Trabalho Prático 1 - Grafos

Projeto e Análise de Algoritmos, 2019-1

Matheus Henrique do Nascimento Nunes
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais
mhnnunes@dcc.ufmg.br

1 INTRODUÇÃO

Batalhas intergaláticas são acontecimentos constantes em um futuro distante da humanidade. Frotas de naves espaciais são enviadas para combater forças inimigas em uma guerra cruel e infindável. O planejamento de um ataque surpresa a uma frota inimiga envolve duas atividades de suma importância: o reconhececimento de naves da frota inimiga e o cálculo do tempo de vantagem (tempo mínimo até todos os tripulantes das naves atingirem seu posto de trabalho correto e se prepararem para o ataque). Deseja-se que estas duas atividades sejam realizadas da melhor forma possível, logo o trabalho de alunos de Projeto e Análise de Algoritmos da UFMG foi requisitado.

Sabe-se que as frotas inimigas são divididas em quatro tipos de naves, ilustrados na figura 1: *Bombardeiros* (1a), *Reconhecimento* (1b), *Transportadoras* (1c) e *Frigatas* (1d). Algumas restrições são impostas a estas naves, por exemplo: em uma nave bombardeira, cada fileira de postos deve possuir no mínimo dois postos de combate; já nas naves de reconhecimento e transporte, o teleporte só é possível entre postos diretamente adjacentes.

Sabe-se também que estas naves são organizadas em postos de combates, de forma que cada tripulante pode ocupar apenas um posto por vez e, para se movimentar entre os postos, os tripulantes devem utilizar o sistema de teleporte nativo da nave. Este sistema de teleporte funciona trocando dois tripulantes por vez, entre postos de combate.

O objetivo do trabalho prático, portanto, é: dado um conjunto de pontos, representando as naves das frotas inimigas e os postos de combate presentes em cada uma delas, conexões entre estes pontos, representando as possibilidades de teleporte entre os postos, e um conjunto de pares (p_i, p_j) representando teleportes a serem realizados entre dois pontos em uma mesma nave:

- Reconhecer e classificar o tipo de cada uma das naves presentes na frota inimiga
- Calcular o tempo de vantagem para a frota aliada (tempo mínimo para que todos os teleportes especificados pelos pares de pontos (p_i, p_j) sejam realizados)

2 FORMALIZAÇÃO

O problema do trabalho prático pode ser dividido em duas etapas, mas o *framework* utilizado nas duas é o mesmo. O conjunto de pontos fornecidos como entrada pode ser interpretado como um conjunto de vértices V de um grafo não orientado e não ponderado $G = (V, E)$, e a sequência de conexões entre estes pontos (pares de pontos (u_i, v_i)) pode ser interpretada como as arestas E entre os vértices de V.

DFS (G)
  Input : Grafo $G = (V, E)$
  Output : conjunto de componentes conexas S
  $S \leftarrow \emptyset$;
  $\forall v_i \in V$, marca v_i como não visitado ;
  foreach $v_i \in V$ **do**
    $DFSVisit(G, v_i)$;
    $S \leftarrow S \cup \{v_i \in V\}$, tal que v_i é descendente de v;
  end

Algoritmo 1: Pseudocódigo da Busca em Profundidade (DFS)

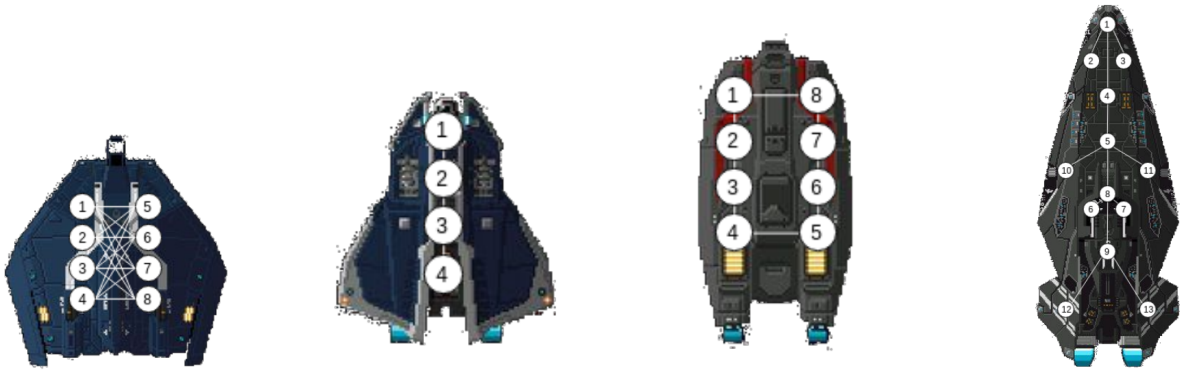
Tratando o problema desta maneira, pode-se reduzir o problema de reconhecer as naves a dois problemas conhecidos em Teoria de Grafos: **identificar componentes conexas** em um grafo, e **classificar cada uma das componentes** em quatro tipos de grafos:

- **Grafo bipartido completo** (onde os nós podem ser divididos em duas partições e nós de uma mesma partição não possuem arestas entre si mas são ligados a todos os nós da outra partição)[1], correspondendo à identificação das naves bombardeiras, como as da figura 1a
- **Grafo caminho** (onde os nós estão dispostos linearmente)[2], correspondendo à identificação de naves de reconhecimento, como as da figura 1b
- **Grafo cíclico** (onde os nós estão dispostos linearmente, com os nós das extremidades ligados entre si)[3], correspondendo à identificação das naves de tranporte, como as da figura 1c
- **Árvore** (grafo simples, não direcionado, conexo e acíclico)[4], correspondendo à identificação de naves frigata, como as da figura 1d

Pode-se observar que uma árvore pode se degenerar em um grafo caminho, se cada nó possuir apenas um filho. Porém foi garantido nas restrições do trabalho que "Uma frigata **nunca** poderá ter a mesma estrutura interna que uma nave de reconhecimento", logo sabe-se que não há ambiguidade no reconhecimento das naves.

O reconhecimento de componentes conexas em um grafo não direcionado e não ponderado é uma aplicação direta do algoritmo de **busca em profundidade**, visto em sala de aula. A cada chamada da função $DFSVisit(G, v)$ na busca em profundidade, descrita pelo algoritmo 1, marca-se o vértice v como visitado e percorre-se a lista de vértices adjacentes a v, chamando a $DFSVisit$ recursivamente para os vizinhos ainda não visitados. Ao final temos um

[1] http://mathworld.wolfram.com/CompleteBipartiteGraph.html
[2] http://mathworld.wolfram.com/PathGraph.html
[3] http://mathworld.wolfram.com/CyclicGraph.html
[4] http://mathworld.wolfram.com/Tree.html



(a) Nave Bombardeira **(b) Nave de Reconhecimento** **(c) Nave de Transporte** **(d) Nave Frigata**

Figura 1: Tipos de Naves da Frota Inimiga

conjunto S onde cada $s_i = (V_{s_i}, E_{s_i}) \in S$ corresponde a um conjunto de vértices $V_{s_i} = \{v_1, v_2, ..., v_n\}$ e um conjunto de arestas $E_{s_i} = (v_i, v_j), \forall (v_i, v_j) \in V_{s_i}$ que formam uma componente conexa em G. A prova de corretude deste algoritmo pode ser encontrada neste link .

O próximo passo é a **identificação do tipo de cada nave** (componente conexa). Desenvolveu-se um algoritmo baseado nos quatro tipos de grafo disponíveis, para classificar os vértices de cada componente em um dos quatro tipos (árvore, caminho, ciclo, bipartido completo). O algoritmo 2 ilustra o pseudo-código utilizado nesta etapa. Utiliza-se conceitos relacionados a cada um dos tipos de grafos descritos anteriormente, que representam os tipos de nave. Sabe-se que um grafo cujos vértices possuem grau menor ou igual a dois, só pode ser um ciclo ou um caminho. Caso o número de arestas deste grafo seja igual ao numero de vértices, sabe-se que o grafo é um ciclo. Caso contrário, é um grafo caminho. Caso o grau máximo dos vértices seja maior do que dois, verifica-se se o número de arestas no grafo é igual ao número de vértices menos 1. Neste caso, tem-se uma árvore. Caso contrário, pela especificação, este grafo só pode ser bipartido completo.

Após a identificação e contagem das naves, o próximo passo é o calculo do **tempo de vantagem**. A resposta trivial para o menor tempo possível em que seja possível trocar os tripulantes de lugar é zero. Porém a especificação do trabalho pede uma resposta não trivial. O problema de trocar a posição dos tripulantes no menor tempo possível pode ser reduzido de maneira direta ao problema de "Token Swapping on Graphs" [2]. A definição do problema é a seguinte: dado um grafo $G = (V, E)$ com n vértices, e uma sequência de n *tokens* $1, 2, ..., n$ distriuídos em vértices distintos de G, deseja-se transformar, com a menor sequência possível de operações, a distribuição inicial de tokens f_0 em uma distribuição final pré-definida f.

Yamanaka et al. mostram em seu paper [2] que a complexidade computacional de "Token Swapping" não é conhecida para grafos gerais. Porém, para alguns tipos especiais de grafos, os autores apresentam um algoritmo 2-aproximativo para a resolução do problema. Os tipos de grafo para os quais o algoritmo funciona são os grafos vistos no trabalho prático: caminho, ciclo, árvore e bipartido completo. O algoritmo proposto pelos autores é bastante simples, e baseado no lema 1.

```
1  checkCC (S)
       Input  : S conjunto de componentes conexas
       Output : T = {t_i ∀ s_i ∈ S}, onde t_i é o tipo de cada
                componente s_i
2      T ← ∅;
3      foreach s_i ∈ S do
4          if max(d(v_i)) ≤ 2, ∀v_i ∈ s_i) then
5              if |E_{s_i}| == |V_{s_i}| then
6                  t_i ← ciclo;
7              end
8              else
9                  t_i ← caminho;
10             end
11         end
12         else
13             if |E_{s_i}| == |V_{s_i}| − 1 then
14                 t_i ← arvore;
15             end
16             else
17                 t_i ← bipartido;
18             end
19         end
20         T ← T ∪ {t_i};
21     end
```

Algoritmo 2: Pseudocódigo da identificação de naves

Lema 1. $OPT(f_0) \geq \frac{1}{2}\Delta(f_0)$

Onde $\Delta(f_0) = \sum_{\forall (u,v) \in swaps} \delta(u, v)$, $\delta(u, v)$ representa o menor caminho entre os vértices u e v, e $swaps$ é o conjunto de pares de vértices (u_i, v_i) que trocam de posição.

Pelo lema 1 sabemos que $\frac{1}{2}\Delta(f_0)$ é um limitante inferior para o valor ótimo, logo este valor foi utilizado para o algoritmo desenvolvido no trabalho. Calcula-se $\frac{1}{2}\Delta_i(f_0), \forall s_i \in S$, e o menor valor $min_{\forall s_i \in S}(\frac{1}{2}\Delta_i(f_0))$ é retornado.

3 METODOLOGIA

O objetivo desta seção é apresentar a metodologia utilizada na implementação do trabalho prático. A partir do formalismo introduzido na seção 2, apresenta-se as decisões de implementação e as estruturas de dados utilizadas. Será fornecido também uma análise assintótica de complexidade das principais funções do programa.

O trabalho foi implementado utilizando a linguagem C++, logo, a complexidade das operações será analisada utilizando como base os valores descritos no cppreference[5]. A estrutura principal utilizada é uma classe Graph, que internamente contém: um array vector<Node> nodes (representando os nós do grafo), um array vector<char> visited que indica se cada nó foi visitado (foi utilizado char pois este tipo consome apenas um byte de memória), e uma variável vector< vector<int> > parent para salvar os nós ancestrais de cada nó do grafo. A struct Node, por sua vez, possui um array de inteiros vector<int> adj contendo as posições dos nós adjacentes ao atual, uma variável unsigned int position indicando a posição do nó na componente, e uma variável int CC indicando em qual componente o nó se encontra.

Inicialmente, um grafo vazio é criado, e o vetor de nós é alocado com o número de nós lido da entrada ($O(|V|)$). Em seguida, a leitura das arestas é realizada, e um inteiro é adicionado em cada uma das listas de adjacência dos nós da aresta, utilizando adj.push_back (complexidade $O(1)$ amortizada). Portanto, ao final da adição das $|E|$ arestas, temos no total uma complexidade $O(|V| + |E|)$ para a leitura e montagem do grafo, em tempo e espaço.

A fim de **dividir o grafo em componentes conexas**, foi utilizado o algoritmo 1 (busca em profundidade). A complexidade de tempo da busca em profundidade é comprovadamente $\Theta(|V|+|E|)$ [1]. Os dados das componentes foram salvos em um array vector< struct CC >, onde cada objeto de struct CC contém: um array vector<int> nodes_in_cc contendo os índices dos nós presentes na componente, uma variável int nvertices contendo o número de nós neste array, uma variável int number_of_arcs contendo o numero de arestas na componente, uma variável int maxdegree contendo o maior grau entre os nós na componente, uma variável start_node contendo o nó inicial da componente (útil para caminhos), uma variável int mintime que salva a menor distância entre os pares de nós na componente, e uma variável CC_type type indicando o tipo de grafo da componente (CC_type é um enum com as opções {path, cycle, tree, bipartite}). A complexidade de memória deste array é dominada pelo array de nós em cada objeto CC. Como cada nó do grafo está em exatamente uma componente, e os outros elementos da estrutura são $O(1)$ (constantes) em memória, o array vector< struct CC > utilizará $O(|V|)$ memória extra.

O **reconhecimento do tipo de cada componente** (nave) foi realizado seguindo o algoritmo 2. O algoritmo percorre cada uma das componentes $s_i \in S$, acessando características que são obtidas em $O(1)$ (tempo constante), como descrito no parágrafo anterior. Sabemos pela especificação do trabalho pratico que o número k de componentes é limitado superiormente pelo número de vértices $|V|$. Logo, o algoritmo executa em tempo $O(k) << O(|V|)$.

Um passo intermediário é realizado, para permitir que este cálculo do tempo de vantagem seja feito de maneira mais eficiente. Foi realizado um **pré-processamento nas componentes**, de forma diferente para cada tipo de componente, identificado no passo anterior. Foram realizadas as seguintes ações:

- **Caminho e ciclo**: foi atribuído um valor inteiro para cada um dos nós, sequencialmente. Desta forma, para calcular a distância entre dois nós, basta calcular o valor absoluto da diferença entre os dois identificadores dos nós. A atribuição dos identtificadores para o grafo caminho foi realizada utilizando uma execução da DFSVisit (presente no algoritmo 1), partindo de um nó de grau 1 (uma das pontas do caminho). Para o grafo ciclo, o mesmo procedimento foi utilizado, porém partindo de qualquer um dos nós. A distância entre os nós 1 e 3 na figura 1b seria 2, por exemplo. A distância entre os nós em um ciclo, entretanto, é calculada usando a fórmula $d(a, b) = min(|a-b|, n-|a-b|)$, onde n é o número de nós do ciclo. A fórmula é utilizada pois em um ciclo, a menor distância pode passar por um dos dois caminhos possíveis a cada nó. Usando como exemplo a figura 1c, $d(1, 3) = 2$, mas $d(1, 7) = 2$ também. Complexidade do pré-processamento $O(|V|)$ em tempo e espaço. Complexidade da consulta: $O(1)$ em tempo e espaço.
- **Bipartido completo**: inicialmente seleciona-se um nó aleatório u da componente, e atribui-se o identificador 0 para este nó. Para todos os v_i adjacentes a u, atribui-se o identificador 1. A distância entre dois nós será 1 caso os identificadores sejam diferentes (estão cada um em uma partição da componente), e 2 caso os identificadores sejam iguais (estão na mesma partição, logo a dois passos de distância). Complexidade do pré-processamento $O(|V|)$ em tempo e espaço. Complexidade da consulta: $O(1)$ em tempo e espaço.
- **Árvore**: a fim de calcular a distância entre dois nós em uma árvore, foi utilizada uma técnica chamada de LCA (Lowest Common Ancestor), com consultas feitas utilizando Binary Lifting[6]. Esta abordagem calcula o nó da árvore que é o ancestral mais próximo dos dois nós (a, b) que estão na consulta, e então calcula-se a distância dos dois nós a este ancestral, e retorna a soma como resposta. Utiliza-se a matriz vector< vector<int> > parent como memória, com complexidade $O(|V|log(|V|))$ em espaço. A complexidade de tempo do pré-processamento também é $O(|V|log(|V|))$ (executar DFS construindo a matriz parent), e a consulta lca(a, b) tem complexidade log(|V |).

O último passo do algoritmo é o **cálculo do tempo de vantagem**. Este cálculo é feito percorrendo cada uma das componentes $s_i \in S$, e calculando a menor distância entre seus nós que serão trocados de posição. Aplica-se o lema 1, e o menor valor encontrado entre todas as componentes é retornado. A complexidade de tempo desta etapa é no máximo $O(|V|log(|V|))$, quando todas as $O(|V|)$ consultas são realizadas em uma árvore, cujo tempo de consulta é $O(log(|V|))$.

[5] https://en.cppreference.com/w/

[6] https://cp-algorithms.com/graph/lca_binary_lifting.html

Pode-se concluir que a complexidade assintótica total do programa é a soma da complexidade dos passos intermediários, logo $O(2(|V|+|E|)+k+2(|V|log(|V|)))$. A mesma análise serve para a complexidade em memória: $O((|V|+|E|)+|V|+|V|log(|V|))$. Caso o grafo seja denso ($|E| \approx |V|^2$), as complexidades serão dominadas pelo primeiro termo, e serão $O(|E|) = O(|V|^2)$. Caso contrário, serão dominadas pelo último termo, sendo $O(|V|log(|V|))$.

4 ANÁLISE EXPERIMENTAL

O objetivo desta seção é apresentar dados que demonstram o desempenho do código entregue como resultado do trabalho prático. Os experimentos foram realizados em uma máquina com processador Intel® Core™ i5-6200U, de 8 núcleos a $2.30GHz$. A máquina opera a 64 bits, e possui 8GB de memória RAM DIMM DDR4. O sistema operacional utilizado foi o Linux Mint Sonya 18.2. O trabalho foi implementado na linguagem C++11.

As instâncias de teste foram geradas utilizando um gerador de grafos aleatórios disponível online[7], utilizando 10 *seeds* diferentes[8] para o gerador de numeros pseudo-aleatórios. Cinco grafos de cada tipo e uma versão completa com todos os tipos de grafo foram geradas, de cinco tamanhos diferentes de nós[9] e arestas[10]. Utilizando todas as combinações destes parâmetros, o número total de instâncias geradas foi 320.

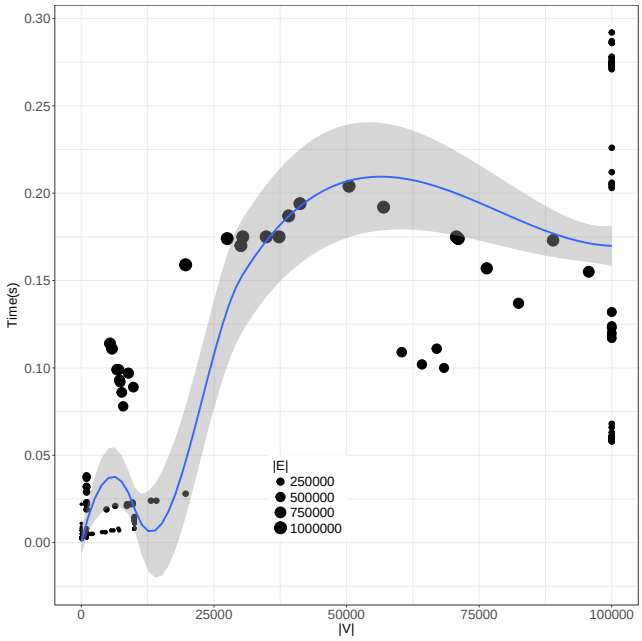


Figura 2: Gráfico de tempo de execução por número de nós

A figura 2 apresenta os resultados do tempo de execução para as instâncias de teste. O eixo x representa o número de nós da instância e o eixo y representa o tempo de execução do programa. O tamanho do ponto representa o número de arestas da instância. Uma regressão LOESS (*Locally Estimated Scatterplot Smoothing*)[11] foi ajustada aos pontos, a fim de revelar alguma tendência presente na relação. Pode-se observar que há um crescimento acelerado do tempo de processamento quando o numero de nós está entre 25 e 50 mil, porém este crescimento é causado pelo aumento no número de arestas (pode-se observar pelo tamanho do ponto que o número de arestas está próximo de 10 milhões). O comportamento da curva é aproximadamente quadrático em alguns pontos, mas sub-quadrático em outros, como esperado após a análise de complexidade teórica.

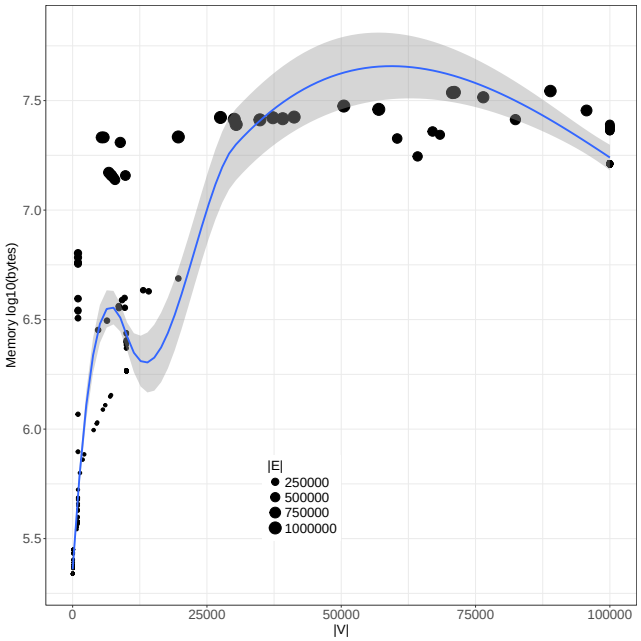


Figura 3: Gráfico da memória utilizada por número de nós e arestas

O gráfico da figura 3 apresenta os resultados para a memória utilizada pelo programa durante a execução. Pode-se observar que a curva gerada pela regressão é bem parecida com a da figura 2, assim como esperado após a análise teórica realizada na seção 3. O resultado observado na prática é bem similar ao resultado teórico: para grafos densos ($|E| \approx |V|^2$, ou seja, muito mais arestas do que vértices) o comportamento do algoritmo será aproximadamente quadrático no número de nós.

Conclui-se que o objetivo do trabalho prático foi cumprido: implementar e analisar algoritmos em grafos. A resolução do problema passou por algoritmos exatos e algoritmos aproximativos, e a análise de complexidade teórica foi comprovada empiricamente.

REFERÊNCIAS

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[2] Katsuhisa Yamanaka, Erik D Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. 2015. Swapping labeled tokens on graphs. *Theoretical Computer Science* 586 (2015), 81–94.

[7] https://github.com/cgpimenta/util
[8] {1, 6, 16, 18, 33, 57, 68, 75, 80, 99}
[9] {10, 20, 100, 1000, 10000, 100000}
[10] {100, 1000, 10000, 100000, 1000000}
[11] https://blogs.sas.com/content/iml/2016/10/17/what-is-loess-regression.html