

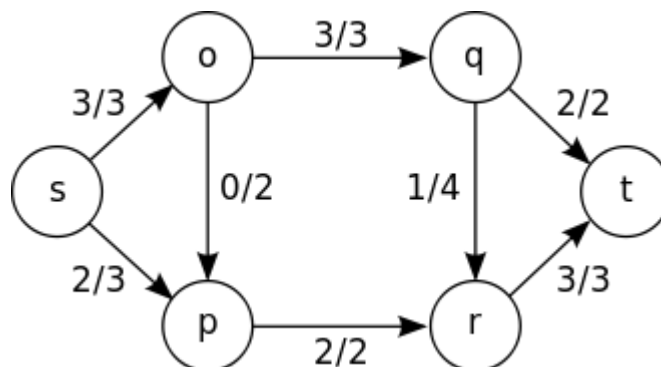
Trabalho Prático 1: Entregando Lanches

Algoritmos e Estruturas de Dados III - 2017/1

Luciano Otoni Milen [2012079754]

1. Introdução

Neste trabalho prático devemos implementar um algoritmo que determine o fluxo máximo em um grafo. O problema consiste em calcular quantos ciclistas podem passar num caminho ponderado, partindo de franquias e alcançando os clientes. Para tal, a modelagem do grafo deve ser feita de forma a armazenar as franquias e clientes em vértices diferentes, onde o ciclista deve conseguir fazer sua entrega passando pelo caminho de fluxo ideal. As arestas são as ciclofaixas e os vértices podem ser franquias ou clientes, ou nenhum dos dois. Lembrando que o grafo é ponderado e direcionado.



Fonte: Wikipedia. A imagem representa a estrutura do grafo: *s* de *source*, que em português significa origem; *t* é o destino do ciclista.

O grafo pode ser representado como uma matriz, onde as posições $[i][j]$ representam a conexão entre os vértices, definindo se elas existem e quais são os pesos.

2. Solução do Problema

Este trabalho envolve um problema já conhecido na computação: o fluxo máximo. Após a leitura sobre o tema é fácil encontrar diversas soluções para resolvê-lo, porém a mais apropriada para o trabalho é o algoritmo de Ford-Fulkerson. Este algoritmo se encaixa na categoria de algoritmos gulosos, que devem tomar decisões ótimas a cada passo. Verifica-se a possibilidade de percorrer um caminho e calcula-se o fluxo máximo

permitido para cada caminho computado. Em seguida, escolhe aquele que permite que o maior número de ciclistas consigam trafegá-lo.

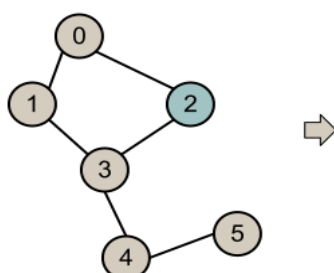
1. Após ler quantas interseções, a quantidade de ciclofaixas, o número de interseções onde existe uma franquía e o número de interseções onde estão os clientes, lê-se as conexões entre os vértices do grafo e, em seguida, a posição dos vértices de franquías e clientes;
2. Visto que é possível ter múltiplas origens e múltiplos destinos (ou seja, vários clientes e franquías) não é possível determinar uma origem ou destino únicos. Desta forma, a estrutura do grafo deve ser alterada da seguinte forma:
 - a. Cria-se 2 vértices a mais do que o especificado na entrada do problema;
 - b. Conecta cada cliente a um destes vértices, e as franquías ao outro vértice criado;
 - c. Considera-se o peso destas conexões como o máximo possível, pois assim o problema é transformado de várias origens e vértices para uma origem e um vértice somente;
3. O programa então entra no algoritmo de Ford-Fulkerson, que será melhor explicado a seguir;
4. Depois de calcular o fluxo máximo do grafo o valor é retornado na tela.

Decisões de implementação

A representação de um grafo através de uma matriz torna o problema consideravelmente mais simples de se compreender e de resolver. Entretanto, para resolvê-lo de forma eficiente, sem dúvida é melhor alocar memória para as matrizes dinamicamente, e assim foi feito. A função `int **alocaGrafoMatriz(int tamanho)` retorna um ponteiro para uma matriz alocada de acordo com o *tamanho* desejado.

O algoritmo de Ford-Fulkerson envolve a implementação da busca em profundidade (BFS), que percorre cada segmento do grafo até chegar ao final, antes de retroceder e percorrer os outros caminhos possíveis. Um vetor de inteiros armazena as posições dos vértices já visitados, para saber se aquele caminho já foi percorrido ou não. Um outro vetor, de vértices superiores, serve para identificar os vértices que precedem aquele que está sendo visitado no momento. O BFS também exige uma fila para ser implementado. Para tal foi criada uma estrutura de fila representada por uma lista encadeada, onde tem-se o elemento da frente, o de trás e métodos para manipulá-la, como alocar memória para a fila, verificar se está vazia e inserir e retirar elementos.

Breadth First Search



1. $q = \{\}$
2. $q = \{2\}$
3. $q = \{0, 3\}$
4. $q = \{1\}$
5. $q = \{4\}$
6. $q = \{5\}$

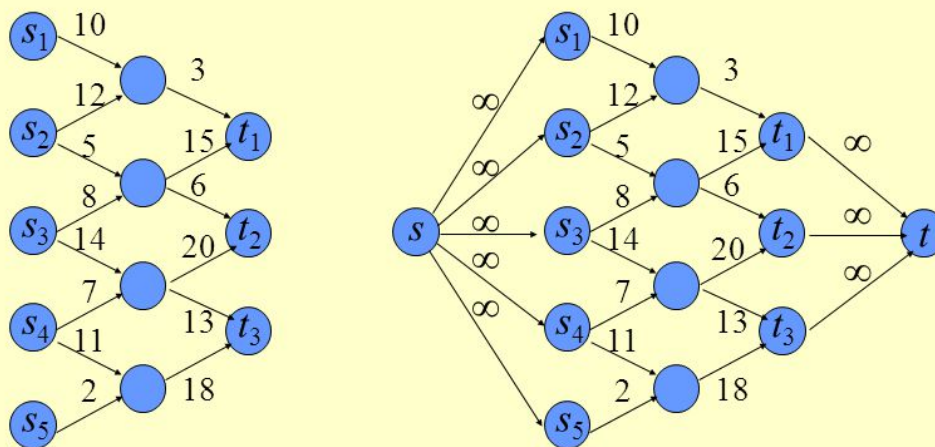
Fonte: Stoiman. A busca em profundidade utilizando uma fila.

Using a queue

Como mencionado anteriormente, o problema deve ser reestruturado quando temos várias fontes e destinos. A imagem a seguir representa perfeitamente o que deve ser feito neste caso:

■ Networks with multiple sources and sinks

- Introduce *supersource* s and *supersink* t



Yangjun Chen

9

Fonte: Yangjun Chen. Na esquerda a estrutura original do problema está representada: cada s_i é a fonte, ou franquia e cada t_i o destino, ou cliente. A solução mais simples é criar dois outros vértices que se conectem a todos os clientes e franquias com peso infinito.

Inicialização do grafo

Para criar um grafo, basta chamar a função `alocaGrafoMatriz(tamanho)`. A função primeiramente aloca a memória da matriz correspondente o tamanho especificado e em seguida a memória para cada linha da matriz. Retorna um ponteiro para a matriz desejada.

BFS

A busca em profundidade, conforme descrito anteriormente, percorre os vértices do grafo para analisar o peso correspondente a cada possível fluxo da franquia ao cliente. Os vértices são armazenados numa fila e retirados à medida que são percorridos. Um vetor identifica se os vértices atuais já foram percorridos. A implementação normalmente envolve a coloração do grafo, mas neste caso não foi necessário mais do que um valor binário para saber se a busca passou pelo vértice ou não.

A complexidade de tempo da busca em profundidade é baixa, definida teoricamente como $O(|V| + |E|)$. A seguir tem-se o algoritmo da BFS descrito em pseudocódigo:

```
Algoritmo: bfs (grafo[][], tamanho, origem, destino,  
vertices superiores[])
```

```
cria Fila  
insere(fila, origem)  
enquanto(!vazia(fila))  
|   item h = primeiro_da_fila  
|   removeItem(fila)  
|   for(i:0..qtd_vertices)  
|   |   se !vertice_visitado[i] AND (existe_aresta)  
|   |   |   insere(fila, i)  
|   |   |   vertice_visitado[i] = true  
return vertice_visitado[origem]
```

Ford-Fulkerson

Dado um grafo direcionado e ponderado, o algoritmo de Ford-Fulkerson calcula o fluxo máximo possível na estrutura apresentada. O algoritmo faz uma busca em profundidade em cada caminho possível no grafo e determina qual o percurso de maior fluxo possível, através de uma simples comparação no cálculo do fluxo total entre os caminhos. A ideia de calcular o fluxo é simples: basta considerar quantos vértices chegam no vértice v e quantos saem dele. Portanto, o cálculo base é saída(v) – entrada(v). O grafo auxiliar, chamado comumente de grafo residual, é inicializado com as mesmas conexões do grafo original. À medida que o grafo original é percorrido o valor do fluxo do caminho é calculado e inserido no grafo auxiliar nas posições correspondentes aos vértices que fazem parte do caminho ideal.

```
calculaFluxo (grafo[][], qtd_vertices, origem, destino)
```

```
cria grafoAux[qtd_vertices][qtd_vertices]  
copia(grafoAux <- grafo)  
inicializa vertices_superiores[qtd_vertices]  
fluxo_max = 0  
enquanto(bfs(grafoAux))  
|   fluxo_caminho = infinito  
|   enquanto(v = destino; v != origem; v = vertices_superiores[v])  
|   |   i = vertices_superiores[v]  
|   |   fluxo_caminho = minimo(fluxo_caminho, grafoAux[i][v])  
|   enquanto(v = destino; v != origem; v = vertices_superiores[v])  
|   |   i = vertices_superiores[v]  
|   |   grafoAux[i][v] -= fluxo_caminho  
|   |   grafoAux[v][i] += fluxo_caminho  
|   fluxo_max += fluxo_caminho  
return fluxo_max
```

A complexidade do algoritmo de Ford-Fulkerson envolve duas variáveis: m , que corresponde ao número de arestas presentes no grafo original e f , que é o fluxo máximo calculado pelo algoritmo. No pior caso, temos que a complexidade total deste algoritmo pode ser simplificada para Erro.

Existem, entretanto, outros algoritmos para o cálculo do fluxo máximo que são independentes do próprio fluxo. Um deles é o Edmonds-Karp, que independente do valor do fluxo assume complexidade média de $O(V * E^2)$. Todavia, sua implementação é consideravelmente mais complexa do que a utilizada neste trabalho.

Algoritmos da fila

Para utilizar a busca em profundidade é necessário implementar uma fila. A estrutura de dados utilizada foi uma lista encadeada, onde cada elemento possui um ponteiro para o próximo da fila, além de dois ponteiros indicando onde ela começa e termina. Devido à simplicidade dos itens da fila há somente um inteiro que identifica o valor do vértice e um apontador para o elemento seguinte da fila. Para trabalhar em cima da estrutura foram criados 4 métodos:

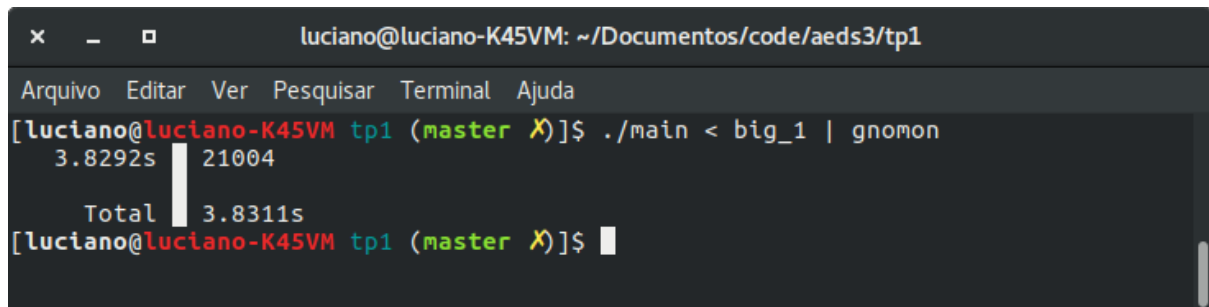
- `removeFila(fila)`: como já é sabido deste tipo de estrutura, o primeiro que sai é o primeiro que entra, logo este método simplesmente identifica a frente da fila como o elemento seguinte àquele que foi removido.
- `insereFila (fila, item)`: já que este elemento foi o último a chegar, entra no final da fila. O apontador de trás muda seu ponteiro para o novo elemento e o que era antes o último identificará o elemento que acabou de chegar como o próximo da fila.
- `criaFila()`: retorna um ponteiro para uma fila devidamente alocada e inicializada.
- `filaVazia (fila)`: avalia se o ponteiro da frente da fila é o mesmo de trás, ou seja, a fila está vazia.

3. Análise de Complexidade

Tempo

- BFS: o algoritmo de busca em profundidade no pior caso deve passar por cada vértice e aresta do grafo. Para tal, temos $O(V)$ para os vértices e $O(A)$ para as arestas. Entretanto, o número de arestas pode chegar a $O(V^2)$ dependendo da estrutura do grafo. As inserções e remoções da fila presente no algoritmo são executadas no máximo V vezes, desta forma podemos concluir que a complexidade total do algoritmo de BFS é $O(|V| + |E|)$.
- `calculaFluxo`: a função principal do código faz chamadas à BFS na medida que percorre o grafo. Os *loops* internos são executados para cada vértice existente no grafo e são feitos duas vezes. Temos então $O(2V)$, que podemos simplificar para $O(V)$. Nesta função há também a atribuição da matriz auxiliar com os valores do grafo original, procedimento de dois laços que executam de 0 até o número de arestas. Temos, portanto, $O(V^2)$. A complexidade total do algoritmo `calculaFluxo` pode ser computada como $O(V^2)$.
- Total: considerando os dois algoritmos principais do programa, já que os outros somente atribuem valores às variáveis e fazem operações numa fila (cada

operação $O(1)$, se BFS é $O(V + E)$ e calculaFluxo é $O(V^2)$ a complexidade total do código é $O(V^2)$.

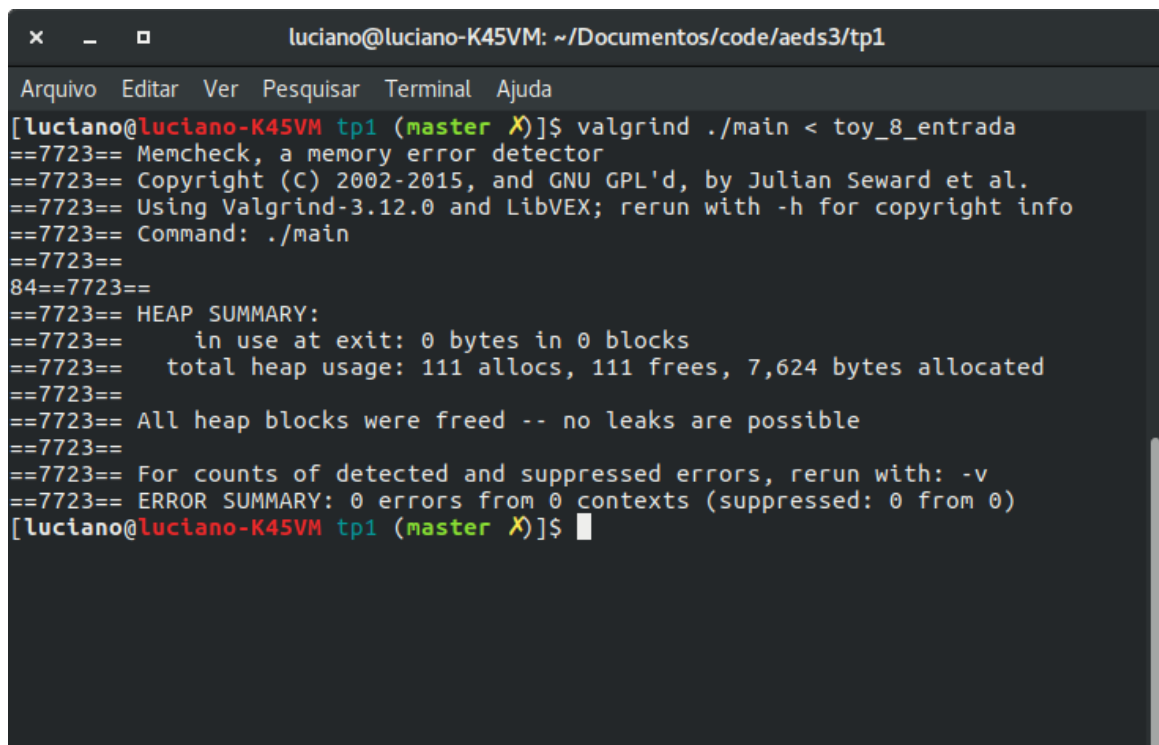


```
luciano@luciano-K45VM: ~/Documentos/code/aeds3/tp1
Arquivo Editar Ver Pesquisar Terminal Ajuda
[luciano@luciano-K45VM tp1 (master X)]$ ./main < big_1 | gnomon
3.8292s 21004
Total 3.8311s
[luciano@luciano-K45VM tp1 (master X)]$
```

Exemplo de gasto do tempo de execução do trabalho para o teste *big_1* utilizando o software Gnomon [3].

Espaço

- BFS: a estrutura utilizada para o processamento do grafo é uma matriz de adjacências e já que existem $V * V$ vértices, a complexidade para a matriz é $O(V^2)$. É necessária também a estrutura de pilha, que armazena até V itens. Já que a matriz ocupa mais espaço que a fila, temos $O(V^2)$ como complexidade total.
- calculaFluxo: novamente a estrutura da matriz ocupa $O(V^2)$ de espaço. Como tem-se a necessidade de criar um grafo auxiliar para processar o grafo residual, tem-se $O(2 * V^2)$, que pode ser simplificada para $O(V^2)$.
- Total: a complexidade de espaço total do programa fica fácil de calcular já que sabemos que cada algoritmo descrito acima ocupa $O(V^2)$ de espaço: $O(2 * V^2)$ pode ser simplificada para $O(V^2)$.



```
luciano@luciano-K45VM: ~/Documentos/code/aeds3/tp1
Arquivo Editar Ver Pesquisar Terminal Ajuda
[luciano@luciano-K45VM tp1 (master X)]$ valgrind ./main < toy_8_entrada
==7723== Memcheck, a memory error detector
==7723== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7723== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==7723== Command: ./main
==7723==
84==7723==
==7723== HEAP SUMMARY:
==7723==    in use at exit: 0 bytes in 0 blocks
==7723==   total heap usage: 111 allocs, 111 frees, 7,624 bytes allocated
==7723==
==7723== All heap blocks were freed -- no leaks are possible
==7723==
==7723== For counts of detected and suppressed errors, rerun with: -v
==7723== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[luciano@luciano-K45VM tp1 (master X)]$
```

Teste de uso de memória com o Valgrind para o teste *toy 9*.

4. Análise Experimental

Para fazer a análise de tempo gasto pelo programa foi utilizado o Gnomon, software desenvolvido sob licença do MIT pelo usuário *zetlen* do GitHub. Basta incluir “| gnomon” após o executável no terminal para contar o tempo gasto. A análise de memória foi feita com o Valgrind, que mostra a quantidade de *bytes* utilizado pelo programa.

O programa foi elaborado e testado em uma máquina Linux Ubuntu GNOME 17 Intel® Core™ i7-3610QM CPU @ 2.30GHz × 8 com 8gb de memória RAM. A tabela a seguir associa alguns testes feitos com o gasto de memória e de tempo.

Teste	Interseções	Ciclofaixas	Tempo (s)	Memória (bytes)
toy1	7	8	0.0088	6 504
toy5	11	12	0.0091	7 752
toy7	12	15	0.0095	8 272
big_1	1000	249 750	3.9201	35 566 848
big_4	4000	3 999 000	0.9081	128 200 368

- Os testes t1, t5 e t10 correspondem aos testes *toy* disponibilizados com o trabalho.
- big_1 e big_3 foi feito com os testes criados pelo aluno Artur Henrique Marzano e disponibilizado no Moodle da disciplina.

Os resultados correspondem à média de 30 execuções de cada teste. Este número foi determinado a partir da observação do comportamento dos testes, que não sofriam alterações tão significativas.

É interessante observar que o gasto de tempo e memória variam de acordo com 4 fatores: o número de interseções, de ciclofaixas, franquias e clientes. Como podemos observar nos testes *big* o fato da entrada possuir 4 vezes mais interseções no *big_4* e mais de 10 vezes mais ciclofaixas em relação ao *big_1* não determina que o segundo executará mais rápido que o primeiro, ainda que o gasto de memória do *big_4* seja muitas vezes superior.

A execução do código juntamente com o Valgrind se torna impraticável para entradas muito grandes. A análise de memória aumenta em muitas vezes o tempo necessário para o algoritmo rodar.

4. Conclusão

Este trabalho foi relativamente simples de implementar, pois o problema do fluxo máximo já foi amplamente estudado na computação. O algoritmo de Ford-Fulkerson praticamente elimina o problema, exceto quando temos que considerar múltiplas origens e destinos. Mesmo assim, a solução para esta dificuldade é simples, bastando adicionar um vértice de arestas de peso infinito para todas as origens e um outro vértice

do mesmo tipo para todos os destinos. A busca em profundidade também se mostra simples de implementar, já que utiliza estruturas triviais como a fila.

Foi possível entender como estes algoritmos funcionam e quão eficiente eles são. Se considerarmos o fluxo máximo de um bairro inteiro, por exemplo, certamente o tempo necessário para computar o resultado em um computador simples como o que foi usado para os testes seria impraticável.

5. Referências

[Slides do professor Nívio sobre filas e grafos](#)

https://en.wikipedia.org/wiki/Breadth-first_search

<http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/>

<https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/the-breadth-first-search-algorithm>

<http://theory.stanford.edu/~megiddo/pdf/optflows.pdf>

http://www.ifp.illinois.edu/~angelia/ge330fall09_maxflow120.pdf

<https://github.com/paypal/gnomon>