Trabalho Prático 2 - Algoritmos I

Fernando Eduardo Pinto Moreira - 2019054536 - fernandoepm@ufmg.br

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

1. Introdução

O problema proposto foi implementar um sistema de ciclovias na cidade de Belleville de forma com que todos os pontos de interesse da cidade sejam alcançáveis por meio de tais ciclovias e que o custo de construção das mesmas seja o menor possível, além de que, caso o custo seja o mesmo para duas possíveis ciclovias distintas, haveria de se escolher aquela que agregasse mais atratividade, isto é, a que ligasse pontos mais atrativos da cidade. Com isso, o objetivo deste trabalho é encontrar a ciclovia que se encaixa nessas especificações.

2. Implementação

2.1. Estrutura de Dados

A implementação do programa teve como base um grafo como estrutura de dados. A escolha de tal estrutura se deu devido à especificação do problema, em que existiam pontos de interesse (representados como vértices) e trechos de ciclovias (representados como arestas) e notou-se que um grafo seria a melhor escolha para chegar até a solução de forma mais eficiente.

O grafo então foi implementado utilizando um struct que contém variáveis para a quantidade de pontos, quantidade de trechos, além de um vector de pair, que armazena quatro valores inteiros, sendo eles: o dois pontos de interesse que a aresta faz a ligação, o custo de tal aresta (ou trecho) e a atratividade desta aresta, que é obtida somando a atratividade de cada um desses pontos que a aresta liga.

2.2. Modelagem computacional

A ideia geral do código é, a partir da leitura do usuário de uma certa quantidade de pontos de interesse, quantidade de trechos possíveis, dos valores turísticos de cada ponto e também quais pontos são ligados a quais e com qual custo, obter um caminho

que atinja todos os pontos de interesse e que não possua ciclos, além de minimizar o custo total e maximizar a atratividade dentro do possível (em caso de empate de custo entre dois trechos). Com isso, foi implementado um struct Grafo.

Para calcular o caminho mínimo dentro das especificações do parágrafo anterior, utilizou-se o conceito de árvore geradora mínima (ou *Minimum Spanning Tree - MST*, do inglês). Este retorna uma árvore dentro de um grafo não-direcionado que possui todos os vértices do grafo, de forma com que não haja nenhum ciclo e que o custo total de tal árvore seja o menor possível. Além disso, para implementar o problema proposto, foi necessário uma alteração, que foi feita nos casos em que o peso de duas arestas é igual. Na implementação padrão de uma MST, quando isso ocorre, uma aresta é escolhida sem um certo padrão, mas neste caso do problema, teve que se escolher aquela aresta que possuía a maior atratividade.

Para implementar a MST, foi utilizado o Algoritmo de Kruskal, que retorna os vértices de cada aresta da árvore geradora mínima (além do custo de cada aresta), e também retorna o custo total da MST, exatamente como especifica o problema.

O algoritmo então começa no main lendo a quantidade de pontos, a quantidade de trechos, os valores turísticos de cada ponto e, após isso, lê-se os pontos que são ligados a alguma aresta e o respectivo custo de tal aresta. Assim, as funções "imprimeCustoTotal" e "imprimePontos" são chamadas.

A função "imprimeCustoTotal" imprime o custo total e a atratividade total da MST. Ela começa ordenando todas as arestas do grafo de forma crescente de acordo com o custo da aresta e, caso os custos de duas arestas forem iguais, ordena-se então de forma decrescente de acordo com a atratividade de tais arestas. Após isso, é feita uma iteração em tais arestas ordenadas e, se a aresta selecionada não cria um ciclo com as outras arestas já checadas, essa aresta é adicionada na árvore geradora mínima. Após isso, o peso total e a atratividade são atualizados e são impressos após a saída da iteração. Para determinar e imprimir o grau de cada ponto de interesse (ou vértice) presente na MST, cada ponto de interesse (presente na MST) foi colocado em um vector de pair, sendo que cada par de pontos (um par que forma uma aresta na MST) possui uma contagem de quantas vezes o mesmo aparece e, a partir deste número, consegue-se saber o grau de cada vértice para qualquer entrada do problema.

Já a função "imprimePontos" imprime todos os pontos que possuem um trecho de ciclovia (aresta) os ligando, além do custo dessas arestas. Possui o mesmo funcionamento da função "imprimeCustoTotal" mas, ao invés de imprimir o custo total e a atratividade total, ela imprime todos os pontos que formam uma aresta na MST e os pesos de cada uma dessas arestas.

3. Análise de Complexidade de Tempo

Começaremos a análise pela leitura dos dados do usuário na função "main". Tal leitura é realizada e a função "adicionaTrecho()" é chamada para adicionar uma aresta no grafo, a qual possui custo O(1) por somente atribuir valores. Logo, essas operações de leitura possuem custo constante, ou seja, O(1).

Posteriormente, a função "imprimeCustoTotal()" é chamada, e seu custo é calculado a seguir. Tal função começa inicializando variáveis e um vector de inteiros, ambos com custo constante e, após isso, a função "sort" do C++ é chamada, cujo custo é O(n*log(n)) segundo a documentação da linguagem. Porém percebe-se que a função sort foi alterada para não apenas ordenar de acordo com o custo, mas também de acordo com a atratividade em caso de empate de custo entre dois trechos, mas essa alteração não muda a ordem de complexidade padrão da função "sort". Após isso, é criado um struct "trechosDisjuntos" que possui custo linear (ou O(n)), pois percorre todos os vértices da grafo para assinalar o rank e o pai. Após a criação desse struct, a função "imprimeCustoTotal()" entra em um "for" que itera E vezes, sendo E o número de arestas do grafo. Dentro deste "for", a função merge é chamada. Esta possui custo constante, pois apenas assinala valores a variáveis. Portanto, a ordem de complexidade deste "for" é dada n * O(1) = O(n). Após isso, a função imprime o grau dos vértices do grafo. Isto é feito passando os valores para um pair que guarda o número de vezes que cada vértice aparece e, para guardar esse número, é preciso de dois "for" aninhados que fazem n operações cada, sendo n a quantidade de vértices do grafo. Portanto, essa operação é n*n*O(1) = $O(n^2)$. Logo, a complexidade total da função "imprimeCustoTotal()" é Max(O(n*log(n)), O(n), O(1), $O(n^2)$) = $O(n^2)$.

Após isso, o "main" chama a função "imprimePontos()", que possui o mesmo funcionamento e, portanto, o mesmo custo que a função "imprimeCustoTotal()" até a parte do "for". Após o "for", é chamada novamente a função sort do C++ com uma alteração para ordenar as arestas de forma crescente de acordo com o custo e, caso o custo seja igual, ordena de forma crescente de acordo com o segundo ponto a ser impresso, e isso é feito com apenas um "for" e, portanto, possui ordem de complexidade igual a O(n). Logo, a função "imprimePontos()" possui custo igual a Max(O(n*log(n)), O(n), O(1), O(n)) = O(n*log(n)).

Logo, o custo total do algoritmo é dado por:

$$O(1) + O(n^2) + O(n*\log(n)) = \mathbf{O(n^2)}.$$

4. Pseudo-código

```
struct Grafo:
    Inicializa quantidade de pontos e quantidade de trechos
    Inicializa trechos como um vector de pair de dois pair de inteiros
    adicionaTrecho():
        Adiciona em trechos a aresta com os valores ponto1, ponto2, custo e atratividade
```

struct trechosDisjuntos:
Inicializa todos os vértices tendo rank = 0 e sendo cada vértice pai de si mesmo

encontraPai():
Retorna o pai do elemento u se u for um elemento diferente de seu pai

merge():
Chama a função encontraPai para x e y
if (rank de x é maior que rank de y) então pai de y é igual a x
else pai de x é igual a y
If (rank de x é igual ao rank de y) adicione 1 ao rank de y

imprimeCustoTotal():
Inicializa custoTotal = 0 e atratividadeTotal = 0
Inicializa vector pontos
Ordena os trechos de forma crescente de acordo com o custo e caso o custo de duas arestas seja igual, ordena de forma decrescente de acordo com a atratividade
Cria struct trechosDisjuntos com o parâmetro quantidade de pontos
For each trecho
Chama a função encontraPai com o parâmetro ponto1 e armazena em set_u
Chama a função encontraPai com o parâmetro ponto2 e armazena em set_v
if (set_u = set_v)
Adiciona ponto1 e ponto2 no vector pontos
Soma o custo da aresta atual ao valor do custoTotal
Soma a atratividade da aresta atual ao valor da atratividadeTotal
Chama a função merge com os parâmetros set_u e set_v
Endif
Endfor
Imprime custoTotal e atratividadeTotal
Inicializa o vector de pair vetorRepetidos
Ordena o vector pontos
Inicializa o pair elemRepetidos e o vector de pair repetidos
Conta quantas vezes cada elemento em pontos se repete
Imprime a quantidade de vezes que cada elemento em pontos se repete

imprimePontos():
Ordena os trechos de forma crescente de acordo com o custo e caso o custo de duas arestas seja igual, ordena de forma decrescente de acordo com a atratividade
Inicializa pontos e vectorPontos

```
    Cria struct trechosDisjuntos com o parâmetro quantidade de pontos
    For each trecho
        Chama a função encontraPai com o parâmetro ponto1 e armazena em set_u
        Chama a função encontraPai com o parâmetro ponto2 e armazena em set_v
        if (set_u = set_v)
            Adiciona pontos no vector vectorPontos
            Chama a função merge com os parâmetros set_u e set_v
        Endif
    Endfor
    Ordena arestas de forma crescente de acordo com o custo
    Imprime ponto1, ponto2 e custo de cada aresta

main():
    Lê quantidade de pontos e quantidade de trechos
    Chama Construtor de Grafo
    Declara array que armazena valores turísticos
    For (i de 0 até a quantidade de pontos)
        Lê valores turísticos de cada ponto de interesse
    Endfor
    For (j de 0 até a quantidade de trechos)
        Lê o ponto 1, ponto 2 e o custo de cada trecho
        atratividade = valTuristicos[ponto1] + valTuristicos[ponto2]
        Adiciona cada trecho ao grafo criado
    Endfor
    Chama função imprimeCustoTotal
    Chama função imprimePontos
```

5. Conclusão

Pôde-se notar, depois da implementação do programa, que o problema final foi resolvido e a logística foi realizada com sucesso, isto é, conseguiu-se encontrar uma ciclovia que atingisse todos os pontos de interesse de Belleville de modo que o custo para a construção fosse o menor possível e em que a atratividade fosse maior dentro do possível (em caso de empate de custo entre dois trechos).

Assim, resolveu-se o problema e, com a construção das ciclovias e diminuição da emissão de poluentes e da poluição sonora, a cidade de Belleville certamente poderá esperar um aumento no número de turistas futuramente.