TP1 – Entregando lanches

Nome: Henrique Eustáquio Lopes Ferreira – **Matrícula:** 2015068990

1. Introdução

Rada abriu uma firma de entregas de lanches operadas por ciclistas, considerando que sua cidade possui boas ciclofaixas de mão única. Então contratou seu amigo Gilmar para lhe ajudar na parte de logística, coordenando a entrega de lanches pela cidade. Tendo em vista que os ciclistas saem de franquias e chegam a clientes, passando por quaisquer pontos na cidade onde se encontram as ciclovias, o problema de Gilmar se resume a, dado que cada ciclovia possui um limite máximo de ciclistas por hora, calcular quantos ciclistas Rada pode enviar por hora para maximizar o número de entregas, respeitando a segurança e as direções das ciclofaixas.

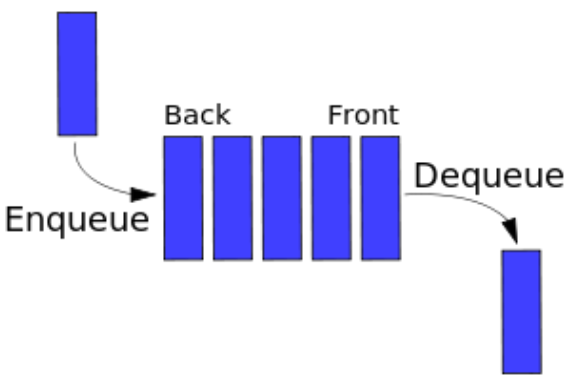
É natural a interpretação do problema como um de grafos, logo a cidade foi representada como um grafo por meio de uma matriz de adjacências. Assim, cada ciclofaixa corresponde a uma posição na matriz onde se encontra seu peso (máximo de ciclistas por hora) e as interseções, por sua vez, correspondem aos índices dessa matriz. Por exemplo, se há uma ciclofaixa interligando as interseções 0 e 1 cujo número máximo de ciclistas é 5, temos, na linha 0, coluna 1 da matriz, o peso 5.

A partir da representação de grafo descrita acima, abordou-se a solução com a implementação Edmonds-Karp do algoritmo de Ford-Fulkerson para fluxo máximo, na qual usa-se o caminhamento BFS (Breadth-First Search), em largura, para identificar os caminhos aumentadores, do vértice inicial ao final, e opera-se nos pesos das arestas para criar os fluxos. Basta, daí, identificar o fluxo máximo entre cada caminho, de uma franquia para um cliente.

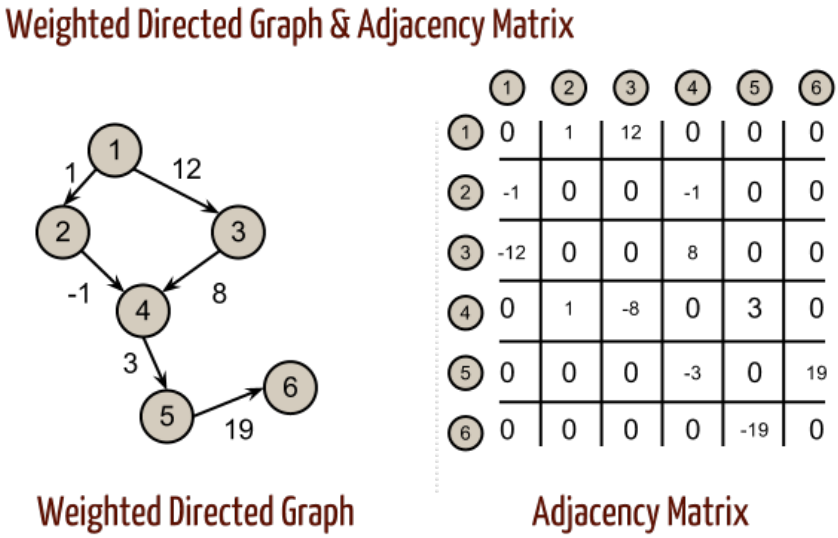
2. Solução do problema

Para a implementação do trabalho foram criados dois principais Tipos Abstratos de Dados:

- Fila auxiliar de inteiros, para o caminhamento BFS. Representação do funcionamento FIFO (First In, First Out):



- Grafo representado por matriz de adjacência, em que cada entrada possui dados sobre peso. Ilustração de exemplo:



	1	2	3	4	5	6
1	0	1	12	0	0	0
2	-1	0	0	-1	0	0
3	-12	0	0	8	0	0
4	0	1	-8	0	3	0
5	0	0	0	-3	0	19
6	0	0	0	0	-19	0

Sobre o processamento, temos que, basicamente, a função *main* do arquivo principal cira a estrutura (o grafo) cidade, a partir dos números informados na entrada, inicializando-se a partir do número de vértices e iterando sobre a quantidade de arestas, adicionando seus pesos à implementação final. Após isso, o programa roda a função Ford-Fulkerson (que por sua vez utiliza a função BFS) para cada combinação franquia-cliente, também informados na entrada, na matriz de adjacências da cidade. Em outras palavras, de cada franquia procura-se chegar a cada cliente, guardando os valores máximos de ciclistas (fluxo máximo retornado) e atualizando o vetor que contém tais quantidades máximas. Isso porque se já chegou-se a um cliente com um quantidade máxima, então procura-se chegar aos próximos clientes, procurando-se sair de não mais do que uma franquia para chegar a um cliente por hora. Logo, a resposta buscada é a soma de todos os valores máximos, armazenados num vetor (cada um de um caminho que sai de uma franquia e chega a um cliente). O princípio de encapsulamento, então, foi priorizado, de forma que somente vetores necessários ao processamento final da resposta se encontram no arquivo principal tp1.c. Além de tais vetores, é importante citar a estrutura cliente, que nada mais guarda além do dado sobre tal cliente já ter sido visitado e, claro, o número correspondente à interseção onde se encontra tal cliente.

O grafo, implementado no arquivo correspondente grafo.c, é inicializado com pesos zerados. Em sua implementação também há, claro, as funções de caminhamento BFS e de Fluxo-Máximo Ford-Fulkerson, explicadas com mais profundidade no próximo tópico. Por fim, também encontram-se uma função de impressão e outra para o tratamento da memória alocada, além do método responsável pela adição de arestas.

Finalmente, tem-se a implementação da fila de inteiros, usada exclusivamente no algoritmo de BFS e que consta completa no arquivo fila_bfs.c. Nela há funções de inicialização, deleção e de enfileiramento (inserção de elemento atrás) e desenfileiramento (retirada de elemento da frente), basicamente.

3. Análise de complexidade

Para esta seção partiremos do princípio de analisar somente aquilo crucial para a solução do problema, em termos de complexidade e espaço. Assim, analisaremos o grafo (seção 3.1), sua estrutura e principais funções, a fila (seção 3.2) para o caminhamento em largura BFS utilizada pela implementação Edmonds-Karp e, por último, a função *main* (seção 3.3) em si no arquivo principal. Preocuparemos, também, em dispor sobre os arquivos correspondentes a cada implementação, para fins de máxima elucidação sobre o comportamento do programa como um todo.

3.1. Grafo

No que diz respeito ao grafo, temos que a matriz alocada dinamicamente que representa a cidade do problema ocupa um espaço de complexidade $O(|V|^2)$, isso porque para o algoritmo de Ford-Fulkerson precisamos supor arestas de sentido contrário para calcular o fluxo. Todo grafo, então, possui uma subestrutura que é encarada como uma matriz de arestas, conforme consta no código, de tamanho $|V|^2$, sendo $|V|$ o número de vértices informado pela entrada como a quantidade de interseções da cidade. Enfim, disso também conclui-se que a função cria_grafo(...) executa em complexidade de tempo $O(|V|^2)$, ao inicializar o grafo com pesos nulos. Por sua vez, a função clear_grafo(…), cujo propósito é desalocar memória da estrutura matriz, executa em $O(|V|)$, liberando linha a linha. A função adiciona_aresta(...), entretanto, apenas muda os valores do atributo peso da entrada correspondente da matriz, executando em tempo $O(1)$.

O algoritmo BFS possui complexidade expressa em termos de $O(|A| + |V|)$, sendo A o número de arestas e V o número de vértices. Isso porque cada vértice e cada aresta serão visitados no pior caso da execução, quando enfileira-se cada vértice a partir da origem e, para cada um, visita-se cada adjacência. Assim, como $O(|A|)$ se encontra entre $O(1)$ e $O(|V|^2)$, podemos ter um pior caso de $O(|V|^2 + |V|) = O(|V|^2)$. Enfim, por utilizar um vetor auxiliar booleano de vértices visitados, tem-se também uma complexidade de espaço de $O(|V|)$, além da fila auxiliar, que pode chegar a ocupar $O(|V|^2)$.

No caso do Ford-Fulkerson, como o algoritmo busca um pseudocaminho aumentador (que sai do vértice inicial em direção ao final e cuja capacidade pode ser aumentada, basicamente) a cada iteração, temos uma complexidade de tempo de $O(|V|*|A|^2)$. Isso já que usamos BFS, que sempre retorna o caminho com o número mínimo de arestas. Assim, ao todo a geração do fluxo máximo acontece em $O(|A|*|V|^3)$. Sobre o espaço,

nesse caso tem-se um vetor auxiliar que identifica a fonte de onde o caminho aumentador atual sai (os “pais” dos vértices), atualizado na função BFS, e um grafo residual, no qual as capacidades podem ser livremente trabalhadas. Respectivamente, temos O(|V|) e O(|V|²).

3.2. Fila

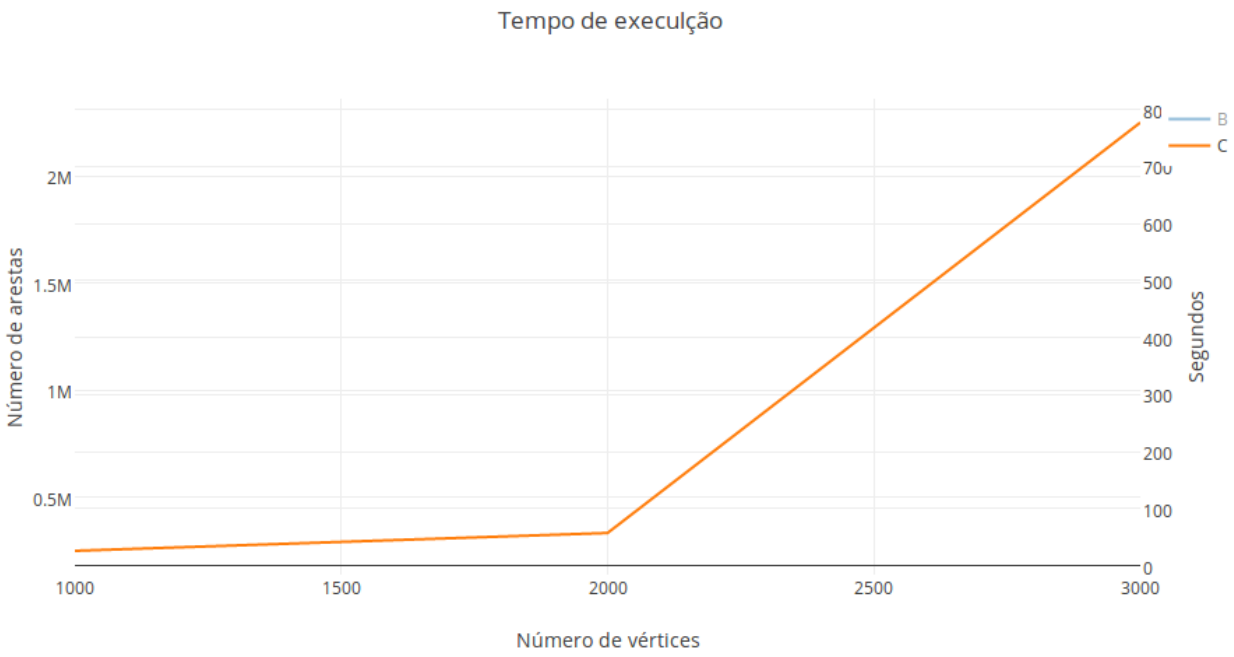
Como tal implementação de fila é por lista encadeada, toda alteração pode ser feita em O(1), exceto quando devemos percorrer toda a lista. Dessa forma, só presenciamos tal necessidade ao desalocarmos memória, de maneira que a função Destroi(…) possui complexidade máxima O(|V|), conforme elucidado quando sobre o algoritmo BFS. Importante notar que não há a necessidade de pesquisa, daí conclui-se que o caso da função Destroi(…) é o único a ter tal complexidade. Sobre o espaço o número máximo de células da fila é também O(|V|), pelo motivo já citado na implementação BFS.

3.3. Função *Main*

A leitura dos dados ocorre em tempo O(|V|), pela função le_ciclofaixas(…) de forma que atualiza-se cada peso no grafo. Também deve-se levar em conta os caminhos percorridos para cálculo do fluxo máximo. Sabemos, então, que devido ao fato de computar-se o fluxo máximo para cada caminho partindo de cada franquia e indo a cada cliente, o algortimo executa em tempo O(|F| + |C|), sendo |F| o número de franquias e |C| o número de clientes. Em termos de espaço, tem-se um vetor de franquias e um de clientes, com tamanhos respectivos de |F| e |C| (preenchidos, respectivamente, pelas funções le_franquias(…) e le_clientes(...)), mais o que pode-se considerar do grafo, O(|V|²), que é utilizado de fato nesse contexto.

4. Avaliação experimental

Para um grafo com 1000 vértices e em torno de 250000 arestas, o tempo de execução do programa é de 25,86 segundos. Aumentando o número de vértices em somas de 1000 e o de arestas em somas na casa de 1000000, o tempo chega facilmente a 12 minutos. Temos o seguinte gráfico que resume o comportamento do algoritmo principal:



O traço laranja corresponde aos segundos, eixo y secundário.

5. Observações finais

O programa foi compilado em ambiente Linux Ubuntu 14.04 LTS 64 bits com 4GB de memória RAM disponíveis e um processador Intel® CoreTM i3-2350M CPU @ 2.30GHz × 4.

Arquivos de código:

- fila_bfs.h
- fila_bfs.c
- grafo.h
- grafo.c
- tp1.c

A compilação se deu através da linha:
gcc -g -Wall -Wextra -Werror -std=c99 -pedantic grafo.c tp1.c fila_bfs.c -o tp1

Todos os testes no valgrind foram efetuados com o comando:
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./tp1 < tp1.in