# US305 – Análise de Complexidade

public static <V, E> Graph<V, E> kruskal(Graph<V, E> g) {

        Graph<V, E> mst;

        if (g instanceof MapGraph<V, E>) {

            mst = new MapGraph<>((!g.isDirected() && isGraphConnected(g)));

        } else if (g instanceof MatrixGraph<V, E>) {

            mst = new MatrixGraph<>((!g.isDirected() && isGraphConnected(g)));

        } else

            return null;

        LinkedList<V> connectedVerts = new LinkedList<>();

        for (int i = 0; i < g.numVertices(); i++) {

            mst.addVertex(g.vertex(i));

        }

        List<Edge<V, E>> lst = new ArrayList<>(g.edges());

        lst.subList(0, lst.size()).sort(new Comparator<Edge<V, E>>() {

            @Override

            public int compare(Edge<V, E> o1, Edge<V, E> o2) {

                return ((Comparable<E>) o1.getWeight()).compareTo(o2.getWeight());

            }

        });

        for (Edge<V, E> e : lst) {

            connectedVerts = DepthFirstSearch(mst, e.getVOrig());

            if (!connectedVerts.contains(e.getVDest()))

                mst.addEdge(e.getVOrig(), e.getVDest(), e.getWeight());

        }

        // System.out.println(mst);

        return mst;

    }

|  |  |  |
| --- | --- | --- |
| Linhas | Código | Complexidade |
| 1-7 | if (g instanceof MapGraph<V, E>) {              mst = new MapGraph<>((!g.isDirected() && isGraphConnected(g)));          } else if (g instanceof MatrixGraph<V, E>) {              mst = new MatrixGraph<>((!g.isDirected() && isGraphConnected(g)));          } else              return null;          LinkedList<V> connectedVerts = new LinkedList<>(); | 1 |
| 8-11 | for (int i = 0; i < g.numVertices(); i++) {              mst.addVertex(g.vertex(i));          } | O(V) onde V é o número de vertíces,uma vez que adicionamos todos os vertices a mts. |
| 12-18 | List<Edge<V, E>> lst = new ArrayList<>(g.edges());          lst.subList(0, lst.size()).sort(new Comparator<Edge<V, E>>() {              @Override              public int compare(Edge<V, E> o1, Edge<V, E> o2) {                  return ((Comparable<E>) o1.getWeight()).compareTo(o2.getWeight());              }          }); | 1 |
| 19-24 | for (Edge<V, E> e : lst) {              connectedVerts = DepthFirstSearch(mst, e.getVOrig());              if (!connectedVerts.contains(e.getVDest()))                  mst.addEdge(e.getVOrig(), e.getVDest(), e.getWeight());          } | O(V+E) |
| 22(DEEP FIRST SEARCH) | 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 |

O Complexo do algoritmo é O(V+ E(V+E))