TP 02 - Trabalho Prático 2
Algoritmos I

Igor Lacerda Faria da Silva

igorlfs@ufmg.br

1 Introdução

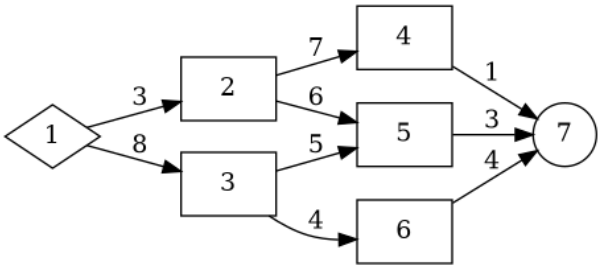
O TP foi desenvolvido em C++, em ambiente Linux (versão do *kernel:* 5.18.0), utilizando algumas estruturas de dados como `std::vector`, `std::list` e `std::priority_queue`. Em termos de boas práticas (formatação, *case* de variáveis e afins), foram as usadas as ferramentas `clang-format` e `clang-tidy` para garantir um código limpo e consistente. Foram escritos testes de unidade para todas as funções do código, usando a biblioteca `<gtest>`. Os testes podem ser compilados e executados com o comando `make test` (assumindo-se que a biblioteca e o make estejam instalados no sistema).

2 Modelagem Computational do Problema

Em suma, o programa lê uma malha rodoviária e um conjunto de buscas, e encontra o caminho com o maior gargalo (peso suportado) em cada consulta. A leitura da entrada é dividida em 3 etapas: leitura dos parâmetros (número de cidades, número de vias e número de consultas), leitura das vias que compõem a malha (origem, destino e peso suportado) e leitura de consultas (origem e destino). Em seguida, é calculado o caminho desejado para cada consulta.

As consultas são armazenadas como uma lista de pares ordenados: (partida, chegada). Essa distinção se faz importante porque as vias não são, necessariamente, de mão dupla: pode ser o caso de uma via só "funcionar" em um sentido, ou ainda, ter pesos diferentes no mesmo trajeto, (considerando-se sentidos diferentes). Além disso, não é menos que natural usar um grafo direcionado e ponderado para representar a estrutura da malha.

Para realizar o cálculo do maior gargalo foi usado um algoritmo que é uma pequena variação do algoritmo de Dijkstra, que ao invés de considerar a menor distância até um dado vértice, considera justamente o gargalo. Para ilustrar o funcionamento desse algoritmo guloso, será discutido um exemplo de execução. Considere o seguinte grafo:



Cada vértice possui possui um número de identificação (em consonância com a forma como o problema é implementado) e cada aresta possui um peso associado. O vértice de partida do exemplo de execução a ser discutido está indicado por um losango e o vértice de destino é indicado por um círculo. Nessa instância, é simples resolver o problema mentalmente: o esperado é que a resposta seja 4.

A princípio, todos os vértices tem o gargalo definido como $-\infty$, menos o vértice de partida que possui gargalo ∞. Inicialmente é calculado o real gargalo para os vizinhos da fonte. Em seguida, é decidido o "melhor caminho" para proceder com base no gargalo da etapa anterior (no código isso é feito com um *heap*). No caso, como o vértice 2 possui menor gargalo (3, contra 8 do vértice 3), ele é escolhido. Olhamos então o gargalo de seus vizinhos. O gargalo de 4, por exemplo, ainda é 3, pois 7 é maior que 3 (e maior que $-\infty$). Similarmente, o vértice 5 tem gargalo 3.

A essa altura, o vértice de menor gargalo que "ainda não foi explorado" é o 4, com gargalo 3, então ele será o próximo a ser explorado. O único vértice alcançável a partir deste vértice é o destino, que por enquanto fica com gargalo igual a 1. Mas o destino não tem filhos, então é o vértice 5, de gargalo 3 que é o próximo a ser analisado. Como ele leva ao destino mais de forma mais *espaçosa*, o gargalo do destino é atualizado para 3.

Nesse momento, o único vértice na "pilha" é o 3. O gargalo de 5 é atualizado, e o de 6 definido como 4, que é obviamente menor, então será o próximo: o destino é atualizado para o valor esperado. A execução termina, voltando no 5 e concluindo que de fato 3 é menor que 4.

A *ideia* de execução do algoritmo é essa, resta então implementar isso de uma forma eficiente, especialmente a etapa de encontrar o menor gargalo atual. Estas e outras questões serão discutidas na próxima sessão.

3 Estruturas de Dados e Algoritmos

A principal estrutura de dados utilizada foi o grafo, implementado na classe `Graph` como um vetor de listas de pares de inteiros. Além dessa classe, existe a `Input`, que lida com a leitura da entrada (criar uma classe para isso facilita nos testes de unidade). Na classe `Input`, é adicionalmente usada uma lista para armazenar as consultas. A outra estrutura de dados utilizada no TP foi a lista de prioridades da biblioteca padrão para escolher o vértice adequado na escolha do caminho.

O principal algoritmo implementado foi a seleção do caminho *mais largo*, como discutido anteriormente. No entanto, algumas mudanças foram feitas na implementação, comparado à seção anterior: o *heap* escolhe o MAIOR gargalo[1]. Uma vez que se tem a ideia intuitiva do algoritmo, a implementação fica bem mais simples. Atenção especial deve ser dada, no entanto, ao uso do *heap* e a comparação que determina o gargalo: o máximo entre o gargalo atual do vértice e o mínimo entre o gargalo do "pai" atual e o peso da aresta atual. Esse é o *cerne* da "gulosidade" do algoritmo.

A seguir, é descrita a implementação dos algoritmos em pseudo-código:

Algorithm 1 Retorna o maior gargalo de qualquer caminho entre src e tgt

```
1:  function widestPath(src, tgt)
2:      Wi ← −∞∀ vertex ≠ src
3:      heap ← (0, src)
4:      while heap != empty do
5:          newSrc ← heap top
6:          for vertex in graph do
7:              bottleneck ← max(Wi[vertex], min(Wi[newSrc], vertexWeight))
8:              if bottleneck > Wi[vertex] then
9:                  Wi[vertex] ← bottleneck
10:                 heap ← (bottleneck, vertex)
11:     return Widest[tgt]
```

4 Análise de Complexidade de Tempo

A leitura dos parâmetros tem complexidade constante. A leitura das vias é $\Theta(n)$ no número de vias, e a leitura de consultas é o $\Theta(m)$ no número de consultas. No cálculo do *widest path*, cada vértice é visitado uma única vez, e para cada vértice seus vizinhos são visitados. Desse modo, todas as arestas são visitadas uma única vez. Para cada aresta, são feitos 2 ajustes no *heap* em tempo que é proporcional, no máximo a $\log K$, em que K é o número de cidades do mapa. Portanto a complexidade dessa função é $n \log K$.

Analisando o programa como um todo, tendo em vista que são feitas m consultas de custo $n \log K$ e nenhuma etapa excede esse custo, o programa tem complexidade $mn \log K$.

5 Compilação

```
g++ src/graph.cpp -c -Wall -std=c++17 -Ilib -o obj/graph.o
g++ src/input.cpp -c -Wall -std=c++17 -Ilib -o obj/input.o
g++ src/main.cpp -c -Wall -std=c++17 -Ilib -o obj/main.o
g++ obj/graph.o obj/input.o obj/main.o -o tp2
```

[1] Eu percebi que por algum motivo a execução é muito mais rápida dessa maneira.