Algoritmo e Estrutura de Dados III

Documentação do Trabalho Prático 1

Ramon Gonçalves Gonze

21 de maio de 2017

1. INTRODUÇÃO

Grafos podem ser uma forma bastante prática de representar problemas que envolvam características de objetos com ligações entre si. Dentro deste contexto, um dos problemas nos quais a solução pode ser bem útil, é o de fluxo máximo. O problema consiste em obter, sob um grafo, um fluxo máximo de algo flui através das arestas que ligam os vértices deste grafo. Tem-se um (ou vários) vértice(s) de origem - de onde sai o fluxo - e um (ou vários) vértice(s) de destino - onde o fluxo deve chegar.

Os algoritmos conhecidos atualmente reduzem problemas que possuem mais de uma origem (ou destino) para somente uma origem e um destino. Este trabalho prático consiste na implementação do algoritmo de Edmonds-Karp, que encontra o fluxo máximo de um grafo, que neste caso é a quantidade máxima de ciclistas que pode sair das franquias por hora. As interseções do problema serão representadas pelos vértices do grafo, as ciclovias pelas arestas, e a capacidade de ciclistas em cada ciclovia pelo peso de cada aresta. O grafo será não-direcionado, pois as ciclovias são de mão única.

2. SOLUÇÃO DO PROBLEMA

O algoritmo de Edmonds-Karp busca encontrar um fluxo máximo em um grafo, de um vértice de origem *s* para um vértice de destino *t*. Ambos são criados e acrescentados ao grafo original da entrada. A figura abaixo demonstra como esses vértices são utilizados:

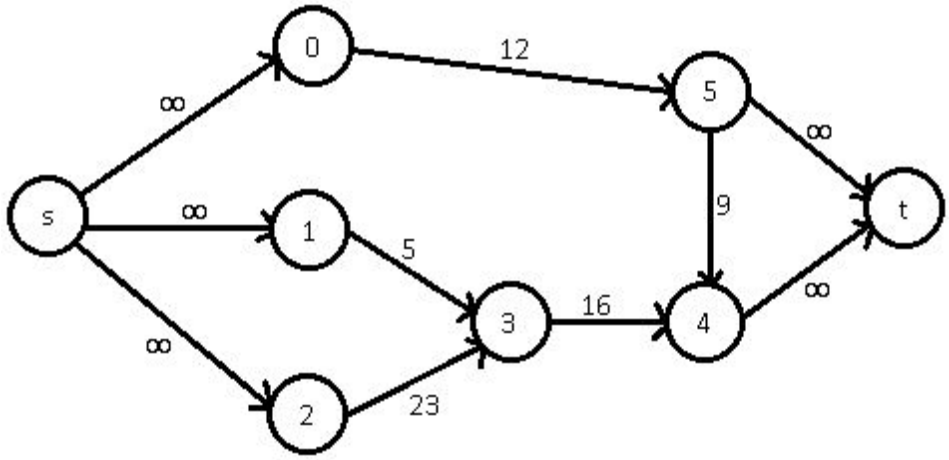

*Figura 1: Transformação de vários vértices de origem (V_0 , V_1 , V_2) em uma única origem **s**, e dois vértices de destino (V_4 , V_5) em um único destino **t**.*

Inserindo arestas de peso infinito entre os vértices *s* e *t* e os demais vértices, o fluxo máximo não é influenciado, e agora possuímos somente uma origem e destino. O pseudocódigo abaixo explica o funcionamento do algoritmo:

Algoritmo 1

```
function maxFlow(G)
1       max_flow ← 0
2       while there is a valid path between s and t do
3               predecessor[] ← BFS(G)
4               smallest_edge ← findSmallestEdge(G, predecessor)
5               for each (u, v) ∈ predecessor[] do
6                       weight((u, v)) ← weight((u, v)) - smallest_edge
7                       weight((v, u)) ← smallest_edge
8               max_flow ← max_flow + smallest_edge
9       return max_flow
```

O algoritmo utiliza a Busca em Largura (*BFS*) a partir do vértice *s* para gerar uma lista de antecessores dos vértices. Dado o vetor *predecessor[]* encontrado na linha 3, haverá somente um caminho entre *s* e *t*. Após encontrar a aresta de menor peso desse caminho, o algoritmo decrementa o valor de todas as arestas *(u,v)* do caminho entre *s* e *t*, e adiciona arestas *(v, u)* com o peso da menor aresta. O valor do fluxo máximo é incrementado com o valor encontrado na linha 4, ao final de cada loop. Quando não houver nenhum caminho válido entre *s* e *t*, ou seja, quando em todos os caminhos

houver uma aresta de peso 0 (capacidade de fluxo 0), a variável *max_flow* terá o valor do fluxo máximo.

A estrutura do grafo foi implementada com uma matriz quadrática de adjacências, de ordem $V + 2$ vértices, sendo V_V e V_{V+1} os vértices adicionais (a origem e o destino respectivamente).

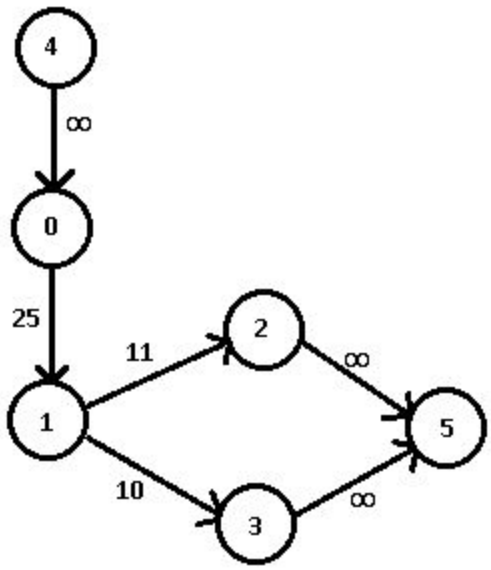

Figura 2: Exemplo de grafo

		0	1	2	3	4	5
	0	0	25	0	0	0	0
	1	0	0	11	10	0	0
	2	0	0	0	0	0	∞
	3	0	0	0	0	0	∞
Origem →	4	∞	0	0	0	0	0
Destino →	5	0	0	0	0	0	0

Figura 3: Matriz de adjacências do grafo representado na Figura 2

Foram implementadas cinco funções principais para a execução do algoritmo acima:

- ➔ createGraph(TGraph *G);
- ➔ readGraph(TGraph *G);
- ➔ destroyGraph(TGraph *G);
- ➔ findSmallestEdge(TGraph *G);
- ➔ BFS(TGraph *G).

Para a função *BFS,* foi utilizada uma **fila** com as funções básicas de *criar fila vazia*, *enfileirar*, *desenfileirar*, *testar se a fila está vazia* e *destruir fila*.

3. ANÁLISE DE COMPLEXIDADE

Sendo V a quantidade de vértices do grafo lido + 2 vértices adicionais, e E a quantidade de arestas do grafo - incluindo as arestas que não existem no grafo, que são representadas com peso 0 na matriz de adjacências - seguem as análises de complexidade das funções:

- ➔ **createGraph(TGraph *G)**: A função faz a alocação de memória para um *TGraph*. Suas complexidades temporal e espacial são *O(1)*.
- ➔ **readGraph(TGraph *G)**: A leitura é feita na seguinte ordem: quantidade de vértices, quantidade de arestas, quantidade de franquias e quantidade de clientes (V, E, F e C respectivamente), sendo $2 \leq V \leq 1000$, $1 \leq E \leq 10.000$ e $1 \leq F, C \leq V$. A função possui loops **for** que vão de *1* até *V*, *1* até *E*, *1* até *F* e *1* até *C*. Não há loops internos a outros. Portanto, a complexidade temporal da função é *max(O(V), O(E))*. É alocada memória para uma matriz $V \times V$, e para os vetores *predecessor[]* e *color[]*, que possuem tamanho *V*. A complexidade espacial da função é $O(V^2)$.
- ➔ **destroyGraph(TGraph *G)**: A função faz chamadas da função **free()** e possui somente um loop, que vai de *1* até *V*. Logo, sua complexidade temporal é *O(V)*, e a espacial é *O(1)*.
- ➔ **findSmallestEdge(TGraph *G)**: Esta função possui dois loops (não internos) que percorrem um caminho do vértice de origem até o vértice de destino, indicado pelo vetor *predecessor[]*. O peso de cada aresta é verificado em *O(1)*, na matriz de adjacências. No pior caso, este caminho irá conter todas as arestas

do grafo, logo sua complexidade temporal é *O(E)* e a espacial é *O(1)*.

- ➔ **BFS(TGraph *G)**: A ideia do algoritmo de Busca em Largura é, a partir de um vértice, enfileirar os vértices adjacentes a ele em uma fila, e, após desenfileirar cada vértice adjacente, enfileirar os vértices adjacentes a ele. A construção do vetor *antecessor[]* do Algoritmo 1 é feita neste momento. Nestas operações, cada vértice é enfileirado somente uma vez (não há self-loops no grafo), e também desenfileirado somente uma vez. As operações de **enfileira** e **desenfileira** são *O(1)*, logo esta parte faz *O(V)* operações. Para cada vértice desenfileirado, sua lista de vértices adjacentes é percorrida. Como o grafo foi implementado com uma matriz de adjacências, a lista de adjacência de cada vértice possui tamanho *V*, sendo esta parte então, também *O(V)*. A complexidade final da função, portanto, é $O(V^2)$. Para a análise espacial, deve-se considerar a fila criada. Essa pode possuir um tamanho máximo de *V*, que ocorre quando todos os vértices do grafo são enfileirados, ou seja, o vértice de origem possui arestas para todos os outros vértices. Isso torna a complexidade espacial da função *O(V)*.

O programa principal é a implementação do Algoritmo 1, de Edmonds-Karp. A complexidade temporal deste algoritmo é conhecida, e é igual a $O(VE^2)$ *(CORMEN, 2009)*, na sua melhor implementação. O loop principal do programa é da ordem de *O(VE)*, que são os caminhos válidos entre o vértice de origem e o vértice de destino. Para cada execução, as funções *BFS* e *findSmallestEdge* são chamadas uma única vez, possuindo estas, complexidades $O(V^2)$ e *O(E)*, como já dito anteriormente. A complexidade final temporal do algoritmo é $O(V^3E^2)$. Já a complexidade temporal, se baseia elementos *matriz de adjacências*; vetor *predecessor*; vetor *color*, e na fila construída na execução de *BFS*. A partir da análise espacial já feita anteriormente de cada um desses elementos, a complexidade espacial final do programa principal é $O(V^2) + O(V) + O(V) + O(V) = O(V^2)$.

4. AVALIAÇÃO EXPERIMENTAL

Os testes foram feitos em um ambiente Linux utilizando a distribuição Ubuntu 16.04 LTS, em um computador com processador core i3 de 2.1 GHz e 6GB de RAM.

Os testes exaustivos foram realizados com o intuito de verificar qual o comportamento do algoritmo a respeito da variação do número de vértices e arestas.

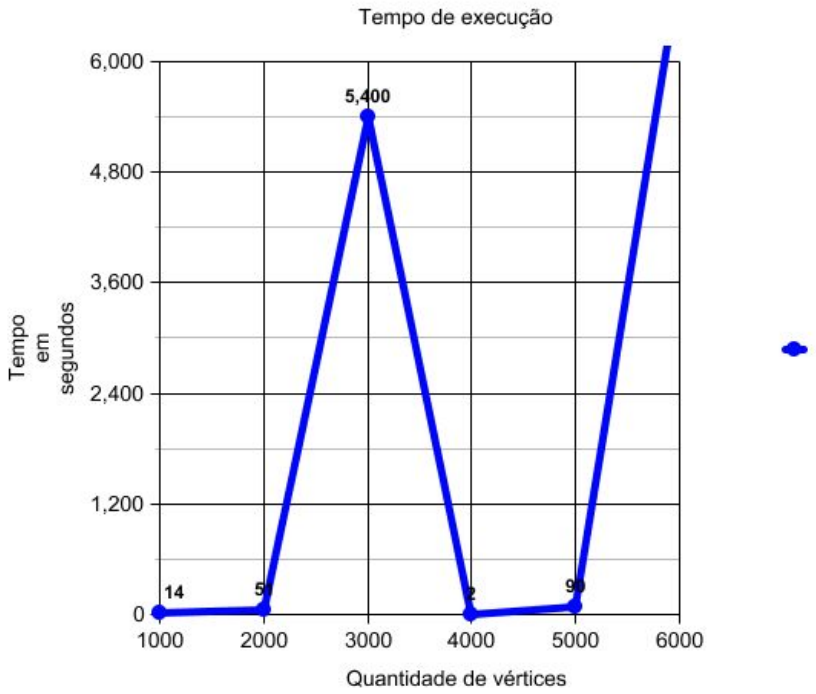

Gráfico 1: Tempo de execução do algoritmo em relação a quantidade de vértices

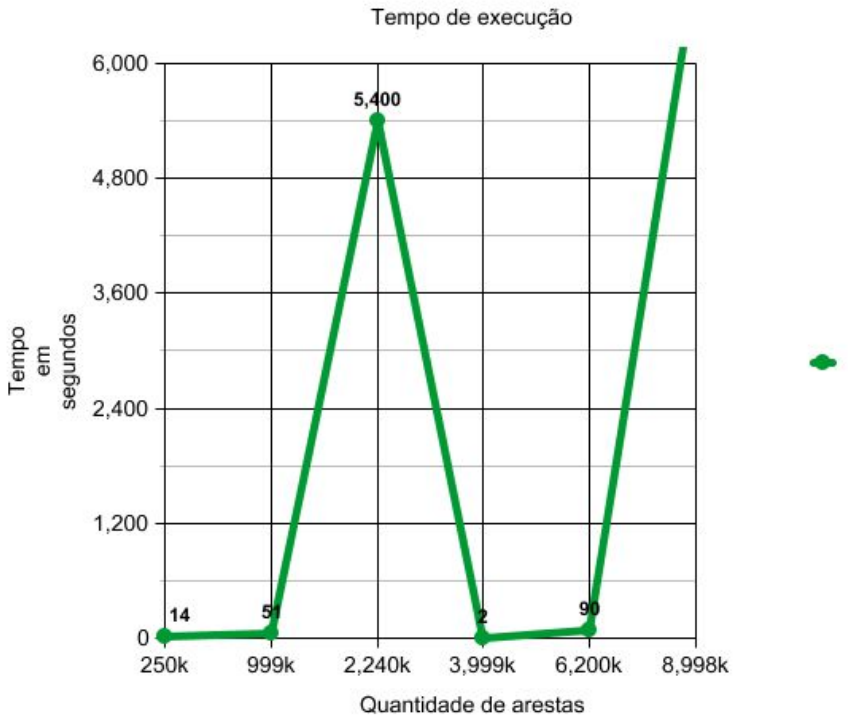

Gráfico 2: Tempo de execução do algoritmo em relação a quantidade de arestas

Apesar de o algoritmo tender a crescer cubicamente por sua complexidade ser $O(V^3E^2)$, os testes realizados foram inconclusivos, pelo fato da execução do algoritmo depender da quantidade de caminhos válidos entre o vértice de origem e o vértice de destino. E portanto, conclui-se que a quantidade de caminhos válidos não cresce na mesma proporção que a quantidade de vértices e arestas.

5. REFERÊNCIAS

CORMEN, T. H. Introduction to Algorithms. Third Edition. London: Massachusetts Institute of Technology, 2009. 1292p.