

Relatório de Implementação dos Algoritmos Referente ao Problema da Árvore Geradora Mínima

Bruno Ramos

Madson Araújo
Nilton Vasques

Nemuel Leal

2 de Abril de 2013

1 Definição

Seja $G=(V,E)$ um grafo não direcionado e conexo, $G'=(V,E')$ é chamado de subgrafo gerador se possui 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 . A 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 Árvore 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 todas 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  $W \leftarrow 0; A \leftarrow \text{vazio};$ 
2 for  $v \in V$  do
3    $a[v] \leftarrow \text{make-set}(v);$ 
4 end
5  $L \leftarrow \text{ordene}(E, w);$ 
6  $k \leftarrow 0;$ 
7 while  $k \neq |V| - 1$  do
8    $\text{remove}(L, (u, v));$ 
9    $a[u] \leftarrow \text{find-set}(u);$ 
10   $a[v] \leftarrow \text{find-set}(v);$ 
11  if  $a[u] \neq a[v]$  then
12     $\text{aceita}(u, v);$ 
13     $A \leftarrow A \cup \{(u, v)\};$ 
14     $W \leftarrow W + w(u, v);$ 
15     $k \leftarrow k + 1;$ 
16  end
17   $\text{union}(a[u], a[v]);$ 
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 $\Theta(|E| \log |E|)$ e domina a complexidade das demais operações. A repetição das linhas 7-17 será executado $\Theta(|E|)$ no pior caso. Logo, a complexidade total das linhas 9-10 será $\Theta(|E| f(|V|))$, onde $f(|V|)$ é complexidade da função **find-set**. As linhas de 12 a 15 serão executados $|V| - 1$ 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 $\Theta(|V| g(|V|))$ onde $g(|V|)$ é a complexidade

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

$$\Theta(|E| \log |E| + |E| \cdot f(|V|) + |V| \cdot g(|V|))$$

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:

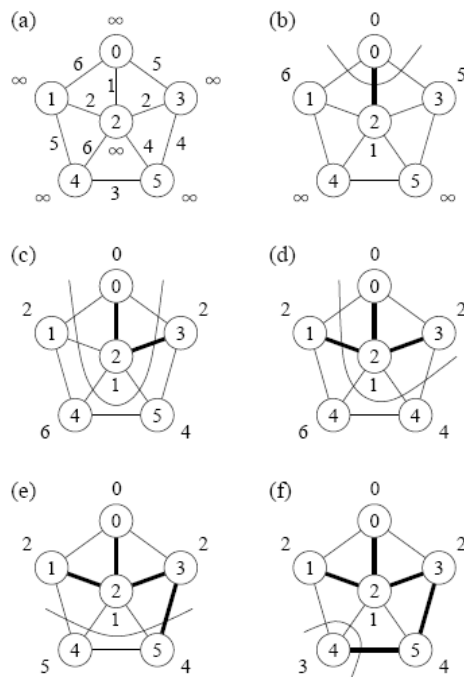
$$\Theta(|E| \log |E| + |E| \cdot \Omega(|V|) + |V| \cdot \Theta(|V|))$$

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 $\Theta(m + n \log n)$ 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 solucionar 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 Dijkstra. 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 estejam 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  $dist[r] \leftarrow 0; Q \leftarrow V;$   
2 for  $v \in V - \{r\}$  do  
3    $dist[v] \leftarrow \infty;$   
4 end  
5  $pred[r] \leftarrow NULL;$   
6  $A \leftarrow \emptyset;$   
7  $W \leftarrow 0;$   
8 while  $Q$  não for vazio do  
9   remover de  $Q$  o vértice  $u$  com menor valor em  $dist$ ;  
10   $W \leftarrow W + dist[u];$   
11  if  $pred[u] \neq null$  then  
12     $A \leftarrow A \cup \{(pred[u], u)\};$   
13  end  
14  for  $v \in Adj[u]$  do  
15    if  $v \in Q$  and  $dist[v] > w[u, v]$  then  
16       $dist[v] \leftarrow w[u, v];$   
17       $pred[v] \leftarrow u;$   
18    end  
19  end  
20 end  
21 retorne( $A, W$ );
```

Algorithm 4: Pseudo Código do algoritmo de Prim

1.2.2 Análise 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 $\Theta(|V|^2)$ e foi a estrutura de dados utilizada neste trabalho. Utilizando de uma heap binária a complexidade cai para $\Theta(|E| \log |V|)$, ainda assim é possível decrescer ainda mais a complexidade utilizando como estrutura de dados uma Fibonacci Heap com complexidade $\Theta(|E| + |V| \log |V|)$ [1].

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() ;
14            i++) {
15            Vertice vertice = mGraph.getVertices()[i];
16            UnionFind.makeSet(vertice);
17        }
18        edges = new int[mGraph.getArestasCount()];
19        for( int i = 0; i < edges.length; edges[i] = i
20            ++);
21        qsort(0, edges.length -1);
22    }
23    @Override
24    public boolean performStep() {
25        if( edgeIndex >= edges.length )
26            return true;
27        Aresta aresta = mGraph.getArestas()[ edges[
28            edgeIndex]];
29        if( aresta.status == Status.WAITING ){
30            aresta.status = Status.PROCESSING;
31            return false;
32        }
33        edgeIndex++;
34        UnionElement u = aresta.u;
35        UnionElement v = aresta.v;
```

```

33         if( !UnionFind.find(u).equals(UnionFind.find(v))
34             ){
35             aresta.status = Status.TAKED;
36             UnionFind.union(u, v);
37         }else{
38             aresta.status = Status.DISCARDED;
39         }
40         return false;
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.
14             getVerticesCount());
15         verticesMatrix.makeAdjacency(0, 0);
16         matrix = mGraph.createAdjacencyMatrix();
17     }
18
19     @Override
20     public boolean performStep() {
21         Aresta bestChoice = null;
22         int bestVertex = -1;
23         int vertices[] = verticesMatrix.getAdjacencys(0)
24             ;
25         for( int v = 0; v < vertices.length; v++){
26             if( vertices[v] == 1){
27                 int adjacencys[] = matrix.getAdjacencys(
28                     v);
29                 for( int i = 0; i < adjacencys.length; i
30                     ++){
31                     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-
6  \*   set\_data\_structure
7  */
8 public class UnionFind {
9
10     public interface UnionElement{
11
12         public UnionElement getRoot();
13         public UnionElement getParent();
14         public void setRoot(UnionElement x);
15         public void setParent(UnionElement x);
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,
30                             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%A2ncia
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     ){
21         makeAdjacencyInternal(element, adjacency);
22         makeAdjacencyInternal(adjacency, element);
23     }
24
25     public void removeAdjacency(int element, int
26     adjacency ){
27         removeAdjacencyInternal(element, adjacency);
28         removeAdjacencyInternal(adjacency, element);
29     }
30
31     private void makeAdjacencyInternal(int element, int
32     adjacency) {
33         int bitIndex = (element*stride+
34         adjacency);

```

```

31     int index          = (int) (bitIndex/MEM_BLOCK_SIZE
32         );
33     int shift          = bitIndex % MEM_BLOCK_SIZE;
34     matrix[index]      |= (0x01 << shift);
35 }
36 private void removeAdjacencyInternal(int element,
37     int adjacency) {
38     int bitIndex        = (element*stride+
39         adjacency);
40     int index          = (int) (bitIndex/MEM_BLOCK_SIZE
41         );
42     int shift          = bitIndex % MEM_BLOCK_SIZE;
43     matrix[index]      &= ~(0x01 << shift);
44 }
45
46 public int[] getAdjacencys(int element){
47     int adjacencys[] = new int[stride];
48
49     int bitIndex      = (element*stride);
50     int bitRemainder  = (bitIndex % MEM_BLOCK_SIZE
51         );
52
53     for( int i = 0; i < stride; i++){
54         int x = ((bitIndex+i)/MEM_BLOCK_SIZE);
55         int y = bitRemainder + i;
56
57         int ret = (matrix[x] >> y) & 0x01;
58         adjacencys[i] = ret;
59     }
60
61     return adjacencys;
62 }
63
64 public boolean checkAdjacency(int u, int v){
65
66     int bitIndex      = (u*stride);
67     int index          = (int) (bitIndex/
68         MEM_BLOCK_SIZE);
69
70     int x = (v/MEM_BLOCK_SIZE) + index;
71     int y = (bitIndex % MEM_BLOCK_SIZE) + v;

```

```
68         return ((matrix[x] >> y) & 0x01) == 0x01;
69     }
70 }
```

Implementação da Matriz de Adjacências Binária

Referências

- [1] Thomas H. Cormen, 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 Geradora 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*.