UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Algoritmos e Estruturas de Dados III
Trabalho Prático 1 – Entregando Lanches

Caio Felipe Zanatelli

Belo Horizonte
21 de maio de 2017

1 Introdução

Algoritmos em grafos são essenciais para a resolução de diversos problemas em Ciência da Computação, dentre eles, problemas de fluxo máximo. Neste contexto, este trabalho apresenta uma implementação cujo o objetivo é encontrar o número máximo de ciclistas que podem sair de um conjunto de franquias, para a entrega de lanches, de forma que todos eles cheguem ao conjunto de clientes em segurança. Para tanto, é fornecido um mapa de interseções, com o conjunto de ciclofaixas que ligam cada par de interseção, se houver. Sabe-se que as ciclofaixas são todas de mão única, e que cada interseção pode conter ou não uma franquia ou um cliente, mas nunca ambos. Além disso, cada ciclofaixa possui um número máximo de ciclistas que podem trafegar por ela, de forma a promover a segurança de cada entregador.

Uma vez definido o problema e modelado em grafos, foi aplicado um algoritmo para o cálculo de fluxo máximo para a sua resolução. Como os limites de entrada são razoavelmente grandes, com $|V| \leq 1000$ e $|E| \leq 10000$, sendo V o conjunto de vértices e E o conjunto de arestas presentes no grafo, foi escolhido o algoritmo *Dinic*, o qual possui complexidade temporal $O(|V|^2|E|)$, se mostrando bastante eficiente, como será introduzido posteriormente na análise experimental da solução aqui proposta.

2 Solução do Problema

Esta seção tem como objetivo apresentar a solução implementada neste trabalho. Para tanto, será abordado brevemente o problema de fluxo máximo e as definições necessárias para a resolução do problema. Em seguida será introduzido o algoritmo *Dinic*, bem como a estrutura de código utilizada para a sua implementação.

2.1 Redes de Fluxo

Por definição, uma rede de fluxo é um grafo direcionado $G = (V, E)$, no qual cada uma de suas arestas (u, v) possui uma capacidade não negativa, dada por $c(u, v) \geq 0$. Dois vértices nesta rede são fundamentais: *source*, denotado por s, e *sink*, denotado por t. Em um problema de fluxo, o objetivo é transmitir a maior quantidada de unidades de fluxo do vértice s ao vértice t, ou seja, s é a fonte (origem) e t o sorvedouro (destino). Com isso, uma definição formal de fluxo é: um fluxo de um vértice s a um vértice t, em uma rede de fluxo $G = (V, E)$, com uma função de capacidade c, é uma função real $f : V \times V \to \mathbb{R}$, a qual satisfaz as seguintes propriedades:

- **Restrição de Capacidade**: o fluxo em uma dada aresta não pode exceder a sua capacidade, isto é, $0 \leq f(u, v) \leq c(u, v)$.
- **Conservação de Fluxo**: Não é permitido vazamentos, ou seja, o fluxo total que entra em um dado vértice deve ser igual ao fluxo total que dele sai, com exceção apenas dos vértices s e t. Portanto, em termos formais, $\forall u \in V - \{s, t\}$, é imposto que $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$. Naturalmente, se $(u, v) \notin E$, não existe fluxo de u a v, logo $f(u, v) = 0$.
- **Valor $|f|$ de um Fluxo**: Por consequência do item anterior, o fluxo que sai de s deve ser igual ao fluxo que entra em t. Por definição, o valor de um fluxo é o fluxo total que sai da fonte subtraído do fluxo que nela entra, isto é, $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$. Por fim, o problema do fluxo máximo objetiva encontrar um fluxo de maior valor em um grafo G.

Definido o conceito de uma rede de fluxo, a Figura 1 ilustra um exemplo. Em (a), a rede de fluxo é apresentada, com as capacidades das arestas definidas; e em (b) o grafo resultante, com a devida distribuição de fluxo em cada aresta. Como explicitado anteriormente, o valor do fluxo é $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$. Neste caso, $|f| = 19$, o qual também é o máximo fluxo possível neste grafo.

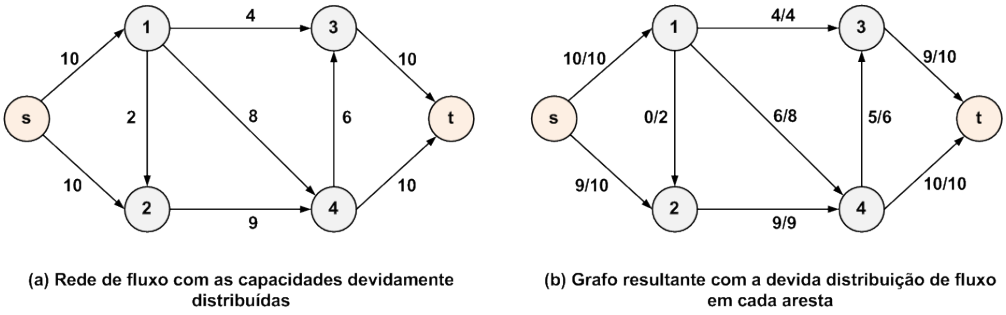

Figura 1 – Exemplo de uma rede de fluxo em (a) e de uma distribuição de fluxo em (b)

.

2.2 Rede ou Grafo Residual

Um ponto fundamental relacionado aos problemas de fluxo máximo é a questão da rede (grafo) residual (G_f), a qual representa como é possível modificar o fluxo em arestas do grafo G. A capacidade de uma aresta de G menos seu fluxo indica a quantidade de fluxo que pode ser adicionada. Se este valor é positivo, a aresta é colocada em G_f, com uma capacidade residual $c_f(u,v) = c(u,v) - f(u,v)$. Dessa forma, as únicas arestas originais de G que estão em G_f são aquelas que podem receber mais fluxo, isto é, $c_f(u,v) > 0$. Porém, se uma aresta (u,v) já possui um fluxo igual à sua capacidade, então $c_f(u,v) = 0$, sendo esta aresta não pertencente a G_f.

Além disso, o grafo residual G_f também contém arestas que não estão em G – as arestas *reversas*, as quais indicam uma possível diminuição de um fluxo. Isso é necessário pois, quando um algoritmo é aplicado a uma rede de fluxo, com o intuito de aumentar o fluxo total, pode ser necessário reduzir o fluxo de uma aresta. Assim, de forma a possibilitar tal diminuição de um fluxo positivo $f(u,v)$ em uma aresta em G, é inserido em G_f uma aresta reversa (v,u), com capacidade residual dada por $c_f(v,u) = f(u,v)$ – isto é, uma aresta que pode admitir fluxo na direção oposta, se limitando a no máximo eliminar o fluxo em (u,v). Portanto, por definição, a capacidade residual $c_f(u,v)$ é dada por

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{se} \quad (u,v) \in E \\ f(v,u) & \text{se} \quad (v,u) \in E \\ 0 & \text{caso contrário} \end{cases}$$

Por fim, a figura Figura 2 ilustra o grafo residual do exemplo apresentado pela Figura 1.

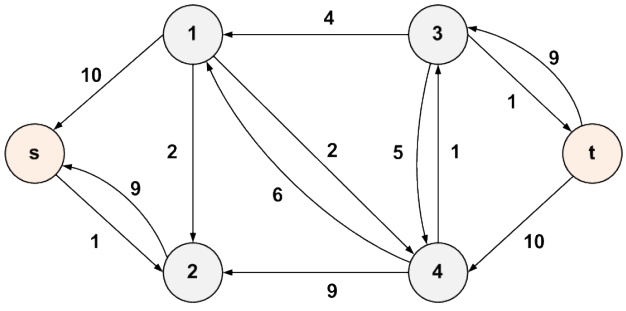

Figura 2 – Exemplo da rede residual do grafo ilutrado pela Figura 1

2.3 *Dinic* – Algoritmo para o Cálculo do Fluxo Máximo

Uma vez definidos os conceitos de fluxo máximo e rede residual, será introduzido o algoritmo implementado neste trabalho – o *Dinic*. Tal algoritmo utiliza uma busca em largura (BFS) para verificar se existe a possibilidade de transmitir mais fluxo, além de construir o grafo de níveis, no qual são atribuídos níveis a todos os nodos. Vale ressaltar, neste ponto, que o nível de um vértice é a menor distância, em termos do número de arestas, da fonte s até o vértice em questão. Com o grafo de níveis construído, são enviados múltiplos fluxos utilizando tal grafo, não se restringindo ao envio do fluxo apenas pelo caminho encontrado pela BFS, como no algoritmo Edmonds Karp, e por isso o *Dinic* é bem mais rápido. Na sequência são descritas as etapas do *Dinic*.

1. Inicializar o grafo residual G_f de acordo com o grafo G dado.
2. Fazer uma BFS em G_f, montar o grafo de níveis e verificar se é possível passar mais fluxo.
   (a) Se não é possível transmitir mais fluxo, retorne.
   (b) Enviar múltiplos fluxos em G_f utilizando o grafo de níveis, ou seja, em todos os fluxos os níveis dos vértices utilizados no caminho devem ser 0, 1, 2, ... (em ordem), de s para t, até que um fluxo bloqueador[1] (*blocking flow*) seja alcançado. Volte para o paso 2.

De forma a possibilitar uma melhor compreensão acerca do funcionamento do *Dinic*, consideremos o grafo G da Figura 1(a). Seu grafo residual G_f inicial será idêntico, pois o fluxo inicial é 0, e logo toda aresta $(u, v) \in G$ também está em G_f, com capacidade residual $c_f(u, v) = c(u, v)$ e $c_f(v, u) = 0$. Seja $Fluxo_{Total}$ o valor do fluxo máximo, com $Fluxo_{Total} = 0$, inicialmente.

- **1º iteração**: Primeiro, a BFS atribui níveis para todos os vértices no grafo residual, além de checar se existe um caminho de s para t tal que seja possível aumentar o fluxo. O grafo resultante é ilustrado pela Figura 3(a). Em seguida, os fluxos bloqueadores são encontrados através do grafo de níveis, ou seja, todo caminho de fluxo deve ter níveis 0, 1, 2, 3. Assim, são enviadas 4 unidades de fluxo no caminho s – 1 – 3 – t, 6 unidades de fluxo no caminho s – 1 – 4 – t e 4 unidades de fluxo no caminho s – 2 – 4 – t. Portanto, $Fluxo_{Total} = Fluxo_{Total} + 4 + 6 + 4 = 14$. A Figura 3($b$) ilustra o grafo obtido após a 1º iteração.

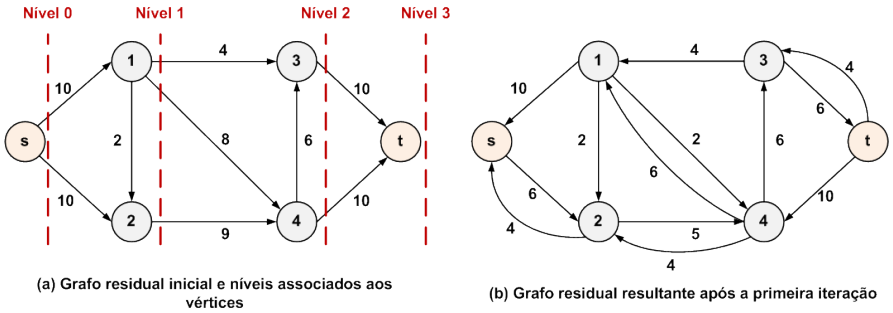

Figura 3 – Primeira iteração do *Dinic* sobre a rede de fluxo da Figura 1

- **2º iteração**: Novamente, os níveis do grafo residual – Figura 3(b) – são obtidos através da BFS, os quais são apresentados pela Figura 4(a). Note que todas as arestas que não vão de um nodo de nível N_i para um nodo de nível N_{i+1} são retiradas do grafo. Novamente, existe caminho de s para t, e então os fluxos bloqueadores são encontrados utilizando o grafo de níveis, isto é, todo caminho de fluxo deve ter níveis 0, 1, 2, 3, 4. Dessa vez, é possível enviar

[1] Um fluxo é dito bloqueador se não é possível enviar nenhum fluxo adicional utilizando o grafo de níveis. Por exemplo, se não existir mais nenhum caminho de s para t tal que este caminho seja composto por vértices de nível 0, 1, 2, ... em ordem.

apenas um fluxo: 5 unidades de fluxo no caminho $s - 2 - 4 - 3 - t$, gerando um fluxo total de $Fluxo_{Total} = Fluxo_{Total} + 5 = 19$.

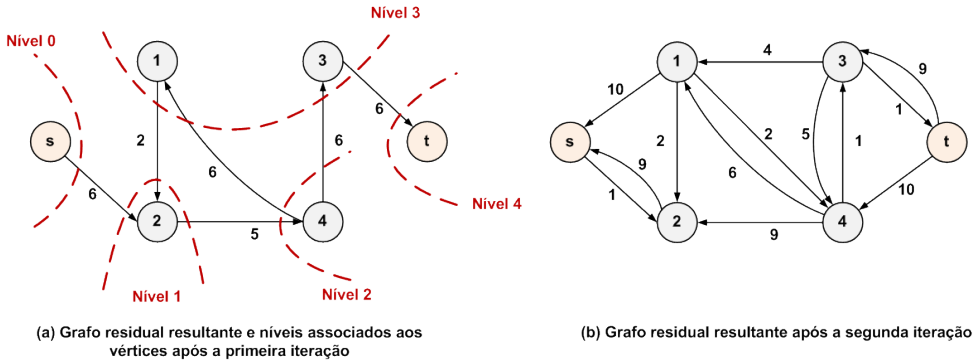

Figura 4 – Segunda iteração do *Dinic* sobre a rede de fluxo da Figura 1

- **3º iteração**: Mais uma vez, a BFS é executada no grafo residual resultante, obtido na iteração anterior – Figura 4(b) –, e um novo grafo de níveis é criado. Dessa vez, não existe mais caminho do vértice s ao vértice t no grafo residual, logo o algoritmo termina. Portanto, neste caso, o fluxo máximo é dado por $Fluxo_{Maximo} = Fluxo_{Total} = 19$.

2.4 Modelagem do Problema

Com todas as definições e explanações necessárias para a resolução do problema devidamente introduzidas, é preciso apresentar a modelagem realizada para a sua implementação.

2.4.1 Criação do Grafo

Como já abordado, o problema resolvido neste trabalho é um problema de fluxo máximo. Entretanto, o grafo a ser criado possui várias fontes (*sources*) e vários sorvedouros (*sinks*), onde as fontes são as franquias e os sorvedouros são os clientes. Neste contexto, para a aplicação do algoritmo *Dinic*, foi necessário criar dois novos vértices: *supersource* (s), o qual se conecta a todas as outras fontes (franquias); e *supersink* (t), se conectando a todos os outros sorvedouros (clientes). Em ambos os casos, utilizando o fato de que as franquias podem sempre enviar o máximo de bicicletas possível, às arestas que conectam os vértices criados foram atribuídas uma capacidade infinita. Com isso, o problema se reduz a executar o algoritmo *Dinic* para obter o fluxo máximo entre os nodos s e t, o qual fornece a quantidade de ciclistas que podem sair das franquias de forma que todos cheguem no conjunto de clientes em segurança. A Figura 5 ilustra esta modelagem, onde F_i, C_i e V_i são as franquias, os clientes e as interseções vazias, respectivamente.

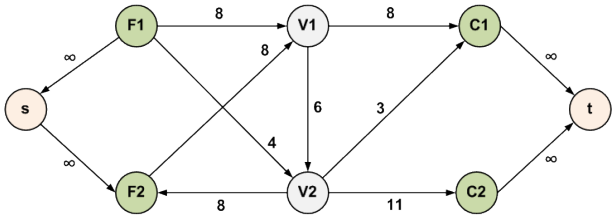

Figura 5 – Exemplo de modelagem para o problema dado

2.4.2 Listas de Adjacências

Para promover o acesso a qualquer vértice adjacente a um dado vértice em tempo constante, tendo em vista que esse acesso é bastante requisitado pelo *Dinic*, foi implementada uma lista por arranjos. Entretanto, não se sabe, inicialmente, a quantidade de arestas que cada vértice terá. Portanto, de forma a minimizar a utilização de memória, a lista foi implementada com tamanho dinâmico, isto é, quando a primeira aresta é inserida, a lista de adjacências do vértice correspondente é alocada com um tamanho inicial fixo. Na medida em que ocorrerem inserções, caso o tamanho da lista atinja sua capacidade máxima, sua capacidade é dobrada e é feita uma realocação de memória. Dessa forma, o acesso a qualquer elemento da lista é feito em tempo constante, pois se trata de uma lista indexada, e o desperdício de memória é evitado.

Além disso, de forma a promover um acesso fácil à aresta reversa de uma aresta qualquer, foi adotada a seguinte estratégia. Foi criado um vetor de arestas, o qual contém todas as arestas originais e reversas do grafo residual. Com isso, a informação que a lista de adjacências guarda passa a ser o índice para o vetor de arestas, e não as informações da aresta em si. Este vetor é indexado a partir de 0 e, sempre que uma aresta é inserida, sua aresta reversa é inserida na posição seguinte. Com essa configuração, o acesso à aresta reversa de uma aresta que está na posição i pode ser feito feito com o comando $edges[i \oplus 1]$, e vice-versa. A Figura 6 ilustra este modelo.

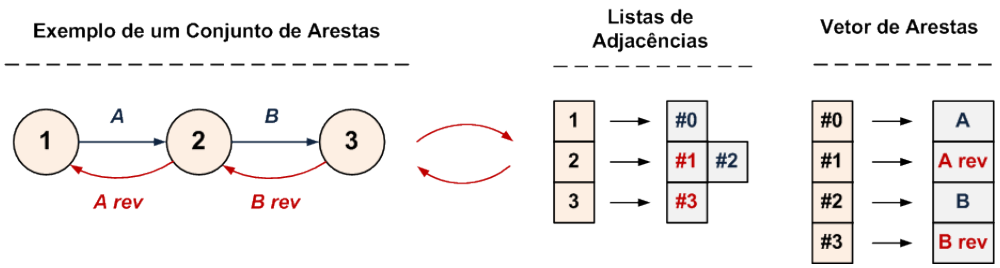

Figura 6 – Exemplo de indexação para um conjunto de arestas

2.4.3 TAD List

O TAD *List*, utilizado pelas listas de adjacências, possui os seguintes campos:

- **data**: vetor de inteiros utilizado para armazenar os elementos da lista.
- **capacity**: inteiro que armazena a capacidade da lista.
- **size**: inteiro que armazena o tamanho da lista – quantidade de elementos inseridos.

As operações fornecidas pelo TAD *List* são as seguintes:

- **initList**: inicializa uma lista vazia, alocando memória e inicializando os atributos.
- **getListSize**: retorna o tamanho da lista.
- **isListEmpty**: retorna 1 se a lista estiver vazia e 0 caso contrário.
- **pushBackList**: insere um elemento no fim da lista.
- **freeList**: libera a memória utilizada pela estrutura *List*.

2.4.4 TAD Queue

Como já abordado, o *Dinic* utiliza uma BFS para construir o grafo de níveis e checar a passagem de fluxo. Assim, o TAD *Queue*, utilizado pela BFS, foi implementado com os seguintes atributos:

- **pStart**: ponteiro do tipo *NodeQueue* para o primeiro elemento da fila.
- **pEnd**: ponteiro do tipo *NodeQueue* para o último elemento da fila.
- **size**: inteiro que armazena o tamanho da fila – quantidade de elementos inseridos.

As operações definidas para este TAD são:

- **initQueue**: inicializa a fila.
- **enqueue**: insere um elemento na fila.
- **frontQueue**: retorna um ponteiro do tipo *NodeQueue* para o primeiro elemento da fila.
- **dequeue**: remove um elemento da fila e o retorna (ponteiro do tipo *NodeQueue*).
- **isQueueEmpty**: retorna 1 se a fila estiver vazia e 0 caso contrário.
- **getQueueSize**: retorna o tamanho da fila.

2.4.5 TAD Graph

Por fim, o TAD *Graph* representa a estrutura do grafo. Nele estão presentes os atributos:

- **adjList**: vetor de lista de adjacências, do tipo *List* – discutido na seção anterior.
- **edges**: vetor de arestas (tipo *Edge*), implementado nos padrões descritos na seção anterior.
- **arrayLevel**: vetor de inteiros para armazenar o nível de cada nodo, utilizado pelo *Dinic*.
- **currentVertex**: índice (inteiro) para controlar a inserção de arestas no vetor *edges*.
- **size**: inteiro que armazena o número de vértices que o grafo possui.

As operações definidas no TAD *Graph* são as seguintes:

- **initGraph**: inicializa o grafo, fazendo as devidas alocações e atribuições.
- **addEdge**: recebe parâmetros de origem, destino e capacidade e insere a aresta no grafo.
- **getMaxFlow**: implementação do *Dinic*. Retorna o fluxo máximo entre os nodos s e t.
- **isMoreFlowPossible**: constrói o grafo de níveis através de uma BFS. Retorna 1 se existe caminho de s a t e 0 caso contrário.
- **sendFlow**: DFS responsável por enviar vários fluxos ao mesmo tempo – *Dinic*.
- **freeGraph**: libera a memória alocada para a estrutura do tipo *Graph*.

3 Análise de Complexidade

Nesta seção serão apresentadas as análises de complexidade de tempo e espaço do algoritmo implementado.

3.1 Complexidade Temporal

As funções presentes nos TADs *List* e *Queue* todas levam tempo constante pare serem executadas, exceto a função *freeQueue*. Isso é fácil de perceber, pois ambas as estruturas inserem um elemento sempre no fim. Como a fila possui um ponteiro para o último nodo, a inserção é $O(1)$. A lista, por ser implementada por arranjos, permite acesso indexado direto para a última posição, o que leva a inserção ser feita em tempo constante. Uma observação é que, eventualmente, a lista dobra sua capacidade, necessitando realocar os elementos nela contidos, entretanto, no pior caso, essa realocação será feita no máximo $log_2(|E|)$ vezes, uma vez que a capacidade sempre dobra quando a lista está lotada. Em relação à remoção de elementos, a fila leva tempo constante, uma vez que o elemento a ser removido se encontra na primeira posição. Além disso, as funções *isListEmpty*, *isQueueEmpty*, *getListSize*,*getQueueSize* e *frontQueue* também são $O(1)$, uma vez que necessitam apenas acessar um dado da estrutura e retorná-lo, como é o caso de *getSize* e *frontQueue*, ou efetuar uma operação lógica, como em *isEmpty*. As funções de inicialização (*initList* e *initQueue*) e a função *freeList*, também levam tempo constante, uma vez que precisam apenas executar atribuição e/ou alocação de memória direta e desalocação de memória, respectivamente, o que é $O(1)$. Por fim, a função *freeQueue* possui complexidade linear, pois necessita que toda a fila seja percorrida para que cada nodo indivual seja desalocado – a desalocação de memória é $O(1)$.

No TAD *Graph*, as funções *initGraph* e *freeGraph* são $O(|V|)$, uma vez que é necessário inicializar e liberar a memória, respectivamente, das listas de adjacências de todos os vértices, operações nas quais são $O(1)$, como já mencionado. A função *addEdge*, por sua vez, também é $O(1)$, pois apenas insere um elemento em uma dada lista de adjacências e uma aresta no vetor de arestas, o que é feito em tempo constante, uma vez que ambas as estruturas são indexadas.

Em relação ao algoritmo *Dinic*, este é constituído de fases, sendo que cada uma delas possui iterações para modificar o fluxo através dos menores caminhos aumentadores de tamanho fixado l. No começo de cada fase, isto é, no *loop* contido na função *getMaxFlow*, é executada uma busca em largura (BFS), para gerar o grafo de níveis G_l, de tamanho l. A BFS utiliza operações de inserção, remoção e verificação de fila vazia, as quais todas são $O(1)$. Com base nisso, como temos que a complexidade da BFS é $O(|V| + |E|)$, e como o grafo de níveis G_l é um grafo conectado, temos que a complexidade para sua construção é $O(|E|)$. O grafo de níveis é mantido durante a união de todos os caminhos aumentadores de tamanho l, até que ele desapareça, o que leva um tempo de $O(|E|)$. A estrutura de níveis de G_l permite que a execução de cada iteração da DFS da função *sendMoreFlow* seja rodada em $O(l) = O(|V|)$, pois o caminhamento neste caso só é permitido a partir de um vértice de nível i para um vértice de nível $i+1$, iniciando no vértice s e terminando no vértice t. Além disso, o grafo de níveis é estritamente podado a cada iteração da DFS, o que leva a um número de iterações em cada fase ser limitado por $|E|$. Quando o grafo de níveis é totalmente podado, não existe mais caminho de tamanho menor ou igual a l. Consequentemente, o tamanho do próximo grafo de níveis será igual ao tamanho do atual menor caminho aumentador, que é estritamente maior que l. Como o tamanho de G_l cresce a cada fase, há no máximo $|V| - 1$ fases e, quando o algoritmo para, o fluxo atual é máximo. Portanto, o tempo de execução do algoritmo *Dinic* é $O(|V|^2|E|)$.

3.2 Complexidade Espacial

Como abordado em seções anteriores, a representação do grafo utiliza uma lista de adjacências, a qual armazena índices que informam onde as arestas estão armazenadas no vetor de arestas.

Dessa forma, como há $|E|$ arestas, a quantidade de elementos armazenados pelas listas de adjacências, no pior caso, é $O(|E|)$. Como cada vértice possui uma lista de adjacência, sabendo que existem $|V| + 2$ vértices ($|V|$ vértices mais a *supersource* e *supersink*), a complexidade total deste armazenamento é $O(|V|)$. O vetor de arestas, por sua vez, possui mais arestas, pois armazena também as arestas reversas. De fato, o tamanho deste vetor é $2 \cdot (|E| + |F| + |C|)$, pois, além das arestas dadas pelo problema, é necessário acrescentar $|F|$ arestas para conectar a *supersource* às franquias e mais $|C|$ arestas para conectar os clientes à *supersink*. Portanto, a ordem de complexidade espacial é dada por $O(|V|) + O(|E|) + O(2 \cdot (|E| + |F| + |C|))$, e como $|F| + |C| \leq |V|$ e $|E| > |V|$, temos que a complexidade espacial deste algoritmo é $O(|E|)$.

4 Análise Experimental

Como apresentado na seção anterior, o pior caso do algoritmo *Dinic* é $O(|V|^2|E|)$. Portanto, de forma a ilustrar melhor seu desempenho, foram elaborados dois casos de teste. No primeiro, a quantidade de vértices foi fixada, e um grafo foi gerado variando suas arestas. Para este modelo específico, foram gerados um conjunto de 50 testes, fixando a quantidade de vértices em 2000. Então, foram criadas arestas de um dado vértice i para 10 a 30 outros vértices do grafo, atribuindo uma capacidade aleatória, até completar o número de arestas desejado. A utilização dos vértices é gradual, ou seja, os primeiros 20 vértices são conectados a outros 20 vértices, os próximos 20 são conectados a outros 20, e assim sucessivamente. Como o conjunto de arestas criado foi na casa de 10^5 a 10^6, as franquias e os clientes foram fixados em 10 vértices, o que gera uma diferença desprezível nos resultados. A Figura 7 ilustra o resultado obtido. Como os testes são bastante exaustivos, em vários momentos há picos no gráfico. Entretanto, percebe-se que os pontos se ajustam de forma que podem ser aproximados por uma reta, o que condiz com a análise de complexidade pois, se $|V|$ é constante, o comportamento assintótico é próximo de uma reta.

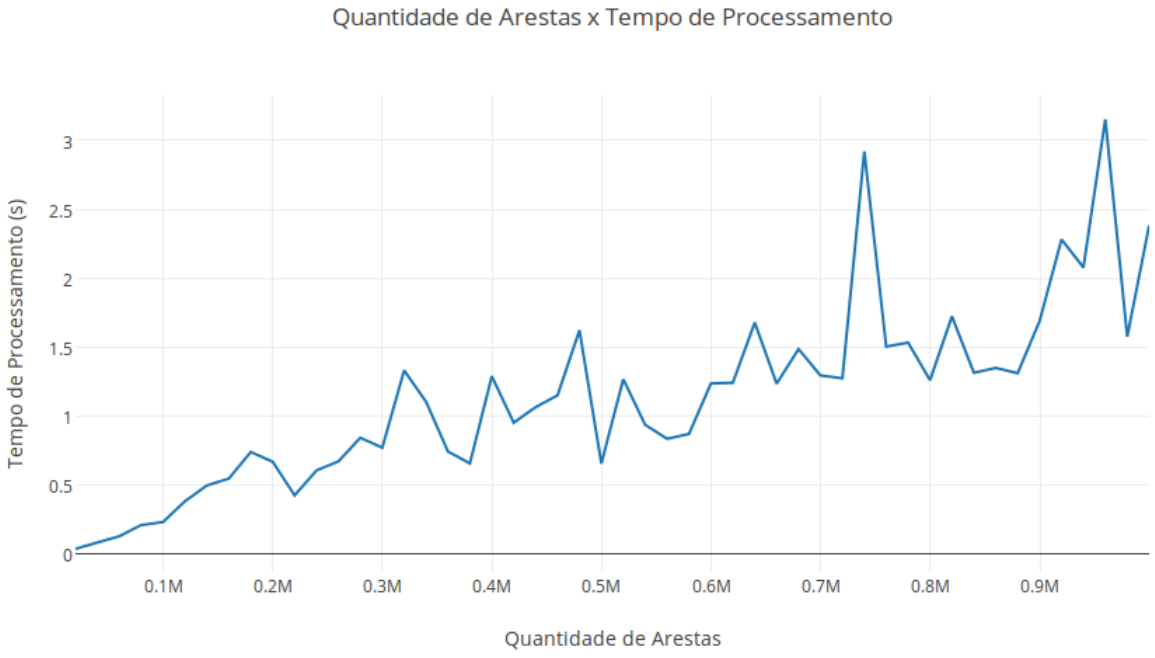

Figura 7 – Conjunto de testes com variação na arestas e vértices fixos

O segundo conjunto de testes executados foi variar o número de vértices. Entretanto, de forma a visualizar melhor o comportamento assintótico do algoritmo, foi gerado um grafo completo para cada variação da quantidade de vértices, evitando que uma má distribuição das arestas afete o resultado da análise. Como observado na Figura 8, o resultado obtido também condiz com a complexidade temporal avaliada anteriormente. Ou seja, como o *Dinic* possui complexidade $O(|V|^2|E|)$, e a quantidade de vértices foi variada, o comportamento assintótico da curva é quadrático.

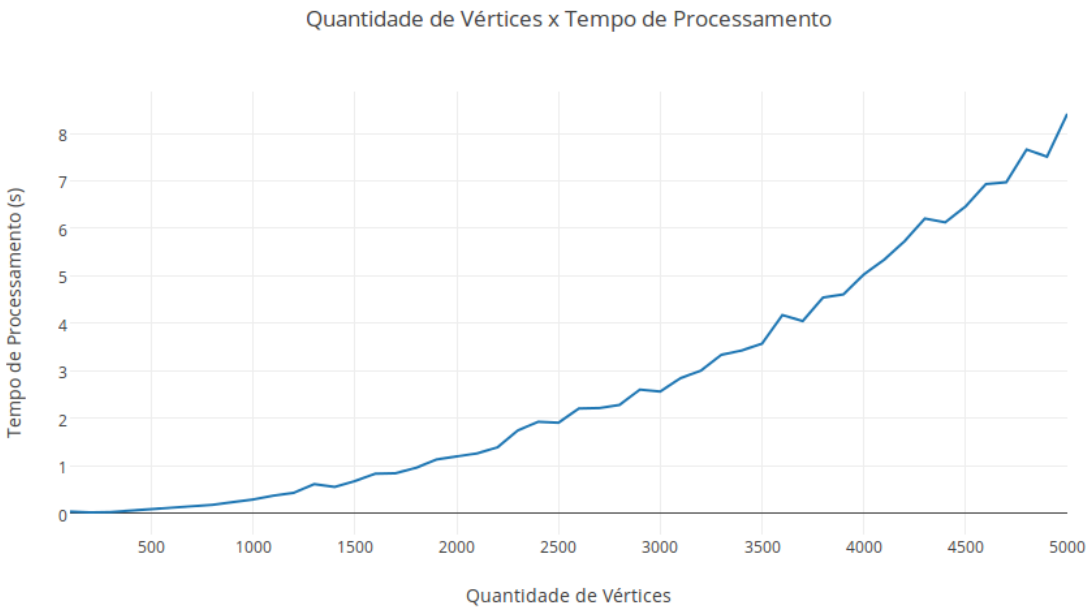

Figura 8 – Conjunto de testes com variação na quantidade de vértices

5 Conclusão

Este trabalho apresentou uma solução para o problema de Fluxo Máximo dado, o qual possui várias fontes e vários sorvedouros. Para tanto, foi implementado o algoritmo de *Dinic*, o qual possui complexidade temporal $O(|V|^2|E|)$, se mostrando bastante eficiente na prática, tendo em vista os limites elevados de vértices e arestas, com $|V| \leq 1000$ e $|E| \leq 10000$.

6 Referências Bibliográficas

Cormen, Thomas H., and Thomas H. Cormen. Introduction to Algorithms. Cambridge, Mass: MIT Press, 2001.

Dinitz. Dinitz' Algorithm: The Original Version and Even's Version. Dept. of Computer Science, Ben-Gurion University of the Negev. Acesso em 21 maio 2017. Disponível em: https://www.cs.bgu.ac.il/ dinitz/Papers/Dinitz_alg.pdf

Geeks for Geeks. Dinic's algorithm for Maximum Flow. Acesso em 21 maio 2017. Disponível em: http://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/