UFMG

Algoritmos 1 – Trabalho Prático 2

Otimização de trajetos

Nome: Arthur Pontes Nader

Matrícula: 2019022294

1. Apresentação

Após o trabalho realizado para alocação de clientes a lojas na Black Friday, agora faz-se necessário melhorar o processo de entrega das compras realizadas online. Para isso, é preciso otimizar o deslocamento de produtos entre lojas da empresa, selecionando os trajetos de tal maneira que todas as lojas fiquem conectadas e o custo total da soma de todos eles seja o menor possível. Para realizar o transporte, a empresa disponibiliza drones, motos e caminhões, sendo que cada um deles tem suas próprias especificidades.

Dessa maneira, esse trabalho teve como objetivo a realização da modelagem e implementação computacional de um programa capaz de otimizar os trajetos que ligam as lojas de tal forma que o custo total de deslocamento seja minimizado. Para solucionar o problema, utilizou-se o algoritmo de Kruskal, que encontra a árvore geradora mínima de um grafo.

2. Modelagem computacional

Pode-se modelar o problema utilizando-se um grafo ponderado e não direcionado. Nesse grafo, os vértices serão as lojas da empresa e cada aresta representará o trajeto entre duas lojas, sendo que o peso associado será a distância entre elas. A seguir, há uma breve descrição das classes e funções implementadas.

- **Classe Vertice:**

  A classe Vertice tem como principal papel o armazenamento dos dados de cada loja, como localização (coordenadas x e y) e número de identificação. Além disso, os objetos dessa classe possuem atributos que serão úteis na implementação da estrutura Union-Find, usada para execução do algoritmo de Kruskal. Os métodos dessa classe apenas retornam os valores dos atributos.

- **Classe Aresta:**

A classe Aresta representa o trajeto entre duas lojas. Assim, ela guarda os dois vértices que compõem a aresta e calcula o peso com base nas respectivas coordenadas. Esse peso corresponderá a distância euclidiana entre as lojas representadas pelos vértices. Tal como a classe Vertice, os métodos implementados apenas retornam o valor de um atributo.

- **Classe Grafo:**

A classe Grafo tem como principais atributos o número de vértices que o grafo terá, além de dois vetores: um que guarda as arestas do grafo e outro que guarda as arestas que irão compor a árvore geradora mínima. Os principais métodos dessa classe são:

Vertice* findSet(Vertice* auxiliar): encontra em uma árvore qual é o representante do conjunto que o vértice passado como parâmetro pertence.

void unionSet(Vertice* v1, Vertice* v2): une os conjuntos dos dois vértices passados como parâmetros.

kruskal(): ordena as arestas pelo menor peso e, usando os métodos "findSet" e "unionSet", constrói a arvore geradora mínima do grafo. Mais detalhes serão explorados na descrição da solução.

std::vector<Aresta*> getArvore(): retorna o vetor que guardas as arestas da árvore geradora mínima.

- **Funcoes:**

Esse arquivo guarda a função necessária para que ordenação do vetor de arestas possa ser feita utilizado o método "sort" da biblioteca padrão.

- **Main:**

A leitura dos dados de cada vértice ocorrerá na "main" do programa. A cada novo vértice lido, gera-se uma nova aresta para cada um dos vértices já existentes. Após isso, instancia-se o grafo e a função responsável pelo algoritmo de Kruskal é chamada. Os dados da árvore geradora mínima são processados e o resultado final é exibido na saída padrão. A memória alocada é liberada e o programa é encerrado.

3. Descrição da solução

A seguir, encontra-se o pseudocódigo resumido das principais funções utilizadas para resolução do problema.

```
findSet(v)
    se v != v.parent
        v.parent = findSet(v.parent)
    retorne v.parent

unionSet(v1, v2):
        se v1.rank > v2.rank
            v2.parent = v1
        senão
            v1.parent = v2
            se v1.rank == v2.rank
                v2.rank = v2.rank + 1

kruskal()
    ordene as aresta pelo menor peso
    i = número de vértices - 1;
    arestaAtual = 0
    enquanto i > 0
        vert1 = findSet(arestas[arestaAtual].v1)
        vert2 = findSet(arestas[arestaAtual].v2)
        se (vert1 == vert2)
            arestaAtual = arestaAtual + 1
            continue
        adicione a aresta à arvore geradora mínima
        unionSet(vert1, vert2)
        arestaAtual = arestaAtual + 1
        i = i - 1
```

Os atributos da classe Vertice necessários para implementação da estrutura Union-Find são: “rank”, que será um limite superior para a altura do vértice na árvore que representará o conjunto e “parent”, um ponteiro para o pai do vértice na árvore. Na instanciação dos vértices, o “rank” é iniciado com o valor 0 e “parent” aponta para o próprio vértice, indicando que no começo cada vértice está sozinho em seu próprio conjunto.

Dessa forma, o método findSet() irá subir pela árvore até encontrar o vértice raiz, que será um representante para o conjunto. Além disso, para aumentar a eficiência, esse método realiza uma compressão de caminho, fazendo com que cada vértice analisado no caminho aponte para o vértice raiz ao final do procedimento.

O método unionSet() é responsável por fazer a união dos conjuntos de dois vértices por meio da alteração do ponteiro “parent”. De forma a manter a árvore balanceada, o novo vértice raiz será aquele de maior “rank”.

O funcionamento do algoritmo de Kruskal se baseia nas propriedades da estrutura Union-Find. Deve-se adicionar as arestas à arvore geradora mínima em ordem de menor peso. Se uma aresta formar um ciclo, ela não deve ser adicionada, sendo que isso pode ser determinado analisando se os vértices constituintes estão no mesmo conjunto. Assim, une-se os conjuntos dos vértices a cada nova aresta adicionada, garantindo dessa maneira que todos os vértice das árvores agora pertencem à mesma componente do grafo. Dessa forma, como a árvore de um grafo com n vértices possui n-1 arestas, ao realizarmos n-1 uniões de conjuntos, forma-se a árvore geradora mínima.

Para solucionar o problema exposto inicialmente, elimina-se da árvore geradora mínima as arestas de maior peso cujo trajeto será realizado por drones. Assim, para cada aresta restante, deve-se analisar se o trajeto será realizado por moto ou caminhão. A soma dos pesos das arestas correspondentes a cada um desses meios de transporte então é multiplicada pelo custo associado, obtendo assim o resultado desejado.

4. Análise da complexidade de tempo

Para realização da análise assintótica de cada método, define-se duas variáveis: “n” será o número de vértices e “m” o número de arestas do grafo. Os principais métodos chamados durante a execução são analisados a seguir, para que se possa concluir sobre a complexidade de tempo da solução final.

- **Métodos da Classe Vertice e Aresta:**

  Os métodos dessas classes realizam apenas retorno de atributos. Isso é feito em tempo constante, portanto, esses métodos são todos O(1).

- **Métodos da Classe Grafo:**

unionSet(): esse método apenas compara o atributo “rank” de dois vértices e atualiza ponteiros. Tudo isso é feito em tempo constante, logo, possui complexidade O(1).

findSet(): já esse método caminha por uma árvore de vértices até encontrar o vértice raiz, ou seja, o representante do conjunto. Devido à lógica de balanceamento na escolha do novo nó raiz feita durante o unionSet(), esse método pode ser executado com complexidade O(log n).

kruskal(): o método começa usando a função de ordenação da biblioteca padrão, que utiliza um algoritmo chamado IntroSort. Assim, para ordenar as arestas, esse algoritmo é O(m log m), mas como $m \leq n^2$, pois há no máximo uma aresta entre dois vértices, o algoritmo também é O(m log n). Em seguida, essas arestas são analisadas em ordem crescente para inserção ou não na árvore geradora mínima. Desse modo, no pior caso, será necessário executar findSet() para as m arestas, fazendo com que a complexidade desse método seja O(m log n).

- **Funcoes:**

A função de comparação necessária para utilização do método “sort” apenas analisa se o peso de uma aresta é menor que a outra. Portanto, esse procedimento é O(1).

- **Main:**

Para leitura dos dados, a cada novo vértice lido, será necessário criar uma nova aresta para todos os outros vértices já existentes. Assim, o seguinte somatório representa o número de arestas criadas:

$$0 + 1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$$

Dessa forma, esse procedimento possui complexidade $O(n^2)$.

- **Conclusão:**

Dessa maneira, a complexidade assintótica do programa como um todo será $O(m \log n + n^2)$.