Trabalho Prático 2
Algoritmos I

Luís Felipe Ramos Ferreira
2019022553

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

lframos_ferreira@outlook.com

1. Introdução

A proposta do trabalho prático 2, da disciplina de Algoritmos, foi implementar um algoritmo que recebe, em sua entrada, uma descrição de uma malha rodoviária e um par de cidades e, a partir desses dados, calcular o valor do caminho que possua o gargalo máximo entre a primeira cidade até a segunda.

Os detalhes da implementação e modelagem deste problema, assim como a análise de complexidade das funções utilizadas e as instruções de compilação, estão descritas ao longo desta documentação.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection, no sistema operacional Windows.

2.1. Modelagem computacional

O problema dado pelo enunciado do trabalho prático pode ser facilmente interpretado como uma problema de modelagem e análise de uma estrutura de um grafo direcionado e ponderado.

Mais especificamente, para modelar o que foi proposto, foi considerado que cada cidade da malha rodoviária seria um vértice da estrutura, e as rodovias que formam os caminhos de uma cidade a outra seriam as arestas. Nesse mesmo sentido, os pesos das arestas representam o peso máximo que é possível carregar na travessia de cada uma delas.

Frisa-se também que, conforme explicitado pelo enunciado, podem não existir caminhos entre duas cidades da malha rodoviária e, caso exista, não necessariamente existe também um caminho de mão dupla, isto é, que permita ir e voltar entre essas duas cidades. Ademais, no caso específico de existir este caminho de mão dupla, o peso máximo suportado pela aresta em um sentido não necessariamente é igual ao peso máximo suportado no sentido oposto.

Para compreender melhor a questão e a interpretação, segue abaixo um exemplo de como o problema seria analisado.

Vamos supor que, da cidade C1 é possível ir até a cidade C2, por meio de uma rodovia que suporta até 1000 unidades de peso, mas não existe um caminho de C2 até C1. Também, de C2, é possível ir até uma cidade C3 através de uma rodovia de peso

máximo 2000, enquanto de C3 até C2 existe uma rodovia de peso máximo 1500. O grafo que representa essa instância do problema, conforme a interpretação supracitada, seria o seguinte:

Como pode-se ver, o grafo representa claramente as conexões da malha rodoviária, conforme as especificações apresentadas no enunciado.

Partindo desse ponto, é necessário agora entender como calcular o resultado desejado, isto é, o caminho entre dois vértices S e T de modo que o peso da menor aresta desse caminho seja o maior possível dentre todos os caminhos (caminho de gargalo máximo). A princípio, quando li o enunciado, ficou claro a semelhança que ele possui com dois algoritmos estudados durante a disciplina, sendo eles o algoritmo de Dijkstra para o menor caminho entre dois vértices e o algoritmo de fluxo máximo de Ford-Fulkerson.

A semelhança com o algoritmo de Dijkstra é nítida. Ambos problemas analisam valores de arestas no caminho entre dois vértices e encontram aquele que se adequa com as especificações. No caso do algoritmo de fluxo máximo, a semelhança vem do momento em que, ao usar um algoritmo de busca para encontrar um caminho qualquer entre os vértices desejados, deseja-se obter a aresta de menor capacidade/valor no caminho.

No entanto, após uma análise mais profunda do que se pede, seguida de pesquisas sobre outros algoritmos de análises de grafos, cheguei à conclusão de que o algoritmo de Ford Fulkerson não me seria útil aqui, e que toda a lógica por trás do algoritmo de Dijkstra seria muito mais importante para a resolução desse problema. Para entender melhor, reitera-se o seguinte fato (simplificado e sem muitos detalhes) acerca do algoritmo de menor caminho: mantêm-se guardado o valor da distância atual conhecida do vértice de origem S até o vértice de destino T. A cada iteração, se é encontrado um novo caminho para o vértice T, que possua uma distância menor que o atual, o valor atual de distância é atualizado. Essa condição é conhecida como a condição de ‘relaxamento’.

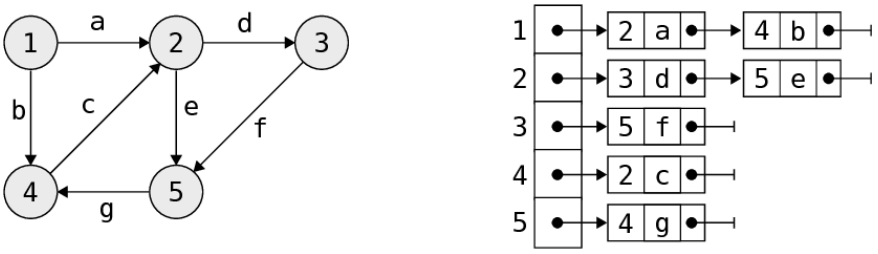
Tendo isso em mente, fica claro que o problema enunciado pode ser resolvido com uma abordagem similar. Se mantivermos o valor do caminho mais largo entre S e T armazenado e, a cada iteração,checar se um caminho novo gerado possui um gargalo maior, iremos chegar ao resultado desejado. Portanto, em suma, basta uma pequena alteração na condição de ‘relaxamento’ no algoritmo de Dijkstra para que possamos resolver o problema.

A implementação dessa resolução assim como mais alguns detalhes acerca de como ela foi escolhida e feita serão descritos a seguir. As fontes que me inspiraram e me ajudaram a chegar à conclusão acima estão dispostas na seção de referências bibliográficas ao final dessa documentação, com os devidos créditos dados.

2.2. Estruturas de Dados

De maneira geral, a biblioteca padrão da linguagem C++ foi muito útil para a representação do problema apresentado. As seguintes estruturas disponibilizadas pela biblioteca foram utilizadas na implementação: *vector, priority_queue, pair, list*. O porquê de seus usos e de que forma foram aproveitados estão detalhados a seguir.

Para a modelagem do problema, como foi explicado na seção de Modelagem Computacional, foi utilizada a estrutura de um grafo para representar a malha rodoviária do problema. A maneira de representar esse grafo, em código, foi por meio de uma lista de adjacências, uma vez que essa representação permite uma maior facilidade na checagem de quais vértices são os "vizinhos" de um determinado outro vértice. Dessa forma, as estruturas *vector*, *list* e *pair* foram usadas, de modo que o grafo seria um vetor de listas que contêm pares de identificador do vértice(*int*) e peso da aresta que liga os vértices(*int*). Segue abaixo uma imagem ilustrativa como exemplo não relacionado a esse trabalho em específico para compreender melhor essa implementação.



Grafo direcionado Lista de adjacência

Tendo essa representação em mente, foi criada uma função chamada *'widestPathAlgorithm'*, que calcula o valor do gargalo máximo de caminho entre o vértice de origem desejado e todos os outros do grafo, e retorna esse valor em uma estrutura de um *vector*. Mas porque calcular esse gargalo para todos os possíveis vértices de destino no grafo, se em cada consulta queremos saber o valor para apenas um vértice de destino?

A princípio, durante a execução do algoritmo, julgando um 'pior caso' em que todos os vértices do grafo devem ser visitados para que se obtenha o valor procurado para o vértice de destino da consulta, o caminho de gargalo máximo é calculado para cada um desses vértices. Como não se sabe quantas consultas serão feitas e nem se serão feitas consultas que partem do mesmo vértice de origem, podemos armazenar os valores já calculados anteriormente, com o objetivo de evitar o custo de realizar cálculos computacionais repetidos.

Agora, vamos falar mais especificamente de como a função funciona. Em primeiro lugar, uma coisa deve ficar clara para o leitor. Segundo o enunciado, os N vértices de grafo seguem um padrão sequencial de número identificador de 1 até N. No entanto, como é sabido, a linguagem C++ possui uma indexação em suas estruturas de dados que começam no valor 0. Para evitar erros de acesso a posições inválidas na memória, ao ler o identificador de um vértice fazemos uma manipulação para esses valores se adequarem à implementação. Em outras palavras, seria algo como: 1 'vira' 0, 2 'vira' 1, 3 'vira' 2, etc. Essa mudança não gera diferença nenhuma ao código e é feita apenas para facilitar a manipulação dos valores obtidos.

Para seu funcionamento, existem duas estruturas de *vector* auxiliares na função. A primeira delas é a '*widestPathVector'*, que armazena o valor dos gargalos máximos descobertos no caminho para cada vértice, e '*visited*', que basicamente armazena verdadeiro se o vértice tiver sido visitado, falso caso contrário. Uma *priority_queue*

também é inicializada, a qual irá conter pares de inteiros do tipo (identificador do vértice, valor do gargalo encontrado até esse vértice).

O ponto principal do algoritmo está justamente dentro dessa fila de prioridades. Podemos dizer, com certeza, que na primeira vez que retiramos um vértice da fila de prioridades, o valor do gargalo junto a ele é o gargalo máximo possível no grafo. Mas por que?

Essa inferência vem justamente do fato da fila de prioridades ordenar os pares inseridos nela de forma não ascendente pelos gargalos armazenados. Vamos supor que após retirar um determinado vértice S da fila de prioridades junto de seu gargalo, exista um outro caminho que possua um gargalo maior até S. Ora, se esse caminho possui um gargalo maior, ele, assim como os passos seguidos para se chegar até ele, deveriam ter sido retirados antes da fila de prioridades. Assim, podemos confirmar a inferência anterior.

Nesse ponto entra a utilidade do vetor de vértices visitados. Após retirar um vértice da fila de prioridades, marcamos esse vértice como visitado. Em todos os outros momentos do laço, quando obtemos o valor de um novo vértice, podemos checar se ele já foi visitado ou não. Se a resposta for sim, podemos ignorá-lo, já que temos, com certeza, o valor do caminho de gargalo máximo até ele.

Dessa forma, o seguinte pseudocódigo, feio em LaTeX, representa o funcionamento do código, conforme as descrições acima apresentadas.

Algorithm 1 Retorna um vetor com os gargalos máximos entre src e todos os outros vértices do grafo

```
1:  function WIDESTPATHALGORITHM(graph, src)
2:      widestPathVector[vertex] ← INT_MIN ∀ vertex ≠ src
3:      widestPathVector[src] ← INT_MAX
4:      visited[vertex] ← 0 ∀ vertex ≠ src
5:      priority_queue ← (src, INT_MAX)
6:      while priority_queue not empty do
7:          currVertex ← priority_queue top first
8:          currVertexWidestPath ← priority_queue top second
9:          priority_queue pop
10:         if visited[currVertex] then continue
11:         visited[currVertex] ← 1
12:         widestPathVector ← currVertexWidestPath
13:         for adjVertex in graph[currVertex] do
14:             if visited[adjVertex] then continue
15:             widestPathAdjVertex ← min(widestPathVector[currVertex], adjVertexPathWeight)
16:             priority_queue ← (adjVertex, widestPathAdjVertex)
17:
18:     return widestPathVector
```

Por fim, na função principal, temos um *vector* de *vector*, que irá armazenar os valores já obtidos de gargalos máximos para cada vértice como origem, conforme explicado anteriormente, tendo como objetivo evitar a repetição de algoritmos dos quais já obtivemos o valor desejado.

3. Análise de Complexidade de Tempo

Função 'operator()' (struct Compare): essa função realiza apenas operações constantes, portanto sua complexidade de tempo é constante.

Função ‘widestPathAlgorithm’: essa função irá percorrer todas as arestas do grafo passado com parâmetro, portanto sua complexidade irá possuir um fator E, onde E é esse número de arestas (*edges*). Para cada um dos vértices, seus vizinhos são checados desde que estes vizinhos ainda não tenham sido visitados anteriormente, e, nesse caso, adicionados na fila de prioridades. Como sabemos e conforme a documentação da biblioteca padrão, a complexidade de inserção em uma fila de prioridade é O(log n). Neste caso, ‘n’ possui o valor do número de vértices no grafo. Portanto, a complexidade possui um fator logV. Assim, a complexidade final do algoritmo é O(ElogV).

A função principal é responsável por ler os valores de entradas para o programa. Dessa forma, essa leitura é linear para o tamanho da malha rodoviária, isto é, número de vértices e arestas do grafo. Para cada uma das N consultas realizadas pelo programa, julgando um pior caso em que cada consulta parte de um vértice de origem diferente, teremos uma complexidade de tempo total para execução do código de O(NElogV).

4. Referências Bibliográficas

https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1013.pdf

http://www2.dcc.ufmg.br/livros/algoritmos-edicao2/cap7/transp/completo4/cap7.pdf

https://www.geeksforgeeks.org/widest-path-problem-practical-application-of-dijkstras-algorithm/

https://cs.stackexchange.com/questions/18522/widest-path-algorithm-steps

Livro da disciplina: Algoritmos: Teoria e Prática.

5. Compilação e execução

- Acesse o diretório TP
- No terminal, utilize o seguinte comando:
  g++ main.cpp -o main
- Para utilizar o executável gerado, utilize no terminal o comando:
  ./main
  seguido da entrada desejada para o programa.
- Exemplo de execução e saída esperada:
  g++ main.cpp -o main
  ./main
  9 14 1
  1 2 2500
  2 3 4000
  1 4 3000
  2 5 3000
  5 2 1500
  3 6 1200
  4 5 1000
  5 6 1500
  4 7 1600
  5 8 2000
  8 5 3000
  6 9 1700
  7 8 1000
  8 9 1500
  1 9

  1500