Trabalho Prático 2

Algoritmos 1

João Antonio Oliveira Pedrosa
Matrícula: 2019006752

[1] Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brasil

joao.pedrosa@dcc.ufmg.br

1. Introdução

O trabalho propõe a implementação de um sistema para auxiliar na melhor escolha para a construção dee ciclovias em uma cidade. O objetivo é, dados todos os pontos importantes da cidade, suas respectivas "atratividades turísticas"e as ligações entre eles, indicar a melhor maneira de conectar todos os pontos importantes da cidade priorizando as rotas de menor custo. Dentre as rotas de menor custo, deve ser escolhida a rota de maior atratividade turística.

2. Modelagem Computacional

A estrutura de entrega das vacinas pode ser traduzida para um grafo, no qual os vértices serão os pontos importantes da cidade e as arestas serão as ligações que existem entre eles.

O problema que precisa ser resolvido pode ser traduzido um problema clássico da teoria dos grafos:

- **Árvore Geradora Mínima:** Dado um grafo não orientado conectado, uma árvore geradora deste grafo é um subgrafo que é uma árvore e que conecta todos os vértices. Um único grafo pode ter diferentes árvores de extensão. Uma árvore geradora mínima é então uma árvore de extensão com peso menor ou igual a cada uma das outras árvores geradoras possíveis. Sendo assim, apenas com essa abstração, nós já resolvemos a primeira parte do problema, as rotas de menor custo são as árvores geradoras mínimas. Basta agora que escolhamos, dentre esta, a de maior atratividade.

Para resolver essa segunda parte, iremos adaptar o algoritmo de Kruskal [Kruskal 1956]. Na implementação original do algoritmo, ordenamos todas as arestas por ordem crescente de custo e percorremos todas, adicionando à nossa árvore toda aresta que não feche um ciclo. Para resolver o problema que queremos, a adaptação que faremos será na parte de ordenação. Ordenarmos normalmente por ordem crescente de custo, porém, quando houve um empate, iremos adicionar primeiramente as arestas de maior atratividade. Com essa simples alteração, o algoritmo resolve o problema.

3. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

3.1. Função *main*

Inicialmente, a main lê alguns valores da entrada e instancia a classe **MST**. Feito isso, ela lê o restante da entrada e insere as arestas do grafo na classe. Por fim, é feita uma chamada do método *Kruskal* e o programa imprime a saída desejada.

3.2. Class Graph

A única classe utilizada na implementação tem o nome de **MST**[1] e possui os seguintes atributos:

- **Do tipo `int`**: O número de vértices no grafo, a soma da atratividade das arestas da AGM e a soma do custo das arestas da AGM.
- **Do tipo `std::vector<int>`**: Um vetor contendo a atratividade de cada vértice, um vetor contendo a quantidade arestas na AGM com ligação a cada vértice do grafo, um vetor indicando a qual conjunto cada vértice pertence e um vetor com o tamanho de cada conjunto.
- **Do tipo `std::vector<tuple<int,int,int>>`**: Vetor contendo informações acerca das arestas presentes na AGM.
- **Do tipo `std::vector<tuple<int,int,int,int>>`**: Vetor contendo informações acerca de todas as arestas do grafo.

Além disso, a classe possui os seguintes métodos:

- **AddEdge(int w, int x, int y)**: Cria uma aresta de x até y com peso w.
- **Find(int x)**: Retorna o representante do conjunto ao qual o elemento x pertence.
- **Union(int x, int y)**: Une o conjunto do elemento x ao conjunto do elemento y.
- **Kruskal()**: Gera a AGM do grafo e calcula os seus atributos.

[1] Sigla para *Minimum Spanning Tree*

3.3. Disjoint Set Union

Uma maneira muito eficiente de implementar o algoritmo de Kruskal é utilizando uma estrutura do tipo União-Busca [Pedrosa 2021]. Nessa estrutura, iremos manter, para cada vértice, a qual conjunto ele pertence e o tamanho de cada um desses conjuntos. Teremos duas funções, uma que retorna o representante do conjunto de um elemento e outra que une os conjuntos de dois elementos.

Algorithm 1 União-Busca

$C \leftarrow$ Vetor contendo, na posição i, um elemento que pertence ao mesmo conjunto que o vértice i. (O representante aponta para si mesmo. Eventualmente todos os vértices apontam para seu representante.)
$S \leftarrow$ Vetor com o tamanho de cada conjunto.

procedure FIND(x)
    if $C[x] = x$ **then**
        return x
    end if
    $C[x] \leftarrow$ **FIND(**$C[x]$**)**
    return $C[x]$
end procedure

procedure UNION(x, y)
    $x \leftarrow$ **FIND(**x**)**
    $y \leftarrow$ **FIND(**y**)**
    if $x = y$ **then**
        return
    end if
    if $S[x] > S[y]$ **then**
        $swap(x, y)$
    end if
    $C[x] \leftarrow y$
    $S[y] \leftarrow S[y] + S[x]$
end procedure

3.4. Algoritmo de Kruskal

O algoritmo de Kruskal percorre todas as arestas do grafo em ordem crescente e adiciona à AGM as arestas que não fecham um ciclo. Usaremos a estrutura de União-Busca para a implementação do Kruskal.

Algorithm 2 Kruskal

INPUT: Lista de Tuplas E representado as arestas do grafo na forma (peso, atratividade, vértice 1, vértice 2).
OUTPUT: Árvore Geradora Mínima F, custo C e atratividade A da árvore.

$C \leftarrow 0$
$A \leftarrow 0$
Sort(E) ▷ Ordenado crescentemente por custo e decrescentemente por atratividade.
for all $[w, a, x, y] \in E$ **do**
  if FIND(x) $\neq$ **FIND**(y) **then**
    UNION(x, y)
    $C \leftarrow C + w$
    $A \leftarrow A + a$
    $F \leftarrow F + (x, y)$
  end if
end for
return $[F, C, A]$

4. Análise de Complexidade

Para essa seção, iremos denotar N como número de pontos importantes da cidade e M como o número de arestas.

- A leitura da entrada, impressão da saída e construção do grafo, têm complexidade $\mathcal{O}(N + M)$.

- As funções da estrutura de **União-Busca** são implementadas com duas otimizações: *PathCompression* e *Small to Large*. Cada uma dessas otimizações, sozinha, faria com que a complexidade de cada uma das funções fosse igual a $\mathcal{O}(\log N)$. [Pedrosa 2021] Entretanto, com as duas otimizações juntas, a complexidade das funções passa a ser o inverso da função de Ackermann $\mathcal{O}(\alpha(N))$. A prova para essa complexidade não é nada trivial e pode ser encontrada em [Tarjan 1975].

- Por fim, o algoritmo de Kruskal possui complexidade igual a $\mathcal{O}(M \log M)$. Como a complexidade das funções **Union** e **Find** é $\mathcal{O}(\alpha(N))$, o que importa realmente para a complexidade do algoritmo é a complexidade da ordenação das arestas.

Sendo assim, separando o algoritmo nessas 3 partes principais, temos uma complexidade de $\mathcal{O}(2 \cdot (N + M) + M \log M) \in \mathcal{O}(N + M \log M)$.

Referências

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

Pedrosa, J. A. O. (2021). MaratonaUFMG: DSU e Kruskal. Disponível em https://www.youtube.com/watch?v=vhPwdX5vaio. Acesso em 22/02/2021.

Tarjan, R. E. (1975). Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225.