Programación 3: Entregable Nro 3
Algoritmos Greedy

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

Se pide realizar 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 Vertice hacia sus adyacentes

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

```
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.

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

        Vertice mejorPuerto=null
        int aux=0

        por( cada puerto p){
                si (getKilometros(puertos[p],padres)<aux){
                        aux=(getKilometros(puertos[p],padres,inicio))
                        mejorPuerto=puertos[p]
                }
        }
  return mejorPuerto
  }
```

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

```
Pila getCamino(Vértice final,Padres[],Vértice inicio){
Vértice index=null
Pila result=vacío
        mientras(padres[index]!=inicio)
        result.agregar(padres[index]) //utilizando el arr de padres voy recorriendo y llenando la pila
                                      //result
                index=padres[index]
        }
        result.agregar(final)
retorno(result)
}
```

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

        }

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