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

Bruno Ramos Madson Araújo Nemuel Leal
Nilton Vasques

April 2, 2013

1 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 encontra 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 se utilizou da estrutura de dados UnionFind. A seguir o pseudo código da implementação com a estrutura de dados UnionFind :

```
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

O algoritmo de Prim...

A
Algoritmos Implementados no Projeto

```java
public class Kruskal implements GraphAlgorithm{
    private Graph mGraph;
    private int edges[];
    private int edgeIndex = 0;

    public Kruskal(Graph grafo) {
        mGraph = grafo;
    }

    @Override
    public void init(){
        edgeIndex = 0;
        for (int i = 0; i < mGraph.getVerticesCount() ; i++) {
            Vertice vertice = mGraph.getVertices()[i];
            UnionFind.makeSet(vertice);
        }
        edges = new int[mGraph.getArestasCount()];
        for( int i = 0; i < edges.length; edges[i] = i ++);
        qsort(0, edges.length -1);
    }
    @Override
    public boolean performStep() {
        if( edgeIndex >= edges.length )
            return true;
        Aresta aresta   = mGraph.getArestas()[ edges[ edgeIndex]];
        if( aresta.status == Status.WAITING ){
            aresta.status = Status.PROCESSING;
            return false;
        }
        edgeIndex++;
        UnionElement u  = aresta.u;
        UnionElement v  = aresta.v;
        if( !UnionFind.find(u).equals(UnionFind.find(v)) ){
            aresta.status = Status.TAKED;
            UnionFind.union(u, v);
```

```
        }else{
            aresta.status = Status.DISCARDED;
        }
        return false;
    }
}
```

Implementação do Algoritmo de Kruskal em Java

```
public class Prim implements GraphAlgorithm{
    private Graph mGraph;
    private AdjacencyMatrix verticesMatrix;
    private AdjacencyMatrix matrix;
    private boolean processing = false;

    public Prim( Graph graph) {
        mGraph = graph;
    }
    @Override
    public void init() {
        processing = false;
        verticesMatrix = new AdjacencyMatrix(mGraph.getVerticesCount());
        verticesMatrix.makeAdjacency(0, 0);
        matrix = mGraph.createAdjacencyMatrix();
    }

    @Override
    public boolean performStep() {
        Aresta bestChoice   = null;
        int bestVertice     = -1;
        int vertices[] = verticesMatrix.getAdjacencys(0);
        for( int v = 0; v < vertices.length; v++){
            if( vertices[v] == 1){
                int adjacencys[] = matrix.getAdjacencys(v);
                for( int i = 0; i < adjacencys.length; i++){
                  if(adjacencys[i] == 1){
                      Aresta aresta = mGraph.getAresta(i, v);
                      if( aresta != null){
                          if( !processing ){
```

```
                                aresta.status = Status.PROCESSING;
                            }else if ( bestChoice != null ){
                                if( bestChoice.weight > aresta.weight ){
                                    bestChoice  = aresta;
                                    bestVertice = i;
                                }
                            }else{
                                bestChoice = aresta;
                                bestVertice = i;
                            }
                        }
                    }
                }
            }
        }
        if(!processing){
            processing = true;
            return false;
        }
        boolean finish = true;
        if( bestVertice != -1 ){
            bestChoice.status = Status.TAKED;
            for(int i = 0; i < vertices.length; i++){
                if( vertices[i] == 1){
                    matrix.removeAdjacency(i, bestVertice);
                }else{
                    finish = false;
                }
            }
            processing = false;
            verticesMatrix.makeAdjacency(0, bestVertice);
        }
        return finish;
    }


```

```
}
```

Implementação do Algoritmo de Prims em Java

References

[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*.