Análise - Tema III

A seguir, iremos demonstrar trechos do código que sejam referentes ao **Tema III.** Com isto, temos o objetivo de enfatizar a complexidade dos algoritmos de Kruskal e Prim e seus dependentes. Contudo, vale salientar que avaliamos os trechos pertinentes ao propósito do tema, descartando partes que não importam, como a interface gráfica.

1. Kruskal

```
24    public static List<Edge> exec(Graph graph) {
25        if(graph.isDirected() || !graph.isGraphConnected() || !graph.isWeighted())
26            throw new RuntimeException("O grafo informado deve ser não dirigido, ponderado e c
27
28        List<Edge> result = new ArrayList<>(graph.maxVertices());//A = Ø
29
30        DisjointSet cd = new DisjointSet(graph.maxVertices());
31
32        for(Vertex vertex: graph.vertices()) { //for each vertex v ∈ G.V
33            cd.makeSet(vertex.index()); //MAKE-SET(v)
34        }
35
36        //sort the edges of G.E into nondecreasing order by weight w
37        List<Edge> sortedEdgeList = graph.edgesWithList();
38        sortedEdgeList.sort(null);
39
40
41        for(Edge next_edge: sortedEdgeList){ //for each edge(u, v) ∈ G.E
42
43            //FIND-SET(u) != FIND-SET(v)
44            if (cd.find(next_edge.u().index()) != cd.find(next_edge.v().index())) {
45
46                result.add(next_edge);//A = A U {(u,v)}
47
48                cd.Union(next_edge.u().index(), next_edge.v().index()); //UNION(u,v)
49            }
50        }
51
52        return result;
53    }
```

Código 1.1 - pacote **mintree.Kruskal**

No código 1.1, o algoritmo inicia criando uma lista de resultados temporários e um vetor abstraído em classe de conjuntos disjuntos.Posteriormente, possuímos uma iteração que irá criar um subconjunto para cada vértice do grafo. As linhas **28 e 30** possuem tempos constantes.

Entretanto, das linhas **32 a 34** temos uma Iteração que é denotada por $\theta(V)$ $onde\ V\ são\ as\ quantidades\ de\ vértices.$ Além disso, temos a função makeSet (código 1.2), que por sinal $\theta(1)$, culminando na complexidade resultante deste trecho de código $\theta(V)$.

```
public void makeSet(int i) {
    if(counter >= maxElements)
        throw new RuntimeException("O limite foi extrapolado");

    conjuntos[counter] = new Subset();
    conjuntos[counter].parent = i;
    conjuntos[counter].rank = 0;
    counter++;
}
```

Código 1.2 - pacote **utils.set.DisjointSet**

A linha 37 possui peso assintótico $\theta(1)$. Todavia, na linha 38, possuímos um algoritmo de ordenação denotada por $E \log_2 E$, que é padrão do Java. Por final, temos a iteração das linhas 41 a 50 que detém a parte principal do algoritmo. Percebe-se que as comparações chamam iterativamente o método recursivo *find* (Código 1.3) que possui complexidade $\log_2 V$.

```java
25    public int find(int i){
26        if (conjuntos[i].parent != i)
27            conjuntos[i].parent = find(conjuntos[i].parent);
28
29        return conjuntos[i].parent;
30    }
```

Código 1.3 - pacote **utils.set.DisjointSet**

Caso a condição seja satisfeita, a função Union é chamada para realizar a junção, Esta função possui custo $\theta(1)$ (Código 1.4).

```java
32    public void Union(int x, int y)
33    {
34        int xroot = find(x);
35        int yroot = find(y);
36
37        if (conjuntos[xroot].rank < conjuntos[yroot].rank)
38            conjuntos[xroot].parent = yroot;
39        else if (conjuntos[xroot].rank > conjuntos[yroot].rank)
40            conjuntos[yroot].parent = xroot;
41        else {
42            conjuntos[yroot].parent = xroot;
43            conjuntos[xroot].rank++;
44        }
45    }
46 }
```

Código 1.4 - pacote **utils.set.DisjointSet**

Sabendo que o algoritmo é aplicado a grafos conexos, infere-se que o número de arestas é maior que o número de vértices, impactando assim o na notação assintótica do algoritmo que seria denotado por $\theta(E \cdot log \,.\, E)$. Entretanto, é indispensável a presença dos vértices na determinação no custo assintótico do algoritmo, portanto reavaliando a sentença temos que:

$$|E| \leq |V|^2$$

$$\log_2 E \leq \log_2 V^2$$

$$\log_2 E = \theta(\log_2 V)$$

Assintoticamente falando, temos que esse algoritmo tem custo assintótico de $E \cdot \log_2 V$.

2. Prim

```
26
27        List<Edge> caminhoMinimo = new ArrayList<>();
28        boolean include[] = new boolean[graph.maxVertices()];
29
30
31        for(int i = 0; i < graph.numberOfVertices(); i++) {
32            graph.vertexAt(i).setData(new AttrVertex(graph.vertexAt(i)));
33            include[ graph.vertexAt(i).index() ] = true;
34        }
35
36        ((AttrVertex) vertexInit.getData()).key = 0;
37
38        FibonacciHeap<Vertex> Q = new FibonacciHeap<Vertex>(graph.vertices());
```

Código 2.1 - pacote **mintree.Prim**

No código 2.1, é possível determinar que as linhas 27,28,36 e 38 **são constantes** com complexidade assintótica $\theta(1)$, isto pode ser provado mostrando trechos do código a seguir:

```
44    public int maxVertices() {
45        return adj.length;
46    }
```

Código 2.2 - pacote **graph.AdjacencyListGraph;**

```
65    @Override
66    public Vertex vertexAt(int i) {
67        return adj[i].getOwnerVertex();
68    }
```

Código 2.3- pacote **graph.AdjacencyListGraph;**

```
279    @Override
280    public Iterable<Vertex> vertices() {
281        return new VertexIterator();
282    }
```

Código 2.4 - pacote **graph.AdjacencyListGraph;**

Posto isso, neste trecho o que mais importa é o laço de repetição da **linha 31 - 34**, que é $\theta(V)$, usado como componente auxiliar para comparação da presença de um vértice em uma fila.

Posteriormente nos trechos do código 2.5 temos o algoritmo de fato, a execução deste depende principalmente da fila implementada.Neste projeto foi usado o heap de fibonacci, que possui função mais custosa (*extractMin*) de $\theta(\log V)$ e as demais $\theta(1)$ que a torna um ótimo candidato.

```
40      while(!Q.isEmpty()) {
41          Vertex u = Q.extractMin();
42          include[u.index()] = false;
43
44          for(Edge edge: graph.edgesIncidentFrom(u)) {
45              Vertex v = edge.v();
46
47              if(include[v.index()] && edge.weight() < ((AttrVertex) v.getData()).key) {
48                  ((AttrVertex) v.getData()).pi = u;
49                  ((AttrVertex) v.getData()).key = edge.weight();
50                  Q.resortElement(v);
51              }
52          }
53      }
```

Código 2.5 - pacote **mintree.Prim**

```
@Override
public Iterable<Edge> edgesIncidentFrom(Vertex u) {
    return adj[index(u)].edges();
}
```

Código 2.6 - pacote **graph.AdjacencyListGraph;**

Percebemos que, o custo para recuperar as arestas para a iteração da linha 44 é O(1) o que culmina na complexidade resultante $\theta(E)$ e o laço *while* mais externo possui complexidade $\theta(V)$.

Logo temos:

$$\theta(V + E + V \cdot \log_2 V) \rightarrow \theta(E + V \cdot \log_2 V)$$

3. Conclusão

Portanto, vimos que neste projeto conseguimos realizar os objetivos prescritos no Tema III sem alterar a complexidade dos algoritmos.