GRAFOS

TABLE OF CONTENTS

DFS:

Proposito: Transversar el grafo por profundidad, recorriendo los distintos caminos comenzando por el mas largo.

Precauciones: ...

Complejidad Computacional:

O(A+N), donde A → Aristas ^ N → Nodos

Requisitos:

- Boolean de nodos visitados
- Stack para recorrer un camino

Pseudo-codigo:

```
DFS(grafo, root_node):
      visitado[root_node] = true
      stack.push(root_node)

      while !stack.isEmpty():
            v = stack.pop()
            for w in vecinos(v):
                  if !visitado[w]:
                        visitado[w] = true
                        stack.push(w)
```

BFS:

Proposito: Transversar el grafo por saltos, en cada iteracion recorremos n caminos.

Precauciones: ...

Complejidad Computacional:

O(A+N), donde A → Aristas ^ N → Nodos

Requisitos:

- Boolean de nodos visitados
- Queue para recorrer un camino

Pseudo-codigo:

```
BFS(grafo, root_node):
      visitado[root_node] = true
      queue.push(root_node)

      while !queue.isEmpty():
            v = queue.pop()
            for w in vecinos(v):
                  if !visitado[w]:
                        visitado[w] = true
                        queue.push(w)
```

PRIM:

Proposito: Encontrar un arbol que conecte a todos los nodos con el menor peso posible, partiendo de un nodo arbitrario.

Precauciones: ...

Complejidad Computacional:

O((A+N) * log(N)), donde A → Aristas ^ N → Nodos

Requisitos:

- Lista de costos para cada nodo
- Lista de nodos visitados

Pseudo-codigo:

```
prim(grafo, root_node):
      for nodos in grafo:
            costos[nodo] = MAX_VALUE
            visitados[nodo] = false

      costos[root_node] = 0
      queue.push(root_node,costos[root_node])

      while !queue.isEmpty:
            current_arista = queue.poll()
            current_nodo = current_arista.get_child()
            visitados[current_node] = true

            if current_arista.costo < costos[current_nodo]:
                  costos[current_nodo] = current_arista.costo
                  grafo_resultante.add(current_arista)

            for arista in vecinos(current_node):
                  if !visitados[arista]:
                        queue.push(arista,costos[root_node])

      return  grafo_resultante
```

KRUSKAL:

Proposito: Encontrar un arbol que conecte a todos los nodos con el menor peso posible.

Precauciones: ...

Complejidad Computacional:

O((A+N) * log(N)), donde A → Aristas ^ N → Nodos

Requisitos:

- Lista de aristas ordenadas por deso ascendente
- UnionFind set

UnionFind pseudo-codigo:

```
Propiedades:
      int id
      int rank
      UnionFind parent

UnionFind(int id):
      this.id = id
      this.rankk = 0
      this.parent = this

find():
      x = this
      while x.parent != x:
            x.parent = x.parent.paren
            x = x.parent
      return x
union(UnionFind y):
      x = this.find()
      if x == y:
            return
      if y.rank > x.rank:
            temp = x
            x = y
            y = temp

      y.parent = x
      if x.rank == y.rank:
            x.rank++
```

Pseudo-codigo:

```
kruskal(grafo):
      for nodo in grafo:
            set[nodo] = new UnionFind(nodo)
            lista_ordenada.addAll(nodo.getAristas())

      lista_ordenada.sort()

      for arista in lista_ordenada:
            if set[arista.desde].find() != set[arista.hasta].find():
                  set[arista.desde].union(set[arista.hasta)
                  grafo_resultante.add(arista)

      return  grafo_resultante
```

Dijkstra:

Proposito: Buscar el camino mas corto de un nodo a todos los otros nodos del grafo.

Precauciones: Este algoritmo no funciona con grafos que contengan pesos negativos.

Complejidad Computacional:

O((A+N) * log(N)), donde A → Aristas ^ N → Nodos

Requisitos:

- Lista de antecesores
- Lista de costos para cada nodo
- Lista de nodos visitados
- Cola de prioridad

Pseudo-codigo:

```
antecesores[] = null
dijkstra(grafo, root_node):
      for nodos in grafo:
            costos[nodo] = MAX_VALUE
            visitados[nodo] = false
            antecesores[nodo] = root_node

      costos[root_node] = 0
      queue.push(root_node,costos[root_node])

      while !queue.isEmpty:
            current_arista = queue.poll()
            current_nodo = current_arista.get_parent()
            visitados[current_node] = true

            for arista in vecinos(current_node):
                  if !visitados[arista]:
                        costo = arista.costo() + current_nodo.costo()
                        if costo < costos[arista]:
                              queue.push(arista,costos[root_node])
                              antecesores[arista] = current_nodo
                              costos[arista] = costo
      return costos
```

Pseudo-codigo para encontrar el camino mas corto hasta algun nodo:

```
caminoHasta(desde, hasta):
      stack.push(hasta)
      while antecesores[hasta] != desde:
            stack.push(antecesores[hasta)
            hasta = antecesores[hasta]
      stack.push(desde)

      while !stack.isEmpty():
            lista.add(stack.pop)
      return lista
```

Floyd-Warshall:

Proposito - Floyd: Buscar el camino mas corto de todos los nodo hasta todos los otros nodos de un grafo direccionado.

Proposito - Warhsall: Busca encontrar si existe algun camino entre distintos nodos.

Precauciones: Este algoritmo no funciona con grafos que contengan ciclos negativos.

Complejidad Computacional:

O(N^3), donde N → Nodos

Requisitos: ...

Pseudo-codigo:

```
floyd(grafo):
      for i in grafo:
            for j in grafo:
                  if grafo.getArista(i,j): //Si existe relacion
                        resultado[i][j] = grafo.getArista(i,j)
                  else: resultado[i][j] = MAX_VALUE

            resultado[i][i] = 0

      for k in grafo:
            for i in grafo:
                  for j in grafo:
                        if resultado[i][k] + resultado[k][j] < resultado[i][j]:
                              resultado[i][j] =  resultado[i][k] + resultado[k][j]

      return resultado

warshall(grafo):
      for i in grafo:
            for j in grafo:
                  result[i][j] = (grafo.getArista(i,j) != null)

      for k in grafo:
            for i in grafo:
                  for j in grafo:
                    resultado[i][j] = (result[i][j] || (result[i][k] && result[k][j]))

      return resultado
```

Coloreo:

Número cromático: Número minimo necesario para pintar un grafo.

Grado: Cantidad de aristas conectadas con el nodo.

Algortimos de Orden:

- Secuencial: Pinta el grafo partiendo desde el primer nodo.
- Welsh Powell: Pinta el grafo a partir del nodo con mayor grado.
- Matula: Pinta el grafo a partir del nodo con menor grado.
- Por color: Luego de pintar un nodo, en base de alguno de los otros ordenes, pinta todos los nodos que no esten relacionados con este.