Algoritmo de Kruskal

Guilherme Branco, Pedro Henrique, Samuel Pala

[1]Departamento de Ciência da Computação - Universidade de Brasília

1. Objetivo

Este trabalho tem como objetivo o estudo do algoritmo de Kruskal e seu funcionamento, sua ideia geral é encontrar a árvore mínima de um grafo, ou seja a árvore com menor peso entre os vértices e que contenha todos os vértices do grafo. Formalmente o algoritmo de Kruskal é dado como G = (V,E), onde G é o grafo final, V os vértices dos dados de entrada e E as arestas dos vértices.
O algoritmo base eh o seguinte:

```
1  Let G = (V, E) be the given graph, with |V| = n
2  {
3          Start with a graph T = (V,$ \phi$) consisting of only the
               vertices of G and no edges;
4          /* This can be viewed as n connected components, each vertex
               being one connected component */
5          Arrange E in the order of increasing costs;
6          for (i = 1, i$ \leq$n - 1, i++)
7          {
8              Select the next smallest cost edge;
9              if (the edge connects two different connected components)
10             {
11               add the edge to T;
12             }
13         }
14 }
```

2. Implementação

Foi utilizado a linguagem C++ e apenas as funcionalidades padrões da própria linguagem.

2.1. Lista de Adjacência

Como especificado foi utilizada uma lista de adjacência, que funciona basicamente como um mapa, a lista é formada por duas estruturas, *extern_node, intern_node*. O no externo possui um nome para identificação e uma lista para seus vizinhos(*intern_node*), este possui um peso, valor correspondente a aresta, e também um nome identificador.

```
1  typedef struct _in
2  {
3    int nome;
4    int peso;
5    _in(int _nome, int _peso): nome(_nome), peso(_peso){}
6    _in(){}
7  }intern_node;
8  typedef struct _en
9  {
10   int nome;
11   list<intern_node> vizinhos;
12   friend ostream& operator<<(ostream& os,const _en& it)
13   {
14     os << it.nome << " ";
15     for(list<intern_node>::const_iterator i=it.vizinhos.begin();i!=it.
         vizinhos.end();i++)
16     {
17       os << i->nome << "[" << i->peso << "] ";
18     }
19     return os;
20   }
21 }extern_node;
```

2.2. Kruskal

A partir da estrutura anterior fazemos uma conversão para um conjunto de vértices e de arestas, afim de melhor retratar o formato do algoritmo de Kruskal original. Após isto é utilizado o algoritmo descrito na seção objetivos com algumas modificações afim de descartar arestas que formam ciclos no grafo a ser gerado. O funcionamento do algoritmo se dá a partir da ordenação de todas as arestas existentes no grafo por ordem de peso, após isto passa-se de aresta em aresta checando se ela forma algum ciclo entre os vértices, se não formar ela é adicionada ao conjunto final, ao ter-se passado por todos os vértices temos uma arvore geradora mínima que no nosso caso fica guardada na mesma estrutura de lista de adjacência já descrita.

```
typedef struct _edge{
  string nome;
  int peso;

  bool operator<(const _edge& rhs) const
  {
    return nome < rhs.nome;
  }

  bool operator>(const _edge& rhs) const
  {
    return peso > rhs.peso;
  }

  bool operator==(const _edge& rhs) const
  {
    return nome == rhs.nome;
  }
}Edge;
```

3. Dificuldades e soluções

Uma dificuldade foi entender como o algoritmo faz a checagem de ciclos num grafo. A solução disso foi criar vários conjuntos, onde, inicialmente, cada conjunto começa com um vértice diferente. Após adicionarmos as arestas ao grafo final, unimos os conjuntos aos quais cada vértice pertence. Desse modo, se ao tentarmos adicionar uma aresta ao vértice final e os vértices já estiverem no mesmo conjunto, a aresta não é adicionada pois isso significaria um ciclo no grafo final, o que não pode ocorrer. Após isto foi uma mera questão de seguir o funcionamento do algoritmo com poucas dificuldades como a checagem de ciclos entre as arestas.

4. Considerações finais

O projeto foi relevante para compreendermos melhor a dificuldade de se construir um protocolo de roteamento levando em conta custos operacionais, como por exemplo um algoritmo de roteamento afim de encontrar uma rota que possa gerar melhor fluxo de dados. Além disso, tivemos uma oportunidade de aprofundar um pouco mais na linguagem C++ ao usarmos estruturas relativamente novas para nós, como a priority queue e o set.

5. Referências Bibliográficas

[1] Slides da disciplina Teleinformática e Redes 2, ministradas pelo professor Jacir Luiz Bordim, em 2º/2015 na Universidade de Brasília

[2] http://lcm.csa.iisc.ernet.in/dsa/node184.html [1]

[3] http://www.cplusplus.com/doc/tutorial/ [1]

[1] Consultados ao longo do desenvolvimento do projeto para melhor entender o funcionamento das estruturas utilizadas