US301 – Análise de Complexidade

```
  public static <V, E> boolean isGraphConnected(Graph<V, E> g) {
        if (g.numVertices() == 0)
            return false;
        boolean[] visited = new boolean[g.numVertices()];
        LinkedList<V> qdfs = new LinkedList<>();
        DepthFirstSearch(g, g.vertices().iterator().next(), visited, qdfs);
        for (int i = 0; i < g.numVertices(); i++) {
            if (!visited[i])
                return false;
        }
        return true;
    }
```

```
    private static <V, E> void DepthFirstSearch(Graph<V, E> g, V vOrig,
boolean[] visited, LinkedList<V> qdfs) {
        if (visited[g.key(vOrig)])
            return;

        qdfs.add(vOrig);
        visited[g.key(vOrig)] = true;
        for (V vAdj : g.adjVertices(vOrig))
            DepthFirstSearch(g, vAdj, visited, qdfs);
    }
```

Linhas	Código	Complexidade

1-5	if (g.numVertices() == 0)
	return false;
	boolean[] visited = new boolean[g.numVertices()];
	LinkedList<V> qdfs = new LinkedList<>();
		1

6	DepthFirstSearch(g, g.vertices().iterator().next(), visited, qdfs);
	boolean[] visited, LinkedList<V> qdfs) {
	if (visited[g.key(vOrig)])
	return;
	qdfs.add(vOrig);
	visited[g.key(vOrig)] = true;
	for (V vAdj : g.adjVertices(vOrig))
	DepthFirstSearch(g, vAdj, visited, qdfs);
		O(V+E) onde V é o número de vertíces e o número de arestas Uma vez que visitamos cada vertice e cada aresta uma vez, visitamos um vertice para o marcar como visitado e para isso precisamos de visitar uma aresta.

7-9	for (int i = 0; i < g.numVertices(); i++) {
	if (!visited[i])
	return false;}
		O(V x (V+E)), percorre-se todos os vertices a confirmar de estes foram verficados.

```
    public static <V, E> int CountMinPath(Graph<V, E> g, V vOrigin, V vDest,
Comparator<E> ce, BinaryOperator<E> sum,
            E zero) {
        LinkedList<V> paths = new LinkedList<>();
        shortestPath(g, vOrigin, vDest, ce, sum, zero, paths);
        return paths.size() - 1;
    }
```

```
    public static <V, E> E shortestPath(Graph<V, E> g, V vOrig, V vDest,
            Comparator<E> ce, BinaryOperator<E> sum, E zero,
            LinkedList<V> shortPath) {
        if (!g.validVertex(vOrig) || !g.validVertex(vDest))
            return null;

        shortPath.clear();
        int numVerts = g.numVertices();
        boolean[] visited = new boolean[numVerts];
        @SuppressWarnings("unchecked")
        V[] pathKeys = (V[]) new Object[numVerts];
        @SuppressWarnings("unchecked")
        E[] dist = (E[]) new Object[numVerts];
        for (int i = 0; i < numVerts; i++) {
            dist[i] = null;
            pathKeys[i] = null;
        }
        shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, pathKeys,
dist);
        E lengthPath = dist[g.key(vDest)];
        if (lengthPath == null)
            return null;
       getPath(g, vOrig, vDest, pathKeys, shortPath);

        return lengthPath;
    }
```

```
    private static <V, E> void shortestPathDijkstra(Graph<V, E> g, V vOrig,
            Comparator<E> ce, BinaryOperator<E> sum, E zero,
            boolean[] visited, V[] pathKeys, E[] dist) {

        int vKey = g.key(vOrig);
        dist[vKey] = zero;
        pathKeys[vKey] = vOrig;
        int vKeyAdj;
        E minDist;

        while (vOrig != null) {
            vKey = g.key(vOrig);
```

```
            visited[vKey] = true;
            for (Edge<V, E> edge : g.outgoingEdges(vOrig)) {
                vKeyAdj = g.key(edge.getVDest());
                if (!visited[vKeyAdj]) {
                    E s = sum.apply(dist[vKey], edge.getWeight());
                    if (dist[vKeyAdj] == null || ce.compare(dist[vKeyAdj], s)
> 0) {
                        dist[vKeyAdj] = s;
                        pathKeys[vKeyAdj] = vOrig;
                    }
                }
            }
            minDist = null;
            vOrig = null;
            for (V vert : g.vertices()) {
                if (!visited[g.key(vert)] && dist[g.key(vert)] != null
                        && (minDist == null || ce.compare(dist[g.key(vert)],
minDist) < 0)) {
                    minDist = dist[g.key(vert)];
                    vOrig = vert;
                }
            }
        }
    }
```

Linhas	Código	Complexidade
1-2	LinkedList<V> paths = new LinkedList<>(); shortestPath(g, vOrigin, vDest, ce, sum, zero,paths);	1

2-11	if (!g.validVertex(vOrig) ||
	!g.validVertex(vDest))
	return null;
	shortPath.clear();
	int numVerts = g.numVertices();
	boolean[] visited = new boolean[numVerts];
	@SuppressWarnings("unchecked")
	V[] pathKeys = (V[]) new Object[numVerts];
	@SuppressWarnings("unchecked")
	E[] dist = (E[]) new Object[numVerts];
		1

12-15	for (int i = 0; i < numVerts; i++) {
	dist[i] = null;
	pathKeys[i] = null;}
		O(V) uma vez que executa para a quantidade de vertíces.

16	shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, pathKeys, dist);	1

17-25(dentro do metodo shortestPathDijkstra)	int vKey = g.key(vOrig);
	dist[vKey] = zero;
	pathKeys[vKey] = vOrig;
	int vKeyAdj;
	E minDist;
		1

	while (vOrig != null) {
	vKey = g.key(vOrig);
	visited[vKey] = true;	

26-35(dentro do metodo shortestPathDijkstra)	for (Edge<V, E> edge : g.outgoingEdges(vOrig)) {
	vKeyAdj = g.key(edge.getVDest());
	if (!visited[vKeyAdj]) {
	E s = sum.apply(dist[vKey], edge.getWeight());
	if (dist[vKeyAdj] == null || ce.compare(dist[vKeyAdj], s) > 0) {
	dist[vKeyAdj] = s;
	pathKeys[vKeyAdj] = vOrig;
	}
	}
	}
		O(V+E), uma vez que percorre todas as arestas.

36-37(dentro do metodo shortestPathDijkstra)	minDist = null; vOrig = null;	1

38-45(dentro do metodo shortestPathDijkstra)	for (V vert : g.vertices()) {
	if (!visited[g.key(vert)] && dist[g.key(vert)] != null
	&& (minDist == null || ce.compare(dist[g.key(vert)], minDist) < 0)) {
	minDist = dist[g.key(vert)];
	vOrig = vert;
	}
	}
	}
		O(V^2+E), uma vez que percorre todos os verticies faz confirmações e consoante estas atribiu a minDist a distância

46-51	E lengthPath = dist[g.key(vDest)];
	if (lengthPath == null)
	return null;
	getPath(g, vOrig, vDest, pathKeys, shortPath);
	return lengthPath;
		1, uma vez que getPath apresenta também uma complexidade de 1.