Relatório de Implementação dos Algoritmos

Bruno Ramos Madson Araújo Nemuel Leal
Nilton Vasques

3 de Abril de 2013

1 Árvore Geradora Mínima

Definição: Seja G=(V,E) um grafo não direcionado e conexo, G'=(V,E') é chamado de subgrafo gerador se possuí os mesmos vértices de G. Portanto se tivermos em G' uma árvore, então o subgrafo é uma árvore geradora. Quando G é um grafo conexo, em que cada aresta possui um valor ou peso p(e), o peso total da árvore geradora é

$$\sum_{e \in E'} p(e)$$

onde p(e) é uma função que retorna o peso da aresta e. Á árvore geradora mínima é a árvore G' que possui o menor peso total dentre todas as árvores possíveis do grafo G[2]. Podemos enunciar a função para encontrar a árvore geradora mínima como

$$min \sum_{e \in E'} p(e)$$

. A partir dessa noção podemos visualizar que encontrar a árvore geradora mínima não é tão trivial assim. Se propormos uma solução pela força bruta, ou seja, encontrar todas as árvores geradoras e assim então verificar qual a que possui o menor peso total. No pior caso quando temos um grafo completo(em que todos os vértices se ligam uns aos outros) teríamos n^{n-2} árvores geradoras onde n é o número de nós, sendo assim teríamos uma solução em tempo exponencial $O(n^n)$ e inviável . Diante deste cenário alguns matemáticos elaboram soluções para o problema das Árvores Geradoras Mínimas, se utilizando de heurísticas gulosas para encontrar a solução ótima. No presente artigo abordaremos o Algoritmo de Kruskal e o de Prim, como estudo de caso.

1.1 Algoritmo de Kruskal

O algoritmo de Kruskal é um algoritmo guloso, que tem por objetivo encontrar uma árvore geradora mínima para um grafo conexo e valorado (com pesos nas arestas). Vale ressaltar que para árvores não conexas, o algoritmo encontra floresta geradora mínima, ou seja uma árvore geradora mínima para cada componente conexo do grafo. O algoritmo pode ser enunciado nos seguintes passos:

Data: Um grafo Conexo
Result: Uma árvore geradora mínima a partir de um grafo conexo
Criar uma floresta F, onde cada vértice do grafo é uma árvore separada;
Criar um conjunto S contendo todos as arestas do grafo;
while S *é não vazio* **do**
    Remova um aresta e com peso mínimo de S;
    Se e conecta duas diferentes árvores, então adicione e para floresta F;
    Caso contrário, discarte e, ou seja se a escolha de e gera um circuito em F, discarte-a;
end

Algorithm 1: Pseudo Código do algoritmo de Kruskal

1.1.1 Implementação

A implementação foi realizada em Java com uso da estrutura de dados de listas encadeadas para manipular os conjuntos disjuntos. O código fonte está disponível no Apêndice A. A seguir o pseudo código da implementação com as manipulações representadas pelas operações Union-Find :

```
   Data: V, E
   Result: A, W
1  W ← 0; A ← vazio;
2  for v ∈ V do
3  |  a[v] ← make-set(v);
4  end
5  L ← ordene(E, w);
6  k ← 0;
7  while k ≠| V | −1 do
8  |  remove(L, (u, v ));
9  |  a[u] ← find-set(u);
10 |  a[v] ← find-set(v);
11 |  if a[u] ≠ a[v] then
12 |  |  aceita(u, v);
13 |  |  A ← A ∪ {(u, v)};
14 |  |  W ← W + w(u, v);
15 |  |  k ← k + 1;
16 |  end
17 |  union(a[u], a[v]);
18 end
19 retorne(A, W);
```

Algorithm 2: Pseudo Código do algoritmo de Kruskal com UnionFind

1.1.2 Análise de Complexidade

A estrutura de dados UnionFind mantém um conjunto de elementos particionados em vários subconjuntos não sobrepostos. O algoritmo que controla essa estrutura possui duas operações principais:

- Find: Determina de qual subconjunto um elemento pertence.
- Union: Faz a união de dois subconjuntos em um só subconjunto.

A ordenação na linha 5 tem complexidade $\Theta(\mid E \mid log \mid E \mid)$ e domina a complexidade das demais operações. A repetição das linhas 7-17 será executado $\Theta(\mid E \mid)$ no pior caso. Logo, a complexidade total das linhas 9-10 será $\Theta(\mid E \mid f(\mid V \mid))$, onde $f(\mid V \mid))$ é complexidade da função **find-set**. As linhas de 12 a 15 serão executados $\mid V \mid -1$ vezes no total, pois para um grafo contendo N vértices, precisamos de apenas N-1 arestas para interligar todos os nós e gerar uma árvore geradora mínima. Assim, a complexidade total de execução destas linhas será $\Theta(\mid V \mid .g(\mid V \mid)$ onde $g(\mid V \mid)$ é a complexidade

de realizar **union**. A complexidade do algoritmo de Kruskal será então:

$$\Theta(\mid E \mid log \mid E \mid + \mid E \mid .f(\mid V \mid) + \mid V \mid .g(\mid V \mid))$$

A estrutura de dados UnionFind foi implementada na sua forma simples, com o uso de uma lista encadeada. Sendo assim a complexidade da função find é $\omega(n)$, e union tem complexidade $\Theta(n)$ [1]. A complexidade final da implementação foi:

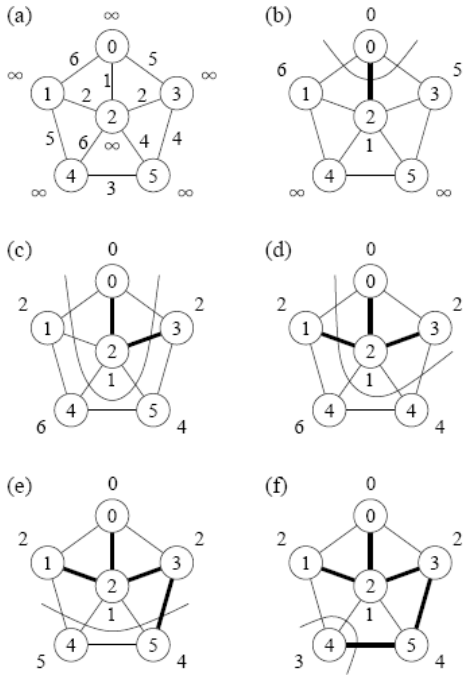
$$\Theta(\mid E \mid log \mid E \mid + \mid E \mid .\Omega(\mid V \mid) + \mid V \mid .\Theta(\mid V \mid))$$

A complexidade pode ser reduzida utilizando de uma estrutura de dados mais refinada para implementar a manipulação dos conjuntos disjuntos, como por exemplo usar uma lista encadeada e *weighted-union heuristic*[1], consegue-se uma complexidade de $\Theta(m+nlogn)$ para realizar as m operações de **make-set**, **find-set** e **union** [1].

1.2 Algoritmo de Prim

Assim como o algoritmo de Kruskal's o algoritmo de Prim utilizada também uma heurística gulosa para solucinar o problema da Àrvore Geradora Mínima. A heurística utilizada é procurar o caminho mais curto, dentre todos os possíveis, de maneira similar ao algoritmo de Djikstra. O algoritmo de Prim's tem uma propriedade de que as arestas em A sempre forma uma árvore simples *Fig.1.*

Fig. 1: Ilustração do Algoritmo de Prim



O algoritmo genérico de Prim procura encontrar o caminho mais curto de um vértice para os vértices vizinhos, até que todos os vértices estejão ligados uns aos outros. O pseudo-código pode ser visualizado logo abaixo:

Data: Um grafo Conexo
Result: Uma árvore geradora mínima a partir de um grafo conexo
Escolha um vértice S para iniciar o subgrafo;
while *há vértices que não estão no subgrafo* **do**
    selecione uma aresta segura;
    insira a aresta segura e seu vértice no subgrafo;
end

Algorithm 3: Pseudo Código do algoritmo de Prim

1.2.1 Implementação

Assim como a implementação anterior o algoritmo de Prim's foi implementado em Java e fez uso da Matriz de Adjacências Binária como estrutura de dados para armazenar os vértices adjacentes de um dado vértice u. O código fonte está disponível no Apêndice A. O pseudo código que foi utilizado para a implementação pode ser visualizado em *Algorithm 4*.

```
   Data: V, E
   Result: A, W
 1 dist[r] ← 0; Q ← V;
 2 for v ∈ V − {r} do
 3 |  dist[r] ← ∞;
 4 end
 5 pred[r] ← NULL;
 6 A ← ∅;
 7 W ← 0;
 8 while Q não for vazio do
 9 |  remover de Q o vértice u com menor valor em dist;
10 |  W ← W + dist[u];
11 |  if pred[u] ≠ null then
12 |  |  A ← A ∪ {(pred[u], u};
13 |  end
14 |  for v ∈ Adj[u] do
15 |  |  if v ∈ Q and dist[v] > w[u, v] then
16 |  |  |  dist[v] ← w[u, v];
17 |  |  |  pred[v] ← u;
18 |  |  end
19 |  end
20 end
21 retorne(A, W);
```

Algorithm 4: Pseudo Código do algoritmo de Prim

1.2.2 Ánalise de Complexidade

A complexidade do algoritmo de Prim está diretamente ligada a maneira de como é implementada a estrutura de dados em Q. Uma simples implementação utilizando matrizes de adjacência vai requerer complexidade $\Theta(\mid V \mid^2)$ e foi a estrutura de dados utilizada neste trabalho. Utilizando de uma heap binária a complexidade cai para $\Theta(\mid E \mid log \mid V \mid)$, ainda assim é possível decrescer ainda mais a complexidade utilizando como estrutura de dados uma Fibonacci Heap com complexidade $\Theta(\mid E \mid + \mid V \mid log \mid V \mid)$ [1].

2 Caminho Mínimo em Grafos Orientados

2.1 Algoritmo de Djikstra

O algoritmo de Dijkstra, proposto pelo cientista E. W. Dijkstra, resolve o problema de encontrar o menor caminho entre dois vertices num grafo orientado ou não com arestas de peso não negativo. O algoritimo de Dijkstra é um metodo guloso para resolver o problema do caminho mais curto. A ideia por trás do método guloso é efetuar uma BFS ponderada sobre um dado grafo, a partir de um nó n. Dijkstra é comumente implementado com uma fila de prioridade como uma heap, de modo que em cada iteração, quando precisamos de obter o próximo nó a ser visitado, então este nó escolhido será o nó mais próximo ao nó n.

O algoritmo de Dijkstra mantem um conjunto S de vertices cujo o peso do menor caminho a partir de s já foi determinado. O algoritmo seleciona repetidamente o vértice $u \in V - S$ com o menor caminho estimado, acrescenta u em S, e relaxa todas as arestas que incidem em u. O seguinte pseudo-código usa uma fila de prioridade Q de vértices, introduzidos pelos seus valores d.

```
   Data: G,w,s
 1 INITIALIZE-SINGLE-SOURCE(G,s);
 2 S ← ∅;
 3 Q ← V[G];
 4 while Q = ∅ do
 5 |   u ← EXTRACTING − MIN(Q);
 6 |   S ← S ∪ {u};
 7 |   foreach vertex v ∈ Adj[u] do
 8 |   |   RELAX(u,v,w);
 9 |   end
10 end
```

Algorithm 5: Pseudo Código do Algoritmo de Djijstra

2.1.1 Análise de Complexidade

Seja n o número de nós e m o número de arestas.

Uma vez que implementamos nossa fila de prioridade como uma heap, então o tempo de complexidade para remover o elemento minimo da heap ou adicionar um novo elemento é $\Theta(logn)$. Agora precisamos considerar o fato de quando atualizamos as menores distância dos nós, ou a fase de relaxação das arestas obtemos $\Theta(n + m)$.

Com isso, o tempo que o algoritmo de Dijkstra gasta em cada nó é $\Theta(mlogn)$, enquanto que se precisamos visitar todos os nós, então a com-

plexidade de tempo do algoritmo de Dijkstra seria $\Theta((n+m)logn)$.

Até agora, consideramos apenas a computação de um único nó para todos os outros nós. Contudo a complexidade para computar a menor distância partindo de todos os nós para todos os outros nós é $\Theta(n(n+m)logn)$. Assim, se temos um grafo completo a complexidade seria $\Theta(n^2logn)$.

2.1.2 Implementação

O algoritmo foi implementado na linguagem python no arquivo dijkstra.py. Utiliza a biblioteca pygraphviz, que pode ser instalada em um computador Debian/Linux com o seguinte comando:

aptitude install python-pygraphviz

O arquivo dijkstra.py possui o seguinte cabeçalho:

```
1  from pygraphviz import *
2  from random import *
3  from threading import Thread
4  import Image
5  import os
```

A função main contem os principais passos do algoritmo: 1) cria um grafo, os vértices e as arestas já são predefinidos; 2) exibe a imagem do grafo gerado; 3) cria uma matriz contendo os menores caminhos a partir de todos o vértices para todos os vértices; 4) solicita o vértice origem e o vértice destino; 5) exibe a imagem do menor caminho de origem para destino.

```
1  def main():
2
3      # criao grafo
4      grafo = criarGrafo()
5      grafo.draw("grafo.jpg", "jpg", "dot")
6      #exibe o grafo
7      Image.open("grafo.jpg").show();
8      #calcula as distancias
9      matriz = None
10     matriz = calcularDistancias(grafo)
11     //selecao dos vertices
12     o = raw_input("\nEscolha um vertice de origem: ")
13     #res = matriz.get(o)
14     d = raw_input("Escolha um vertice destino: ")
15     #desenha o menor caminho entre origem e destino
16     desenharCaminho(grafo, matriz[o][1], o, d)
```

A função calcularDistancias irá executar o algoritmo de Dijkstra para todos os vértices como origem irá armazenar o resultado em uma matriz.

```
def calcularDistancias(grafo):
    matriz = {}
    for v in grafo:
        dijk = Dijkstra()
        #calcula menor distancia de v para todos os
           outros vertices
        MenorDistancias = dijk.menorDistancias(grafo, v)
        # Armazena a  menor dist ncia partindo de v
           at  os outros v rtices
        matriz[v] = MenorDistancias

    return matriz
```

O metodo menorDistancias da classe Dijkstra executa o algoritmo de dijkstra e retorna a lista dos menores caminhos do vertice v para os outros vertices.

```
    def menorDistancias(self, grafo, v):

        #1   passo: iniciam-se os valores:
        distancia = {}

        for no in grafo:
            if no in grafo.neighbors(v): #se o no esta
               na lista dos vizinhos de v.
                distancia[no] = (float(grafo.get_edge(v,
                    no).attr['label']), v) # armazena o
                   peso da aresta
            else: # nao vizinho ou o mesmo vertice
                distancia[no] = (float('+inf'), None) #
                   distancia = + infinito

        # Armazena a dist ncia partindo de v at  os
           outros v rtices
        distancias = distancia.keys()
        vertices = [v]
        distanciaCopy = distancia.copy()
        listaDist = [distanciaCopy]


        while (len(distancias) > len(vertices)):
            ditems = distancia.items()
            daux = []
            for a in ditems:
```

```
23                  if a[0] not in vertices:
24                      daux += [a]
25
26              menor = daux[0]
27              for a in daux:
28                  if a[1][0]<menor[1][0]:
29                      menor = a
30
31              vertices += [menor[0]]
32
33              for no in grafo.neighbors(menor[0]):
34                  if no not in vertices:
35                      if distancia[no][0] > (menor[1][0]+
                           float(grafo.get_edge(menor[0], no
                           ).attr['label'])):
36                          distancia[no] = (menor[1][0]+
                               float(grafo.get_edge(menor
                               [0], no).attr['label']),
                               menor[0])
37
38
39              distanciaCopy = distancia.copy()
40              listaDist += [distanciaCopy]
41
42
43          return (vertices, distancia, listaDist)
```

2.2 Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall usa uma formulação de programação dinâmica para resolver o problema de caminhos mais curtos de todos os pares em grafo orientado G = (V,E). O algoritmo é executado no tempo $\Theta(V^3)$. O algoritmo de Floyd-Warshall se baseia na observação a seguir. Sejan V = 1,2,...,n os vértices de G, e considere um subconjunto 1,2,...,k de vertices para algum k. Para qualquer par de vértices $i, j \in V$, considere todos os caminhos desde i até j cujos vértices intermediários são todos traçados a partir de 1,2,...,k, e seja p um caminho de peso minimo dentre eles. O algoritmo de Floyd-Warshal explora um relacionamento entre o caminho p e caminhos mais curtos desde i até j com todos vértices intermediários no conjunto1,2,...,k-1. O Relacionamento depende do fato de k ser ou não um vértice intermediário do caminho p.

Se k não é um vértice intermediário do caminho p, então todos os vértices

intermediários do caminho p estão no conjunto 1,2,...,k-1. Desse modo, um caminho mais curto desde o vértice i até o j com todos os intermediários no conjunto 1,2,...,k-1 também é um caminho mais curto desde i até j com todos os vértices intermediários no conjunto 1,2,...,k.

Se k é um vertice intermediário do caminho p, então desmembramos p em ip_1kp_2j. P1 é um caminho mais curto desde i até k com todos os vértices intermediários no conjunto 1,2,...,k. Como o vértice k não é um vertice intermediário do caminho p1, vemos que p1 é um caminho mais curto desde i até k com todos os vértices intermediários no conjunto 1,2,...,k-1. De modo semelhante, p2 é um caminho mais curto até o vértice j com todos os vértices intermediários no conjunto 1,2,...,k-1.

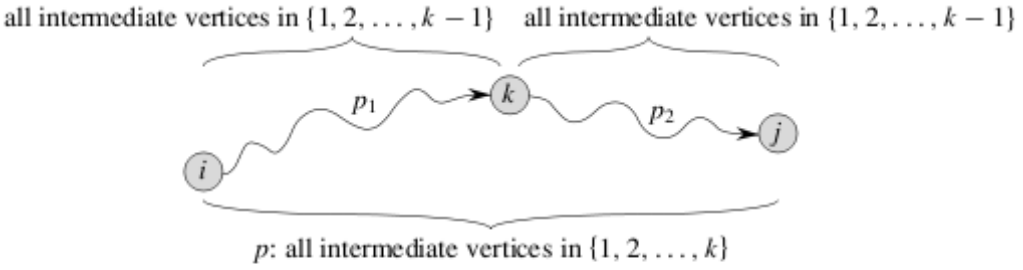


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

A formulação recursiva seguindo a discussão acima é dada por:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

Com base na recorrência acima, o seguinte procedimento bottom-up pode ser usado para calcular o dij(k) , a fim de aumentar os valores de k. A sua entrada é uma matriz n x n W definido como na equação. O procedimento retorna a matriz D(n) com os pesos dos menores caminhos.

2.2.1 Implementação

O algoritmo foi implementado na linguagem python no arquivo floyd.py.

A função main cria um grafo predefinido e depois chama a funcao floydWarshall com o gafo sendo passado como parâmetro.

```
1 if __name__ == '__main__':
```

Data: W
$n \leftarrow rows[W]$;
$D^{(}0) \leftarrow W$;
for $k \leftarrow 1$ *to* n **do**
  for $i \leftarrow n$ *to* n **do**
    for $j \leftarrow 1$ *to* n **do**
      $\mathrm{d(k)} \leftarrow min(\mathrm{d}^{(}k-1), d^{(}k-1) + d^{(}k-1)\ )$;
    end
  end
end
return $D^{(}n)$;

```
2
3     #Cria o grafo
4
5     grafo = {'A':{'A':0,'B':INF,'C':12,'D':6,'E':4},
6
7              'B':{'A':INF,'B':0,'C':INF,'D':1,'E':INF},
8
9              'C':{'A':INF,'B':INF,'C':0,'D':15,'E':3},
10
11             'D':{'A':INF,'B':INF,'C':15,'D':0,'E':4},
12
13             'E':{'A':INF,'B':7,'C':INF,'D':INF,'E':0}
14
15             }
```

A funcão floydWarshall primeiro armazena a distancia dos vertices em seguida executa o algoritmo de Floyd-Warshal para obter o menor caminho para todos os pares por fim exibe a tabela com os resultados.

```
1  def floydWarshall(grafo):
2
3      nos = grafo.keys()
4
5      distancia = {}
6
7      #armezena a distancia do no n para o no k
8      for n in nos:
9
10         distancia[n] = {}
11
12         for k in nos:
```

```
13
14              distancia[n][k] = grafo[n][k]
15      # Floyd-warshal - programacao dinamica
16      for k in nos:
17
18          for i in nos:
19
20              for j in nos:
21
22                  if distancia[i][k] + distancia[k][j] <
                       distancia[i][j]:
23
24                      distancia[i][j] = distancia[i][k]+
                           distancia[k][j]
25      # imprime a tabela com resultado
26      printSolution(distancia)
```

Appendices

A

Algoritmos Implementados no Projeto

```
1  public class Kruskal implements GraphAlgorithm{
2      private Graph mGraph;
3      private int edges[];
4      private int edgeIndex = 0;
5
6      public Kruskal(Graph grafo) {
7          mGraph = grafo;
8      }
9
10     @Override
11     public void init(){
12         edgeIndex = 0;
13         for (int i = 0; i < mGraph.getVerticesCount() ;
               i++) {
14             Vertice vertice = mGraph.getVertices()[i];
15             UnionFind.makeSet(vertice);
16         }
17         edges = new int[mGraph.getArestasCount()];
```

```
18          for( int i = 0; i < edges.length; edges[i] = i
               ++);
19          qsort(0, edges.length -1);
20      }
21      @Override
22      public boolean performStep() {
23          if( edgeIndex >= edges.length )
24              return true;
25          Aresta aresta   = mGraph.getArestas()[ edges[
               edgeIndex]];
26          if( aresta.status == Status.WAITING ){
27              aresta.status = Status.PROCESSING;
28              return false;
29          }
30          edgeIndex++;
31          UnionElement u  = aresta.u;
32          UnionElement v  = aresta.v;
33          if( !UnionFind.find(u).equals(UnionFind.find(v))
               ){
34              aresta.status = Status.TAKED;
35              UnionFind.union(u, v);
36          }else{
37              aresta.status = Status.DISCARDED;
38          }
39          return false;
40      }
41  }
```

Implementação do Algoritmo de Kruskal em Java

```
1   public class Prim implements GraphAlgorithm{
2       private Graph mGraph;
3       private AdjacencyMatrix verticesMatrix;
4       private AdjacencyMatrix matrix;
5       private boolean processing = false;
6
7       public Prim( Graph graph) {
8           mGraph = graph;
9       }
10      @Override
11      public void init() {
12          processing = false;
13          verticesMatrix = new AdjacencyMatrix(mGraph.
               getVerticesCount());
```

```
14          verticesMatrix.makeAdjacency(0, 0);
15          matrix = mGraph.createAdjacencyMatrix();
16      }
17
18      @Override
19      public boolean performStep() {
20          Aresta bestChoice   = null;
21          int bestVertice     = -1;
22          int vertices[] = verticesMatrix.getAdjacencys(0)
                ;
23          for( int v = 0; v < vertices.length; v++){
24              if( vertices[v] == 1){
25                  int adjacencys[] = matrix.getAdjacencys(
                        v);
26                  for( int i = 0; i < adjacencys.length; i
                        ++){
27                   if(adjacencys[i] == 1){
28                      Aresta aresta = mGraph.getAresta
                            (i, v);
29                      if( aresta != null){
30                          if( !processing ){
31                              aresta.status = Status.
                                    PROCESSING;
32                          }else if ( bestChoice !=
                                null ){
33                              if( bestChoice.weight >
                                    aresta.weight ){
34                                  bestChoice  = aresta
                                        ;
35                                  bestVertice = i;
36                              }
37                          }else{
38                              bestChoice = aresta;
39                              bestVertice = i;
40                          }
41                      }
42                   }
43                  }
44              }
45          }
46          if(!processing){
47              processing = true;
48              return false;
```

```
49          }
50          boolean finish = true;
51          if( bestVertice != -1 ){
52              bestChoice.status = Status.TAKED;
53              for(int i = 0; i < vertices.length; i++){
54                  if( vertices[i] == 1){
55                      matrix.removeAdjacency(i,
                            bestVertice);
56                  }else{
57                      finish = false;
58                  }
59              }
60              processing = false;
61              verticesMatrix.makeAdjacency(0, bestVertice)
                    ;
62          }
63          return finish;
64      }
65
66
67  }
```

Implementação do Algoritmo de Prims em Java

B

Código Fonte das Estrutura de Dados Implementadas no Projeto

```
1   package br.ufba.datastructures;
2   /**
3    * @author niltonvasques
4    * UnionFind data structure
5    * http://en.wikipedia.org/wiki/Disjoint-
       set_data_structure
6    */
7   public class UnionFind {
8
9       public interface UnionElement{
10
11          public UnionElement getRoot();
12          public UnionElement getParent();
13          public void setRoot(UnionElement x);
```

```
14          public void setParent(UnionElement x);
15
16      }
17
18      public static void makeSet( UnionElement x ){
19          x.setParent(x);
20      }
21
22      public static UnionElement find( UnionElement x){
23          if ( x.getParent() == x )
24              return x;
25          else
26              return find(x.getParent());
27      }
28
29      public static void union( UnionElement x,
            UnionElement y){
30          x.setRoot( find(x) );
31          y.setRoot( find(y) );
32          x.getRoot().setParent( y.getRoot() );
33      }
34
35  }
```

Interface Para Operações Union-Find com Lista Encadeada

```
1   package br.ufba.datastructures;
2
3   /**
4    * @author niltonvasques
5    * http://pt.wikipedia.org/wiki/Matriz_de_adjac%C3%
         AAncia
6    */
7   public class AdjacencyMatrix {
8
9       private static final int MEM_BLOCK_SIZE = 32;
10      int matrix[];
11      int stride;
12      public AdjacencyMatrix(int n) {
13          stride = n;
14          int size = (int)(stride * stride);
15          int lenght = 1 + (size/32);
16          matrix = new int[ lenght ];
17      }
```

```
18
19      public void makeAdjacency(int element, int adjacency
            ){
20          makeAdjacencyInternal(element, adjacency);
21          makeAdjacencyInternal(adjacency, element);
22      }
23
24      public void removeAdjacency(int element, int
            adjacency ){
25          removeAdjacencyInternal(element, adjacency);
26          removeAdjacencyInternal(adjacency, element);
27      }
28
29      private void makeAdjacencyInternal(int element, int
            adjacency) {
30          int bitIndex            = (element*stride+
                adjacency);
31          int index       = (int) (bitIndex/MEM_BLOCK_SIZE
                );
32          int shift       = bitIndex % MEM_BLOCK_SIZE;
33          matrix[index]   |= (0x01 << shift);
34      }
35
36      private void removeAdjacencyInternal(int element,
            int adjacency) {
37          int bitIndex            = (element*stride+
                adjacency);
38          int index       = (int) (bitIndex/MEM_BLOCK_SIZE
                );
39          int shift       = bitIndex % MEM_BLOCK_SIZE;
40          matrix[index]   &= ~(0x01 << shift);
41      }
42
43      public int[] getAdjacencys(int element){
44          int adjacencys[] = new int[stride];
45
46          int bitIndex        = (element*stride);
47          int bitRemainder    = (bitIndex % MEM_BLOCK_SIZE
                );
48
49          for( int i = 0; i < stride; i++){
50              int x = ((bitIndex+i)/MEM_BLOCK_SIZE);
51              int y =  bitRemainder + i;
```

```
52
53              int ret = (matrix[x] >> y) & 0x01;
54              adjacencys[i] = ret;
55          }
56
57          return adjacencys;
58      }
59
60      public boolean checkAdjacency(int u, int v){
61
62          int bitIndex        = (u*stride);
63          int index           = (int) (bitIndex/
                MEM_BLOCK_SIZE);
64
65          int x = (v/MEM_BLOCK_SIZE) + index;
66          int y = (bitIndex % MEM_BLOCK_SIZE) + v;
67
68          return ((matrix[x] >> y) & 0x01) == 0x01;
69      }
70  }
```

Implementação da Matriz de Adjacências Binária

Referências

[1] Thomas H. Cormem, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, volume ISBN 0-262-03293-7, chapter 21: Data structures for Disjoint Sets, pages 498–524. MIT Press, second edition, 2001.

[2] Fernando Nogueira. Problema da Árvore Gerador Mínima. *UFJF*.

[3] F. Prado, T. Almeida, and V. N. Souza. Introdução ao Estudo sobre Árvore Geradora Mínima em Grafos com Parâmetros Fuzzy. *UNICAMP - Faculdade de Engenharia Elétrica*.