Pseudo Código

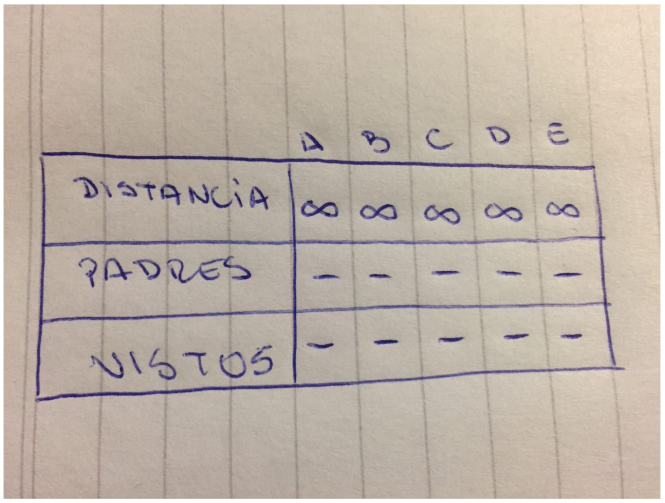
```
function HashMap<Nodo,Nodo> getPuertos (Grafo g){
  ArrayList <Nodo> ciudades = g.getCiudades()
  ArrayList <Nodo> puertos = g.getCiudadesPuerto()
  HashMap <Nodo,Nodo> ret
  ArrayList<HashMap <Nodo,Nodo>> soluciones
  for puertos as puerto{
    soluciones.push(calcularCaminos(g,puerto))
  }
  for ciudades as ciudad {
    for soluciones as solu{
      if (solu.keySet().contains(ciudad)){
        if (solu.calcularPesoA(ciudad) < ret.get(ciudad).calcularPesoA(ciudad)){
          ret.put(ciudad,solu)
        }
      }
    }
  }
  return ret
}

function HashMap<Nodo,Nodo> calcularCaminos(Grafo g,Node n){
  HashMap <Nodo,Integer> distancias
  HashMap <Nodo,Nodo> padres
  Hashmap <Nodo,boolean> EstadoVisita
  Queue cola
  for Nodo actual in g.getVertices(){
    distancias.put(actual) = MaxValue
    padres.put(actual) = null
    vistos.put(actual) = false
  }
  distancias.put(n) = 0
  cola.add(n,distancias.get(n))
while !cola.isEmpty(){
    Nodo actual= cola.getMinimo()
    EstadoVisita.put(Actual) = true
    for Vertice vecino in actual.getVecinos(){
      if(EstadoVisita.get()){
        if(distancias.get(actual) > distancias.get(vecino)+getPesoFromTo(actual,vecino)){
          distancias.get(actual) = distancias.get(vecino)+getPesoFromTo(actual,vecino)
          padres.put(vecino) = actual
          cola.add(vecino)
        }
      }
    }
```

```
  }
  return padres
}
```

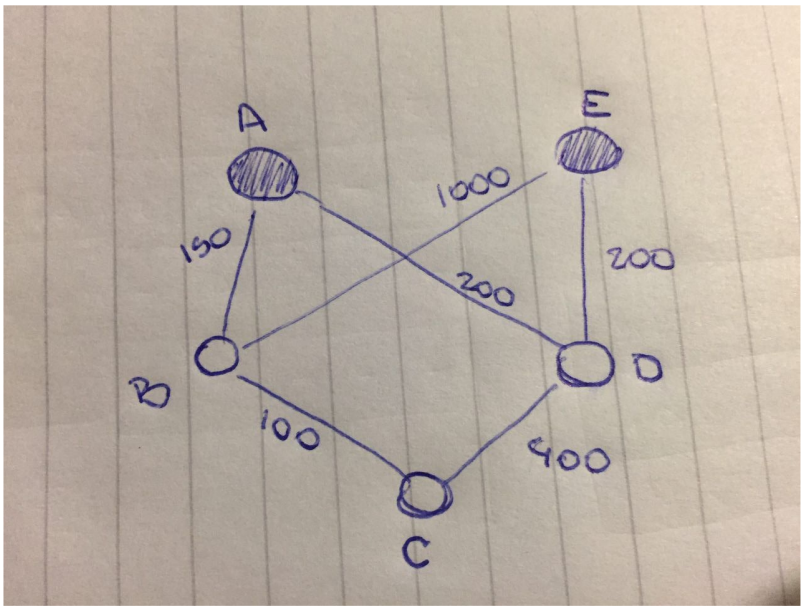
Seguimiento de la función

Partiendo de un grafo:



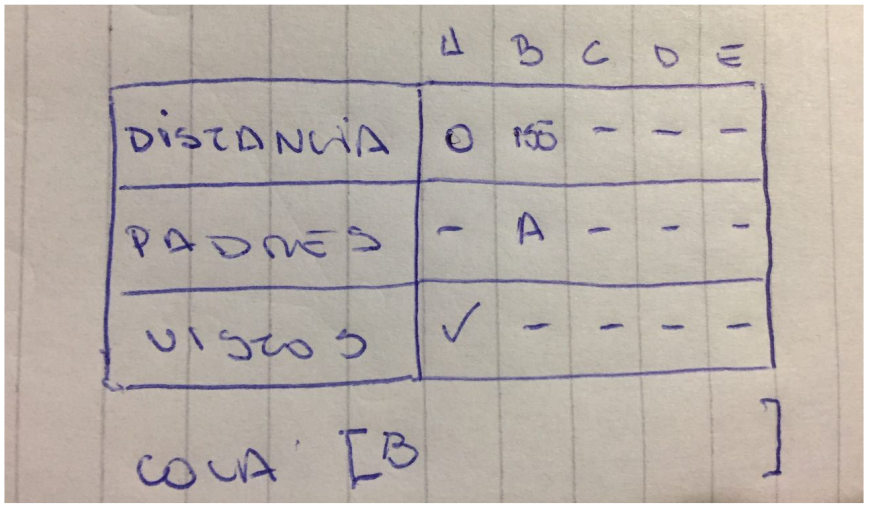
El programa comienza ejecutando la función **getPuertos(grafo)** y pasando como parámetro el grafo anterior.
Dicho método genera 2 arreglos de nodos. Uno con todos los nodos que tienen puerto, usando **getCiudadesPuerto** y otro con los que no, usando **getCiudades**.
Luego se recorrerá todos los nodos que no tengan puerto, y por cada uno se analizará si el camino que la variable solución tiene actualmente, es el óptimo.

Estos valores se consiguen usando el método **calcularCaminos** que recibe un grafo y un nodo destino. Cuando se inicia el método, se setean 3 **HashMaps**: Uno con distancias, que se inicializa con los valores más altos posibles en java (esto es para q a la hora de comprar con el primer valor, siempre sea el mínimo). Otro **HashMap** se usa para los padres. Este se usa para guardar el nodo por el que se llega al nodo que estoy actualizando. Es por eso que se inicializa en null. Por último el **EstadoVisita**, que guarda el nodo, como visitado o no.
Al final quedaría algo como esto:

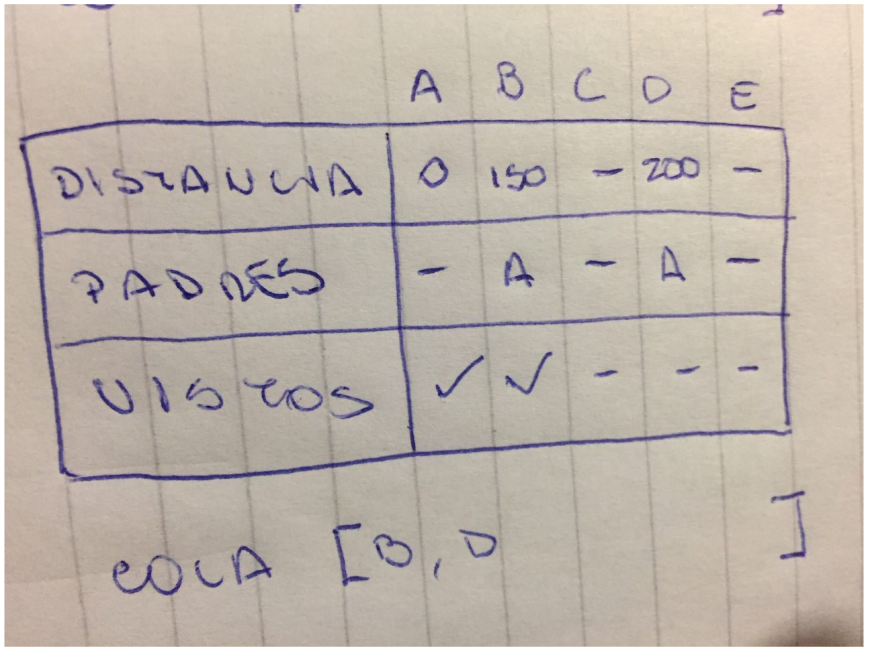


Seguido de eso, se llama al primer nodo, que es A. La distancia del mismo se setea en 0, y no tiene ningún nodo del que precede.
Después se agrega el nodo a la cola.

Luego, se hace un loop mientras la "cola" no este vacia. En este caso tiene al nodo A, asi que comienza.
Primero saca el nodo actual de la cola, y lo marca como visitado.
Luego busca todos sus adyacentes que no estén visitados, y por cada uno pregunta, si la distancia para llegar a él (la distancia por la que se llegó a A + la distancia que hay de A al destino) es menor que la que tiene ese nodo. En este caso preguntaría "(0 + 150)<MAX_NUM". Cómo retorna que si, se agrega esa suma a la distancia mínima actual de B. También cambia el padre de B y le agrega "A".
Antes de pasar al siguiente nodo, agrega B a la cola.
Quedaria asi:



Sigue con el siguiente adyacente de A. La suma para llegar a D es minimo, asi q lo setea con 200. Setea su padre, también con A, y por último agrega D a la cola.



Cuando termina de visitar todos los adyacentes de A, vuelve a entrar al loop, pero con el próximo nodo de la cola. El ciclo continuará recursivamente, hasta que se visiten todos los nodos. De este modo, la tabla quedará cargada con las distancias mínimas a cada nodo, y por cuales hay que pasar para llegar al mismo.
Si en algún momento se encuentra un valor más chico del que está en ese momento en el nodo que se está analizando, se reemplaza este valor, y se cambia el padre.

Al final del método, quedaría la siguiente tabla.