TP01 - Ciclovias de Belleville

Rita Rezende Borges de Lima - 2020065317

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

ritaborgesdelima@dcc.ufmg.br

1. Introdução

O problema proposto nesse trabalho consiste em encontrar uma rota sem ciclos que ligue todos os principais pontos turísticos de Belleville de modo que o custo de produção seja o menor possível e o valor turístico agregado o maior. O custo de produção de cada trecho é passado pelo usuário e o valor turístico pode ser calculado somando o valor de cada trecho, esse por sua vez é a soma dos valores dos vértices que o trecho conecta. A rota descoberta será utilizada para a contrução de uma ciclovia de mão dupla para a cidade.

2. Implementação e Modelagem

A modelagem do trabalho foi feita de modo a mapear os pontos turísticos como vértices e as rotas como arestas de um grafo G = (V, E) ponderado e não direcionado como visto na figura 1. O problema proposto pode ser entendido como a busca pela MST, do inglês árvore geradora mínima, que possua a maior atratividade turística.

2.1. A representação do Grafo

O mapa da cidade de Belleville é representado no programa como um vetor de tuplas, onde cada tupla representa um trecho entre dois pontos turísticos, ou seja, uma aresta do grafo. A tupla possui quatro atributos, o custo de produção do trajeto, o valor turístico dos locais ligados pelo trecho e por fim os dois índices dos trechos conectados.

Tabela 1: Tuplas da figura 1

Custo	Valor Turistico	índice v1	índice v2
1500	200	0	1
900	210	0	2
800	210	2	3
3000	210	1	2
2000	200	0	3
900	210	1	3
800	210	2	3

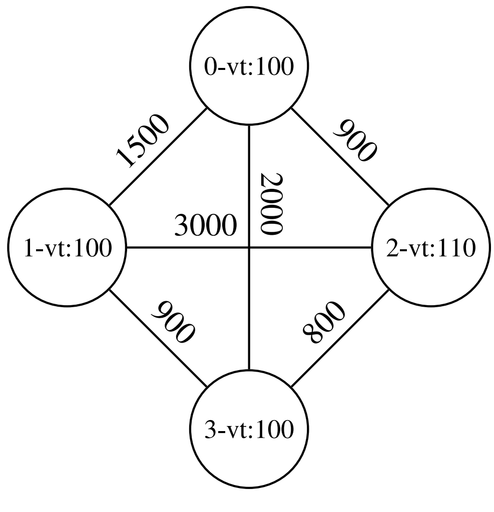

Figura 1: Mapa de Belleville

2.2. A estrutura de Conjuntos Disjuntos Union-Find

Para a implementação do algoritmo de Kruskal, o qual será discutido na sessão seguinte, usaremos uma estrutura de Conjuntos disjuntos. Inicialmente cada vértice do grafo fará parte de um próprio conjunto. A estruturá permitirá duas operações, consultar qual conjunto cada elemento pertence e unir dois conjuntos.

2.3. O algoritmo de Kruskal

Para encontrarmos a árvore geradora mínima, ou seja, a ciclovia da cidade, utilizamos o algoritmo de Kruskal com uma leve modificação, esse segue os seguintes passos:

- Inicialmente cada vértice de nosso grafo será o elemento único de um conjunto na estrutura DSU.

- Ordenamos o vetor de arestas do grafo. O algoritmo original ordena as arestas em função de seus respectivos pesos. No caso desse projeto as arestas são ordenadas em função do custo produção de forma crescente, contudo, caso o custo de dois ou mais trechos sejam iguais virá primeiro na ordenação a aresta de maior valor turístico.

- Cada aresta do grafo é iterada seguindo a ordem pré-estabelecida, caso essa aresta conecte dois conjuntos disjuntos da DSU, conectamos os conjuntos e adicionamos a aresta a MST.

Algorithm 1: Algoritmo de Kruskal

input : Um vetor de arestas do grafo G = (V,E) e a estrutura DSU
output: A árvore geradora mínima de maior valor turístico, T

Ordernar as $|E|$ arestas
custo $\leftarrow$ 0, valor_turismo $\leftarrow$ 0
T $\leftarrow \emptyset$
foreach $a \in$ *arestas* **do**
    (w, t, u, v) $\leftarrow$ a
    if *Find_set(u)* $\neq$ *Find_set(v)* **then**
        T $\leftarrow$ T $\cup$ a
        custo $\leftarrow$ custo + w
        valor_turismo $\leftarrow$ valor_turismo + t
        Union(u, v)
return T

2.3.1. Demonstração de corretude do Algoritmo de Kruskal modificado

Lemma 2.1. *A implementação do algoritmo de Kruskal retorna a árvore geradora mínima de maior valor turístico.*

Proof. Considere G = (V,E) um grafo fortemente conexo ponderado e T as arestas escolhidas no Algoritmo de Kruskal que geram uma MST. Por indução matemática no número de arestas em T.

- ∗ Mostraremos que se T pode gerar uma MST em determinado estágio do algoritmo, este continua assim quando uma nova aresta é adicionada a ele no algoritmo de Kruskal.
- ∗ Quando o algoritmo termina, T conterá uma solução para o problema, uma MST.

Passo Base: Caso T = $\emptyset$, um grafo conectado ponderado sempre tem pelo menos uma árvore geradora mínima, logo ainda é possível chegar a uma MST.

Passo Indutivo: Considere que T ainda pode gerar uma MST antes de processar uma dada aresta A = (u,v). Caso u e v estejam em um mesmo conjunto, T permanece o mesmo de acordo com o algoritmo. Caso u e v estejam em conjuntos distintos, A será adicionada em T. Considere U o conjunto de nós que contém u, note que:

- ∗ U é um subconjunto de V.
- ∗ T é um conjunto de arestas que pode, com adição de outras arestas, ser uma MST e nenhuma aresta de T chega em U (já que as arestas em T ou tem ambos os fins em U, ou nenhum).
- ∗ A é uma aresta de menor custo, e maior valor turístico comparada a outras arestas de mesmo custo. Uma vez que o algoritmo de Kruskal é guloso esse analisa esta aresta somente após examinar arestas de maior prioridade que A.

As três condições acima garantem que uma árvore geradora mínima ainda pode ser formada, portanto, podemos concluir que o T $\cup$ A também pode com a adição de outras arestas se tornar uma MST. Quando o algoritmo termina, T fornece não apenas uma árvore geradora, mas a árvore geradora mínima com maior valor turístico.

■

2.4. Classes implementadas

2.4.1. DSU

A classe de conjuntos disjuntos é utilizada para a representação de forma simples de quais vértices já estão conectados durante a execução do algoritmo de Kruskal. A DSU possui dois atributos, o primeiro é um vetor onde cada posição representa um vértice do grafo e o valor armazenado ali é o indíce do representante de seu conjunto. O outro atributo é um vetor onde cada posição representa o diametro do conjunto ao qual o respectivo elemento faz parte. A classe também possui os seguintes métodos:

- **O construtor** da classe que recebe a quantidade de vértices do grafo e aloca espaço para os dois vetores atributos da classe.
- **A função *int find (int vertex)*** que encontra o representante do conjunto ao qual o vértice passado pertence.
- **O método *void union(int vertice1, int vertice2)*** que verifica se dois vértices estão em um mesmo conjunto e se não estiverem os unimos.

2.4.2. Grafo

A classe grafo representa todos os locais de nosso mapa, suas respectivas rotas e características. Em seus atributos temos um vetor de tuplas representando as arestas do grafo, um vetor para a quantidade de conexões que cada vértice tem dentro da árvore geradora mínima e outro vetor de tuplas que contém as arestas presentes na MST. A classe também possui os seguintes métodos:

- **O construtor** da classe que recebe a quantidade de vértices do grafo e aloca espaço para o vetor de conexões e preenche este com 0s.
- **O método** ***void insert_edge (int vertex1, int vertex2, int weight, int tour_value)*** que recebe como parâmetro os valores de uma aresta e adiciona estes ao vetor de arestas do grafo.
- **O método** ***void print_answer()*** que imprime os valores armazenados no vetor de conexões de cada vértice na mst e as arestas da árvore geradora mínima presente no vetor MST.
- **A função** ***pair<int,int> kruskal (DSU *dsu)*** que implementa o algoritmo descrito na sessão 2.3, e retorna um par contendo o custo total da MST e o valor turístico.

2.5. Entradas e Saídas do programa

As entradas e saídas do programa são as padrão do sistema (stdin).

- **Entradas do programa:** Inicialmente recebemos 2 inteiros, $|V|$, a quantidade de pontos turísticos da cidade e $|E|$, a quantidade de possíveis trechos a serem adicionados. Depois em uma linha única lemos mais $|V|$ inteiros, o valor turístico de cada ponto. Por fim as próximas $|E|$ linhas representam cada um dos possíveis trechos e contém três inteiros, os indíces dos vértices do trecho e seu custo de produção.

- **Saída do programa:** A primeira linha da saída do programa consiste de dois inteiros, o custo total da da rota e seu valor turístico. A segunda linha contém $|V|$ inteiros, a quantidade de conexões na rota que cada um dos vértices possui. Por fim, mais $|V| - 1$ linhas, todos as arestas que a árvore geradora mínima possuirá. Estas linhas tem três inteiros, os índices dos pontos de interesse e o custo do trecho.

3. Análise de Complexidade

3.1. Tempo

Existem três operações que ocorrem no projeto: leitura, o algoritmo de Kruskal e a impressão da saída do programa, dessa forma iremos analisar cada uma dessas partes para depois entendermos o todo.

- **Leitura:** A leitura dependerá exclusivamente da quantidade de pontos de interesse, $|V|$, e da quantidade de possíveis trechos para a ciclovia, $|E|$. Primeiro, a leitura de $|V|$ inteiros ocorre seguida da leitura $|E|$ linhas com três inteiros. Para cada uma destas linhas a função de inserção no grafo é chamada, contudo, por ser uma simples função de adição em um vetor padrão da stl sua complexidade é constante. Dessa forma é possível afirmar que a complexidade da função de leitura é linear e da forma $O(|E| + |V|)$

- **O algoritmo de Kruskal:** A função que implementa o algoritmo de Kruskal depende apenas da quantidade de arestas do grafo, $|E|$. Duas operações são feitas dentro dessa função, a primeira é uma ordenação das arestas utilizando o sort padrão da *STL* que assim como outros algoritmos de ordenação eficientes possui complexidade $O(|E| * log(|E|))$. A segunda operação é o processamento das $|E|$ arestas, onde para cada uma destas podemos ou não utilizar as função union-find da estrutura DSU. Ambas as funções da estrutura Union-Find possuem complexidade da forma $O(\alpha(|E|))$ onde $\alpha(|E|)$ é a função Inversa de Ackermann. Desta meneira o limite superior dessa operação é a complexidade da ordenação do vetor de arestas, ou seja, $O(|E| * log(|E|))$.

- **Impressão:** A impressão executa dois laços, um que itera V vezes imprimindo a quantidade de conexões de cada vértice, e um laço que itera para cada uma das $|V| - 1$ arestas da árvore geradora mínima. Assim, a complexidade da função é linear e de forma $O(|V|)$.

Na função main, chamamos cada uma das operações descritas apenas uma vez, o que ocasiona na complexidade geral: $O(|E|+|V|)+O(|E|*log(|E|))+O(|V|)$. Como o grafo é conectado,$|E| >= (|V| - 1)$ logo a complexidade pode ser reduzida para a forma $O(|E| * log(|E|))$.

3.2. Espaço

No algoritmo todas as estruturas dependem de duas variáveis, a quantidade de pontos de interesse, $|V|$ e a quantidade de possíveis trajetos, $|E|$. Usamos quatro estruturas no projeto:

- **Vetor de representantes da DSU**: seu tamanho é sempre $|V|$, de maneira que sua complexidade espacial é da forma: $\theta(|V|)$.
- **Vetor de tamanho do conjuntos da DSU**: também tem como tamanho $|V|$, de maneira que sua complexidade espacial é da forma: $\theta(|V|)$.
- **Um vetor de tuplas contendo as arestas do grafo:** este possui tamanho $|E|$, de maneira que sua complexidade espacial é da forma: $\theta(|E|)$.
- **Um vetor de tuplas contendo as arestas da árvore geradora mínima:** este possui tamanho $|V|$, de maneira que sua complexidade espacial é da forma: $\theta(|V|)$.
- **Um vetor de tuplas contendo as arestas da árvore geradora mínima:** este possui tamanho $|V| - 1$, de maneira que sua complexidade espacial é da forma: $\theta(|V|)$.

4. Conclusões

A partir da implementação do trabalho foi possível colocar em prática os conhecimentos adquiridos na disciplina a respeito de modelagem de grafos e árvores geradoras mínimas. Por meio do Algoritmo de Kruskal foi possível localizar uma rota passando por todos os pontos de interesse que apresentasse o menor custo possível. E por meio da pequena modificação, escolher dentre as árvores geradoras mínimas, qual tem maior valor turístico.

5. Bibliografia

- Ziviani, N. (2006).Projetos de Algoritmos com Implementações em Java e C++ Editora Cengage
- Kleinberg, Jon; Tardos, Eva. Algorithm Design, Pearson Education India, 2006
- Demonstração Kruskal - Acessado em 17 de fevereiro de 2021
- Aula de DSU - Acessado em 17 de fevereiro de 2021