Analizando os Algoritmos de Kruskal e Prim

Micael Andrade Dos Santos

[1]Departamento de Computação (DCOMP) – Universidade Federal de Sergipe (UFS)
Av. Marechal Rondon, s/n – Jardim Rosa Elze – CEP 49100-000
São Cristóvão – SE – Brazil

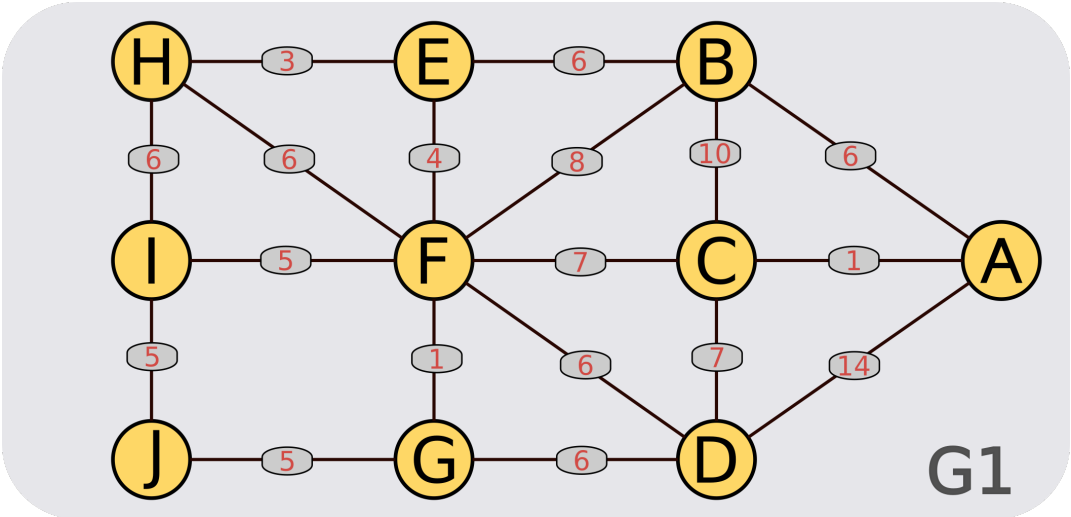
kaell_andrade@academico.ufs.br, micael.santos@dcomp.ufs.br

Resumo. *Este relatório tem como objetivo fazer uma análise simples de dois algoritmos clássicos utilizados para encontrar uma árvore de custo mínimo. Primeiro descreveremos de forma básica o que é uma árvore geradora mínima. Em seguinda, será mostrado um pseudocódigo de dois algoritmos clássicos para determinar essas árvores. O primeiro algoritmo analizado será o de Prim seguido do algoritmo de Kruskal.*

1. Árvore geradora mínima

Uma árvore geradora mínima (MSTs) de um determinado grafo $G1$ ponderado e conexo é uma árvore que contém todos os vértices de $G1$ de tal forma que o grafo $G1$ continua conexo, porém cada aresta dessa árvore contém o menor custo. [Cormen et al. 2009]. Dessa forma, como podemos encontrar uma árvore geradora mínima para o grafo $G1$[1] abaixo ?

Figura 1. G1



[GOLDBARG 2012]

2. Prim

O algoritmo de *Prim* consiste em particionar os vértices em dois conjuntos. O conjunto de vértices abertos e o conjunto dos vértices fechados. Podemos tomar o conjunto $BRANCO$ como o conjunto de todos o vértices que estão abertos *(não visitados)* e, o conjunto $PRETO$ de todos os vértices que estão fechados *(visitados)*. Segue abaixo o algoritmo de *Prim* o qual foi usado para encontrar uma árvore mínima de $G1$[1].

```
1  def prim(G:GRAFO(V,E)) -> GRAFO(V,E):
2      v_i = G[0];
3      v_i.cor = 'PRETO';
4
5      T:GRAFO = G(V, []); #Árvore com todos os vértices de G.
6      T_count = 1;
7
8      while(T_count != |V|):
9          min_edge = select_min_edge(G); #Aresta (u, v, peso);
10         v.cor = 'PRETO'; #add v ao conjuntos de visitados;
11         T.adicionar_aresta(u, v, peso);
12         T_count+=1; #Outro vértice acaba de ser visitado;
13     return T;
14
15 def select_min_edge(G:GRAFO(V,E))->EDGE:
16     conjunto_arestas = E;
17
18     menor_peso = INFINITO;
19     menor_aresta = None;
20
21     for aresta{u,v, peso} in conjunto_arestas:
22         #Vértices de conjuntos diferentes
23         if(u.cor == 'PRETO' and v.cor == 'BRANCO'):
24             if aresta{u,v,peso} < menor_peso:
25                 menor_aresta = aresta{u,v,peso};
26                 menor_peso = peso;
27     return menor_aresta;
```

A função *Prim* recebe um Grafo $G(V, E)$ e retorna uma árvore geradora mínima *(que também é um grafo)*. A linha 2 seleciona o primeiro vértice do conjunto V *(poderia se qualquer um)*. Em seguida, adicionamos esse vértice ao conjunto dos vértices visitados *(PRETO)*. A linha 5 define uma árvore T com todos os vértices de G, porém sem nenhuma aresta. A linha 6 controla a quantidade de vértices descobertos, apenas um foi descoberto no início. O laço da linha 8 garante que todos os vértices serão visitados.

A linha 9 chama a função *select_min_edge* a qual irá selecionar a menor aresta do conjunto E tal que essa aresta conecta dois vértices de conjuntos diferentes, ou seja, uma aresta conectando um vértice do conjunto $BRANCO$ a outro vértice do conjunto $PRETO$. A linha 10 irá adicionar o vértice *BRANCO* da aresta encontrada anteriormente ao conjunto dos vértices visitados *PRETO*. Em seguida, na linha 11, a aresta concontrada agora faz parte da nossa árvore.

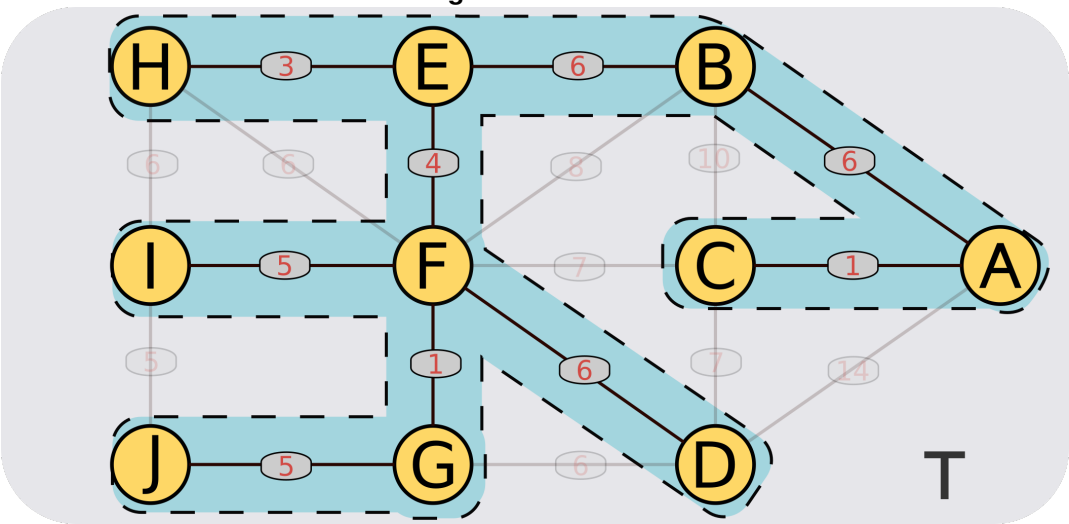
A Função *select_min_edge* é bem sugestiva. Basicamente ela seleciona uma aresta de G tal que essa aresta seja mínima e conecta vértices de conjuntos distintos *(linha 23)*.

Ao chamar a função *Prim* para o grafo da figura 1 temos a seguinte lista de adjacência da árvore mínima T. Logo em seguinda temos uma representação gráfica dessa árvore 2.

```
1 (A) => [('C', 1), ('B', 6)]
```

```
(B) => [('A', 6), ('E', 6)]
(C) => [('A', 1)]
(D) => [('F', 6)]
(E) => [('B', 6), ('H', 3), ('F', 4)]
(F) => [('E', 4), ('G', 1), ('I', 5), ('D', 6)]
(G) => [('F', 1), ('J', 5)]
(H) => [('E', 3)]
(I) => [('F', 5)]
(J) => [('G', 5)]
```

Figura 2. Árvore T



3. Kruskal

O algoritmo de *Kruskal* não é tão diferente de *Prim*. Enquanto Prim usava o conceito de dois conjuntos $BRANCOS$ e $PRETOS$ para selecionar as arestas. O de *Kruskal* irá utilizar o conceito de *florestas*, ou seja, um conjunto de árvores. Inicialmente o conjunto total de árvores é igual ao conjunto total de vértice do grafo $G1$. Sendo assim, cada vértice do grafo é uma árvore separada que irá sendo juntada até formar uma só.

As arestas que irá fazer parte da árvore mínima precisa ser a menor aresta de tal forma que ela conecte duas árvores distintas da nossa floresta, evitando assim um ciclo. Iremos utilizar a estrutura *Union-Find* para trabalhar com os conjuntos de árvores *(floresta)*. Segue abaixo o algoritmo de *Kruskal*.

```
1  def kruskal(G:GRAFO(V,E)) -> GRAFO(V,E):
2      T:GRAFO = G(V, []); #Árvore com todos os vértices de G.
3      conjunto_arestas = E;
4
5      conjunto_arestas.ordenar_crescente();
6
7      #Vetores para controlar os conjuntos das árvores;
8      parente = [0,1,2,3...|V-1|];
9      rank = [0...|V|];
10
11     while(conjunto_arestas):
12         aresta = conjunto_arestas.pop(0);
13         x = find(parente, aresta.u);
14         y = find(parente, aresta.v);
15         if x!=y: #Estão em arvores diferentes ?
16             T.adicionar_aresta(v, u, peso);
17             #Agora v e u pertence a mesma árvore
18             union(parente, rank, x, y);
19     return T;
20
21 def find(parente, i):
22     if(parente[i] == i):
23         return i
24     return find(parente, parente[i]);
25
26 def union(parente, rank, x, y):
27     raizdeX = find(parente, x);
28     raizdeY = find(parente, y);
29     if (rank[raizdeX] < rank[raizdeY]):
30         parenete[raizdeX] = raizdeY;
31     elif rank[raizdeX] > rank[raizdeY]:
32         parente[raizdeY] = raizdeX;
33     else:
34         parente[raizdeY] = raizdeX;
35         rank[raizdeX] += 1;
```

Assim como *prim*, a função *kruskal* recebe um Grafo $G(V, E)$ e retorna uma árvore geradora mínima. Na 2 definimos nossa variável que irá armazenar árvore mínima. A linha 5 define o conjunto de aresta de G de tal forma que essas arestas estão ordenadas em ordem crescente. As linhas 8 e 9 definem dois vetores para controlar a estrutura de dados *Union-Find*, pois iremos trabalhar com conjuntos dijuntos onde cada vértice inicialmente é uma árvore.

O laço da linha 11 será executado enquanto houver arestas para serem exploradas. A linha 12 captura e remove uma aresta, de tal forma que essa possui peso mínimo, visto que houve a ordenação anteriormente. Porém, precisamso saber se essa aresta conecta dois vértices de árvores diferentes da nossa floresta. Inicialmente isso é verdade, visto que cada vértice está em árvores diferentes. A linha 16 adiciona essa aresta a nossa árvore

geradora mínima, visto que ela conecta árvores diferentes. Na linha 18, precisamos juntar esses dois vértices de árvores diferentes em apenas uma árvore. Isso será feito até que todas árvores da nossa floresta se torne uma só.

A função *find*, linha 21, encontra em qual árvore um determinado vértice pertence. Já a função *union*, linha 26, irá unir duas árvores distintas em apenas uma, no final teremos a seguinte árvore geradora mínima.

```
1  (A) => [('C', 1), ('B', 6)]
2  (B) => [('A', 6), ('E', 6)]
3  (C) => [('A', 1)]
4  (D) => [('F', 6)]
5  (E) => [('H', 3), ('F', 4), ('B', 6)]
6  (F) => [('G', 1), ('E', 4), ('I', 5), ('D', 6)]
7  (G) => [('F', 1), ('J', 5)]
8  (H) => [('E', 3)]
9  (I) => [('F', 5)]
10 (J) => [('G', 5)]
```

Note que obtemos a mesma lista de adjacência quando executamos o algoritmo de *Prim*, apenas a ordem da lista de adjacência foi trocada, mas o conjunto de aresta obtido foi o mesmo. Isso ocorre porque ambos os algoritmos são gulosos, ou seja, eles selecionam sempre a menor aresta.

Referências

[Cormen et al. 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

[GOLDBARG 2012] GOLDBARG, E. (2012). *Grafos: conceitos, algoritmos e aplicações*. Elsevier.