

TP 02 - Trabalho Prático 02

Algoritmos I

Daniel Oliveira Nascimento

1. Modelagem computacional do problema

Dado um conjunto de cidades interligadas por rodovias — as quais possuem um peso associado a cada uma e podem ser de mão única ou de mão dupla — o problema consiste em identificar a capacidade máxima de carga que pode ser enviada entre pares de cidades. Diante desse contexto, o problema foi modelado como um grafo ponderado e direcionado, no qual: os vértices são as cidades; as arestas são as rodovias que saem de uma cidade e vão para outra; e o peso de cada aresta define a carga que pode ser enviada entre as duas cidades ligadas por ela.

Uma vez modelado em um grafo, trata-se de um *widest path problem* ou *maximum capacity path problem*, o qual pode ser resolvido adaptando algum algoritmo de *shortest path* para se basear nos gargalos dos caminhos. Tendo isso em vista, foi utilizada uma versão modificada do algoritmo de *Dijkstra* no qual, dada uma consulta entre uma cidade u e uma cidade v , são calculados os *widest paths* entre a cidade u e todas as outras cidades e, por fim, é retornado o gargalo (menor peso) do caminho entre u e v .

2. Estruturas de dados e algoritmos

Nesta seção serão descritos os principais algoritmos e TADs utilizados na elaboração da solução.

a. TAD Neighbor

Uma estrutura de dados simples utilizada para armazenar cada cidade vizinha de uma outra cidade. É responsável por armazenar o ID da cidade vizinha, bem como o peso associado à rodovia que sai de uma determinada cidade à cidade vizinha.

b. TAD Graph

Construída para transformar o mapa das cidades em um grafo utilizando lista de adjacências. O formato de lista foi escolhido por ser mais performático do que o formato de matriz no que tange a percorrer todas as arestas do grafo.

Vale destacar que a lista de adjacências utiliza a *TAD Neighbor*, pois a lista de adjacência de uma cidade é exatamente as cidades vizinhas à cidade em questão.

Essa TAD também armazena o método responsável por calcular o peso máximo que pode ser transportado entre duas cidades.

c. Algoritmo maxWeight

Para obter o maior peso possível a ser transportado entre duas cidades, foi implementado um algoritmo baseado em *Dijkstra* com algumas modificações:

maxWeight:

para cada nó do grafo:

marca o peso associado ao nó como $-\infty$;

adiciona o nó a uma lista;

marca o peso associado à fonte como $+\infty$;

enquanto a lista possuir elementos:

u = vértice da lista com maior peso;

se o peso de u for $-\infty$:

não há mais vértices que podem ser

encontrados a partir da fonte

então encerra o loop;

para cada vizinho de u :

obtem o maior peso possível (gargalo) que pode
ser levado de u até o vizinho atual;

se esse peso for maior que o peso atual associado
ao vizinho, então atualiza o peso do vizinho;

ao final do loop, todos os pesos foram obtidos, então retorna
o peso do destino, que é o gargalo entre a fonte e o destino.

O motivo pelo qual o algoritmo de *Dijkstra* modificado funciona é parecido com a própria prova de corretude da versão original, porém com algumas modificações:

Invariante: para cada vértice $u \in S$: $pi[u]$ = maior capacidade mínima do caminho $s \rightarrow u$.

Prova por indução:

Caso base: $|S| = 1$ é fácil, pois o único elemento do conjunto é a fonte ($S = \{s\}$) e $pi[s] = \infty$.

Hipótese indutiva: Assumindo que o algoritmo funciona para $|S| \geq 1$:

- a) Dado v como o vértice com o maior $pi[v]$ que ainda não foi atingido pela fonte, então o caminho é traçado de s até v e $pi[v]$ é tomado como a maior capacidade mínima do caminho $s \rightarrow v$;
- b) Sendo assim, é necessário provar que qualquer outro caminho $s \rightarrow v$ terá um valor menor que $pi[v]$; tal fato é verdadeiro, uma vez que $pi[v]$ é o maior valor possível para um vértice que ainda não foi atingido pela fonte (vide “a”). Ou seja, qualquer que seja $u \neq v$, $pi[u] \leq pi[v]$;
- c) Uma vez qualquer outra possível solução usaria um $pi[u]$ que é menor que o $pi[v]$ obtido, então $pi[v]$ é de fato a maior capacidade mínima do caminho $s \rightarrow v$.

d. Solução

Uma vez detalhados os algoritmos e as TADs, é possível compreender o fluxo total da solução:

solution:

```
ler número de cidades (N);
ler número de rodovias (M);
ler número de consultas (Q);
verificar se as leituras condizem com os limites impostos;
criar um grafo de N vértices;
para cada aresta (M):
    ler um par de cidades u e v, bem como o peso w associado à rodovia
    que sai de u e vai para v;
    verificar se as leituras condizem com os limites impostos;
    adicionar a rodovia como uma aresta no grafo;
para cada consulta (Q):
    ler um par de cidades u e v:
    verificar se as leituras condizem com os limites impostos;
    calcular o peso máximo que pode ser enviado entre as duas
    cidades utilizando o algoritmo maxWeight;
```

3. Análise de complexidade de tempo assintótica

Dado que N é o número de cidades, M o número de rodovias e Q o número de consultas:

Para ler todos os pares de cidades (rodovias) e montar o grafo, realizam-se M iterações, portanto este passo é $\Theta(M)$.

Para ler todas as consultas, executam-se Q iterações. Em cada uma, no entanto, é computado o algoritmo *maxWeight*. Este, por sua vez, executa N iterações para inicializar as variáveis auxiliares ($\Theta(N)$) e até mais N iterações para computar os caminhos dentro da fila de prioridade. Neste segundo passo, em cada iteração: podem ocorrer até N iterações para obter o elemento com maior peso na fila ($O(N)$) e até N iterações para percorrer os vizinhos desse elemento ($O(N)$). Portanto, o custo do segundo passo é $O(Q * (N * (N + N))) = O(QN^2)$.

Portanto, o pior caso do algoritmo pode ser descrito pela seguinte fórmula:
 $\Theta(M) + O(QN^2) = O(QN^2)$.

4. Execução do programa

Para executar o programa, basta compilar e executar utilizando os seguintes comandos:

```
g++ -Wall -g -c src/Neighbor.cpp -o obj/Neighbor.o -I ./include/
g++ -Wall -g -c src/Graph.cpp -o obj/Graph.o -I ./include/
g++ -Wall -g -c src/main.cpp -o obj/main.o -I ./include/
g++ -Wall -g -o ./bin/tp02.out ./obj/Neighbor.o ./obj/Graph.o ./obj/main.o
```

```
./bin/tp02.out < ./data/e1.in
```

Onde `./data/e1.in` pode ser substituído pelo caminho do arquivo de entrada desejado.

5. Referências bibliográficas

Dijkstra:

<https://favtutor.com/blogs/dijkstras-algorithm-cpp>

Widest path problem:

https://en.wikipedia.org/wiki/Widest_path_problem

Bibliotecas C++:

<https://en.cppreference.com/w/cpp/header>

Corretude de Dijkstra:

<https://web.engr.oregonstate.edu/~glencora/wiki/uploads/dijkstra-proof.pdf>