Trabalho Prático 2

Roberth A. Parreiras[1]

[1]Universidade Federal de Minas Gerais (UFMG)

roberthparreiras@ufmg.br

Resumo. *Esse trabalho prático tem como objetivo resolver o problema de transporte de suprimentos entre as cidades de Minas Específicas, arrazadas pelo deslizamento de terra. Dessa forma, para melhorar o transporte de suprimentos entre as cidades envolvidas, foi desenvolvido um aplicativo chamado* MaxiMini App *que executa uma busca da rota com maior limite de peso possível, entre um par de cidades, e retorna seu maior peso.*

1. Introdução

Como o número de catástrofes estão aumentando de forma alarmante em Minas Específicas, a ONG(Organização Não-Governamental) Luz Vermelha tem trabalhado constantemente para ajudar as vítimas desses desastres; resgatando pessoas em áreas de risco e levando alimento para os mais necessitados.

Devido ao grande número de cidades atingidas pelo deslizamento de terra, ficou inviável buscar manualmente a rota que permite transportar a maior quantidade de suprimentos, sendo muito demorado e recorrente erros de cálculo. Nesse sentido, foi proposto desenvolver um aplicativo chamado *MaxiMini App* que encontra o melhor caminho entre qualquer par de cidades, dado o peso máximo que cada rodovia pode transportar.

O trabalho foi dividido da seguinte forma: a seção 2 apresenta a modelagem em alto nível do aplicativo. A seção 3 apresenta o pseudo-código. A seção 4 apresenta a descrição da solução. A seção 5 apresenta a análise de complexidade. A seção 6 apresenta a demonstração de porque o algoritmo funciona. A seção 7 apresenta a conclusão. Por fim, a seção 8 apresenta as instruções de compilação.

2. Modelagem

Para resolver o problema, pode-se transformar as rotas e cidades em um grafo direcionado com arestas ponderadas. Dessa forma, foi utilizado um algoritmo *dijkstra* adaptado que retorna o maior peso entre um vértice origem e um vértice destino, com os parâmetros *lista de adjacência* do par peso e vértice, vértice origem e vértice destino. Nesse sentido, o algoritmo *dijkstra* adaptado(vou nomeá-lo como *CarregamentoMaximo*), cria uma *fila de prioridade* que vai receber o valor do par peso e vértice, para deixar o vértice de maior prioridade no topo.

Assim, entra-se em um loop e enquanto a *fila de prioridade* não estiver vazia, checa-se qual é o maior peso que uma aresta pode ter entre o par de vértices dados, origem e destino; sendo retornado a posição do vértice destino do vetor que armazena o peso de cada aresta.

3. Pseudo-código de CarregamentoMaximo

```
INICIO_FUNCAO
CarregamentoMaximo(vetor de pares Par, inteiro origem,
inteiro destino)

Cria vetor capacidades;
FOR(i inicia em 0 e enquanto for menor que Par.TAMANHO,
adiciona i = i + 1):
    INSERE em cada posição de capacidades o valor -1;
END_FOR

INSERE na primeira posição de capacidades o valor infinito;

Cria fila de prioridade PQ;
INSERE o valor 0 na posição origem em PQ;

FOR(i inicia em 1 e enquanto for menor que Par.TAMANHO,
adiciona i = i + 1):
    FOR(par peso, aresta P saindo de Par[i]):
        INSERE o peso e vértice na PQ;
    END_FOR
END_FOR

WHILE(!PQ.VAZIO):
    topo recebe PQ.TOPO;
    vértice recebe topo.SEGUNDO;
    PQ.DELETA;

    FOR(par peso, aresta P saindo de Par[vértice]):
        INSERE em peso o valor da comparação
        max(min(capacidades[vértice], P.PRIMEIRO),
        capacidade[P.SEGUNDO]);

        IF(peso MAIOR QUE capacidades[P.SEGUNDO]):
            INSERE na posição capacidades[P.SEGUNDO]
            o valor do peso;

            INSERE NA PQ o peso e P.SEGUNDO;
        END_IF
    END_FOR
END_WHILE

RETORNA a posição destino de capacidades;

END_FUNCAO
```

3.1. Pseudo-código da função main

A função *main* apenas lê o arquivo de entrada, insere em *CarregamentoMaximo* e imprime o resultado das consultas.

4. Descrição da solução

Para resolver o problema, foi utilizado um *dijkstra* adaptado, denominado *CarregamentoMaximo*, que contém uma *fila de prioridade* que armazena o par peso, vértice; sempre mantendo o topo com o par de maior peso. O loop while que faz as comparações entre os pesos de cada vértice, sempre insere na *fila de prioridade* o par peso, vértice que possui maior peso entre a comparação: **sendo π o valor da capacidade de cada cidade, temos**

$max(min(\pi$[vértice atual], peso entre atual e vértice vizinho), π[vértice vizinho]).

Assim, a função retorna o valor da posição do vértice destino contendo o peso máximo entre ele e o vértice origem.

5. Análise de complexidade

5.1. Complexidade de Tempo da função *CarregamentoMaximo*

Sendo o número de vértices igual a **V** e o número de arestas igual a **E**, cria-se inicialmente um vetor de tamanho O(**V**) que armazena os pesos de cada vértice e sendo acessado alguma posição com tempo O(**1**).

Também há a criação inicial de uma *fila de prioridade* da biblioteca padrão *queue* que faz operações de inserção, mostrar topo, testar se é vazia e deleção, sendo **push()**, **top()**, **empty()** e **pop()**, respectivamente. Dessa forma, todas essas operações são de complexidade de tempo O(**1**).

Entretanto, como a *fila de prioridade* sempre vai estar com o maior valor do peso no topo, ele utiliza um *heap* interno que faz as operações **make-heap**, **push-heap** e **pop-heap**. O **make-heap** é a operação inicial que vai criar o *heap* da FP(*fila de prioridade*), com tempo de O(**3 * V**). Já o **push-heap** é executado logo após a FP receber um peso e vértice, pelo **push()**, e, assim, testa qual o maior vértice entre o primeiro e último da FP, com complexidade de tempo O(**log(V)**). Por último, o **pop-heap** é executado quando o elemento do topo da FP é retirado pelo **pop()**, tendo que reorganizar novamente qual o maior vértice entre o primeiro e o último, e chegando na complexidade de tempo O(**2 * log(V)**).

Sendo assim, a *fila de prioridade* tem complexidade total de O(**3 * V) +** O(**log(V)) +** O(**2 * log(V)**), sendo dominado por O(**log(V)**).

Agora, quando entra no loop *while*, há as operações de **top()**, e logo após, **pop()**. Sabendo que grau é o número de vizinhos de **E**, então, dentro do *while* tem um *for* que executa O(**grau * (E)**) caminhamentos nos vizinhos par peso, vértice de cada vértice. Faz isso para achar qual o maior peso entre o par de vértice origem e destino, caminhando nas arestas do vértice atual, que sempre atualiza as posições do vetor que armazena os pesos quando o peso é maior, com o **push()**. Então, como há a execução de O(**grau ***

(E)) caminhamentos, vezes o comando **push()**, a complexidade de tempo é

O(**(grau * (E)) * log(V)**).

Assim, a complexidade de tempo total de *CarregamentoMaximo* é (operações constantes + FP + (loop + push())) =O(**1 + log(V) + ((grau * (E)) * log(V))**), sendo dominado por O(**(grau * (E)) * log(V)**).

5.2. Complexidade de espaço da função *CarregamentoMaximo*

Cria um vetor que armazena os pesos de cada vértice, com a complexidade de espaço o número de vértices, sendo O(**V**).

Também há a criação da *fila de prioridade*, com a chamada do **make-heap** que cria um *heap* a partir do vetor dado, com complexidade de espaço O(**E**).

Assim, a complexidade de espaço total de *CarregamentoMaximo* é (criação de espaços constantes + FP + vetor) = O(**1 + (V) + (E)**), sendo dominado por O((**V**) + (**E**)).

5.3. Complexidade de Tempo da função *Main*

A função *Main* contém um primeiro *for* para receber via terminal todas as informações das linhas do arquivo que contém os vértices, arestas e número de consultas. É criado uma *lista de adjacência* de cada vértice, sendo utilizado o comando **push-back()** para armazenar os pesos e vértices. Essa operação é O(**1**). Dessa forma, essa complexidade de tempo é O(**E**).

Logo após, há um segundo for que imprime na tela os resultados das consultas feitas na função *CarregamentoMaximo*. Assim, a complexidade de *CarregamentoMaximo* vezes o número de consultas, sendo o número de consultas **Q**, é O(**(Q * (grau * (E)) * log(V))**).

Portanto, a complexidade de tempo total de *Main* é (operações constantes + primeiro *for* + segundo *for*) = O(**1 + E + (Q * (grau * (E)) * log(V))**), sendo dominado por O(**(Q * (grau * (E)) * log(V))**).

5.4. Complexidade de Espaço da função *Main*

A função *Main* cria uma *lista de adjacência* de tamanho O(**V + E**), sendo **V** vértices e **E** arestas. O segundo loop, que é o loop que imprime o valor de cada consulta, tem a criação de um espaço adicional por causa da função *CarregamentoMaximo*, de complexidade O(**V + E**).

Assim, a complexidade total é a criação da (*lista de adjacência* + o espaço adicional da função *CarregamentoMaximo*) = O(**(V + E) + (V + E)**).

5.5. Complexidade Total do programa

A Complexidade de Tempo Total do programa é (*CarregamentoMaximo* + *main*) = O((**(grau * (E)) * log(V)**) + (**(Q * (grau * (E)) * log(V)**)).

A Complexidade de Espaço Total do programa é (*CarregamentoMaximo* + *main*) = O((**(V) + (E)**) + (**(V + E) + (V + E)**))

6. Demonstração

Sendo π o valor da capacidade de cada vértice, S o conjunto dos vértices explorados e d[*u*] = capacidade máxima de um vértice do caminho $s \rightarrow u$.

Assim, para o vértice $u \in S$, então d[*u*] = capacidade máxima mínima de um vértice do caminho $s \rightarrow u$.
Prova: [por indução]

$CASO\ BASE$: S = 1, pois S = s e d[*s*] = ∞.

$HIPÓTESE\ DE\ INDUÇÃO$: Assuma que para qualquer vértice $\in S$ tem o caminho contendo a capacidade máxima mínima, com todas as iterações sendo corretas.

. A iteração do caso base é d[*s*] = ∞.
. A iteração atual é d[*u*] = $max(min(\pi$[vértice atual], peso entre atual e u), π[u])
. Suponha que *u* seja o vértice destino e que, após as iterações, d[*u*] tem a máxima mínima capacidade. Então o caminho $s \rightarrow u$ é o caminho ótimo.
. Dessa forma, devemos mostrar que para qualquer outro caminho entre $s \rightarrow u$, eles serão $\leq d$[*u*]. Assim, para qualquer outro caminho em S, que deve caminhar por outros vértices(vamos chamar esse outro caminho de *V*), ele terá a capacidade máxima mínima $\leq d$[*u*].
. Então, para qualquer caminho $s \rightarrow u$ em *V*, após as iterações, será d[*v*] $\leq d$[*u*], sendo d[*v*] a capacidade máxima mínima de *V*.
. Visto isso, qualquer outro caminho $s \rightarrow u$, em *V*, terá o limite máximo mínimo em d[*v*], nunca sendo maior que d[*u*].
. Portanto, d[*u*] é capacidade máxima mínima ótima para o caminho $s \rightarrow u$. **Q.E.D**.

7. Conclusão

Esse trabalho prático tem como objetivo implementar um algoritmo que encontra a rota com capacidade máxima mínima ótima para transportar suprimentos aos necessitados. Nesse sentido, foi criado um aplicativo chamado *MaxiMini App*, que encontra essa rota via um *dijkstra* adaptado, com uma complexidade de tempo e espaço aceitável. Finalmente, foi demonstrado que esse algoritmo funciona para todo tipo de entrada de arquivo.

8. Instrução de compilação

A configuração abaixo foi utilizada para testar o programa:
- Windows 10 Home;
- Compilador Git bash;
- Editor VScode;
- Processador Intel core i5-8265 1.60 GHz;
- Mémoria RAM 8 GB;

Para compilar o programa, deve-se digitar no terminal ”g++ -c main.cpp”.
Para cria um executável do programa, deve-se digitar no terminal ”g++ -o tp02.exe main.cpp”.