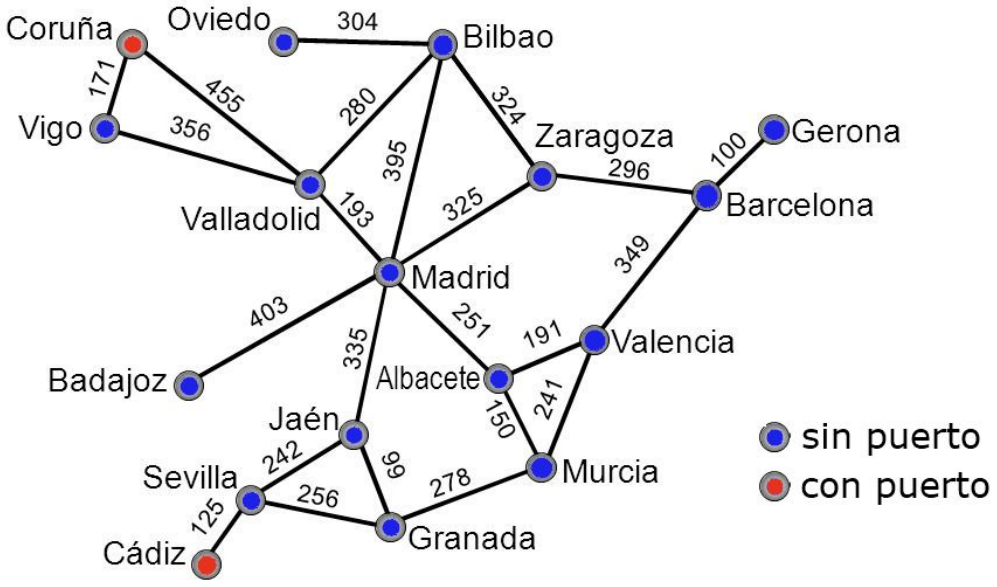
```
Pila<vértices> getCamino(Vértice final,Padres[],Vértice inicio){
Vértice index=final
Pila<vértices> result=vacío
result.agregar(final)
        mientras(padres[index]!=inicio)
        result.agregar(padres[index]) //utilizando el arr de padres voy recorriendo en reversa
                                      //encontrando los antecesores y llenando la pila result

        index=padres[index]
        }

retorno(result)
}
```

Utilización del código en un ejemplo concreto:

para entender mejor el código paso a explicar cómo se aplicaría con el grafo de ciudades provisto por la cátedra.



Primero seteo el arreglo de distancias temporales (*temp[ ]*) en infinito, el arreglo de padres (*padres[ ]*) en null y el arreglo solución en vacío (*sol[ ]*).
el algoritmo recibirá un conjunto donde se encuentran todos los nodos pertenecientes al grafo, una lista de ciudades portuarias, la matriz de adyacencia y a su vez un nodo inicio con el cual va a calcular el camino más conveniente desde dicho inicio hacia los demás vértices según dijkstra.

tomamos un nodo inicio, como por ejemplo *Zaragoza*, el primero paso es poner el peso temporal del arreglo *temp[ ]* en donde se encuentra el inicio en 0 ya que va a ser el punto de partida.

Nodo	*Temp []*	*Padres []*	*candidatos*	*Solución []*
Zaragoza	0	NULL	X	
Barcelona	∞	NULL	x	
Madrid	∞	NULL	x	
Bilbao	∞	NULL	x	
.....	∞	NULL	X	

Luego mientras el conjunto de candidatos no este vacio y no exista solución, selecciono el vértice con el peso temporal más pequeño(Zaragoza) y lo elimino de *candidatos*. Si el Vértice elegido es factible para una posible solución se agrega al conjunto *solución[ ]* y se analizan cada uno de sus adyacentes.

Empezando por Barcelona, si el temporal de dicho adyacente es mayor al temporal de Zaragoza **(0)** sumado a el costo del camino de Zaragoza a Barcelona **(296 consultando la matriz de adyacencia)** se reemplazará en el arreglo *temp[ ]* y se modifica el arreglo *padres[ ],* en el índice donde se encuentre Barcelona, de la siguiente manera:

Nodo	*Temp []*	*Padres []*	*candidatos*	*Solución []*
Zaragoza	0	NULL		x
Barcelona	296	Zaragoza	x	
Madrid	∞	NULL	x	
Bilbao	∞	NULL	x	
.....	∞	NULL	X	

Repitiendo el mismo paso para cada uno de los adyacentes que restan:

Nodo	*Temp []*	*Padres []*	*candidatos*	*Solución []*
Zaragoza	0	NULL		x
Barcelona	296	Zaragoza	x	
Madrid	325	Zaragoza	x	
Bilbao	324	Zaragoza	x	
.....	∞	NULL	X	

Una vez que se cumple el análisis a cada vértice adyacente a Zaragoza se procede a buscar el nodo con el peso temporal más pequeño en el arreglo *temp[ ],* Barcelona en este caso, si cumple con los requisitos para el análisis se lo elimina de *candidatos* se lo agrega a *solución[ ]* y se vuelve a repetir el ciclo. Todo esto hasta terminar con todos los vértices del conjunto de candidatos.

Luego de finalizar todo el proceso se va a retornar el arreglo *padres[ ]* a partir del cual voy a recrear el camino más conveniente, según el algoritmo, hacia una ciudad portuaria y calculando las distancias ver cual de ellas es la más conveniente.