Projeto e Análise de Algoritmos - 2019/1
Trabalho Prático - Grafos
Space Invaders

Lucas G. S. Félix [1]

[1]Departamento de Ciência da Computação - UFMG

lucasgsfelix@gmail.com

1. Introdução

Dada a expansão da internet o uso de dados se tornou cada vez maior. Por meio disso, a utilização de estrutura de dados e modelagens distintas visando a descoberta de informação vem sendo aplicada por pesquisadores e empresas. Uma dessas modelagens aplicadas é a modelagem por meio de grafos, as quais são utilizadas em diversos contextos, como mineração de dados, otimização, redes complexas, estudos matemáticos e sociológicos entre outros.

No problema especificado no Trabalho Prático (TP) da disciplina de Projeto e Análise de Algoritmos (PAA), *Space Invaders*, é apresentado o problema de batalha entre duas raças alienígenas as quais possuem diferentes naves para invasão de um sistema solar.

Visando realizar o reconhecimento destas naves e calcular o tempo de vantagem para realizar o contra ataque, é proposto aqui um algoritmo, o qual utiliza de aplicações clássicas de grafos para realização de tais tarefas.

O método completo é dividido em duas etapas. A primeira etapa realiza a identificação de dos quatro tipos de nave, onde cada nave possui características únicas que permitem realizar a identificação sua identificação. Por meio desta identificação pode-se saber quantas naves há no ataque e quais os tipos das naves. Enquanto a segunda parte consiste no cálculo de limite inferior não trivial, o qual informa a distância mínima de vantagem que se tem da frota inimiga.

Nas seções posteriores são apresentadas a Solução Proposta 2, a Análise de Complexidade 3, a Análise Teórica 4, e por último é apresentada a Conclusão 5.

2. Solução Proposta

Nesta seção são apresentadas as soluções para cada uma das etapas do TP. Na primeira parte é apresentada a técnica para identificação das naves. Já a segunda parte é apresenta estratégia utilizada para cálculo do tempo de vantagem.

Antes da apresentação dos métodos para identificação das naves e cálculo do tempo de vantagem, será apresentado como foi feito a modelagem do grafo após a leitura. As implementações deste trabalho foram feitas na linguagem C++, desta linguagem foi utilizada a estrutura de dados *Vector*, a qual é semelhante a uma lista encadeada e se comporta de maneira dinâmica, podendo possui comportamento também semelhante a Pilhas (First In - Last Out) e Filas (*First In - First Out*). Vale destacar que a classe é padrão da linguagem utilizada.

2.1. Leitura e Modelagem dos Dados

Na etapa de leitura e modelagem de dados foi utilizado um *buffer*. O *buffer* armazena os dados da base de entrada e é posteriormente utilizado para a modelagem do grafo.

Inicialmente os dados estão armazenados na base em colunas duplas, onde na primeira linha está o número de postos de combate (N) e quantidade total de teleportes possíveis dentro de uma nave, (M). A partir da segunda, as $M + 1$ primeiras linhas irão conter os teleportes possíveis que formam as diversas naves presentes no grafo. E as outras N últimas linhas, identificam tripulantes que devem retornar a seu posto de combate. Destaca-se que $N + M + 1$ será o número total de linhas da base de dados.

Na solução proposta, inicialmente armazenamos os dados da base em um *buffer* linear, o qual depois será utilizado para produzir os grafos utilizados. Para a implementação proposta foi utilizada a representação do grafo por meio de uma lista de adjacência. Ao modelar o grafo temos $G = (V, E)$, onde V sãos vértices do grafo e E são arestas.

Ressalta-se que aplicando este tipo de representação há economia de memória, dado que a mesma gasta apenas $O(V + E)$ para armazenamento de vértices e arestas. Já uma modelagem por meio de matriz de adjacência gasta $O(V^2)$. Destaca-se, também, que no pior caso a representação por lista de adjacência terá complexidade $O(V^2)$ o qual só ocorrerá quando o grafo for denso, ou seja, $E = V^2$.

Considerando a modelagem de dois grafos G e G' onde G é um grafo não-orientado que representa as $M + 1$ primeiras linhas, e G' é um grafo não orientado que representa as N últimas linhas.

Temos que G é um grafo não-orientado, pois representa as ligações de postos de combate que representam uma nave. Assim, dá mesma forma que o posto $v \in V$ tem ligação com $u \in V$, u tem ligação com v, sendo uma ligação bidirecional.

Já o grafo G' é um grafo direcionado pois modela um tripulante que está em um posto de combate $u \in V$, o qual deve retornar ao seu posto de combate $v \in V$ original. Assim, a ligação é unidirecional, pois apenas o tripulante que está em u deve retornar para v e não o contrário. Por fim, se u e v forem iguais, então o tripulante já está no posto de comando o qual deveria estar.

A Figura 1 mostra como era representada inicialmente a base de dados, como a mesma é armazenada no *buffer*, e como o *buffer* é transformado em um grafo não orientado para as $M + 1$ primeiras linhas, e um grafo orientado para as N últimas linhas.

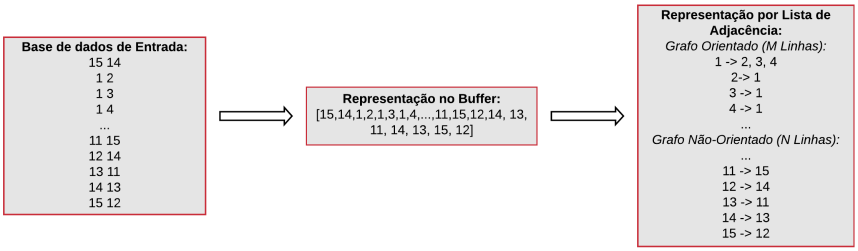


Figura 1. Leitura e modelagem do Grafo

2.2. Identificação das Naves

Após a modelagem dos grafos G e G' mostrou-se possível a identificação das naves presentes em G. Para isso, foi utilizada uma abordagem que utilizava como algoritmo principal a busca em profundidade (*Deep First Search (DFS)*). O algoritmo DFS é um algoritmo essencial de busca em grafo, sendo que neste trabalho foi realizada uma implementação com base na proposta de [Cormen et al. 2009].

Por meio da aplicação da busca em profundidade foi possível calcular os tempos de descoberta e fechamento, o ascendente do vértice, além da realização do cálculo de componentes, os quais são essenciais para identificação correta das naves.

A componente de um grafo G é um subgrafo c onde $c \in G$. No contexto do *Space Invaders*, cada componente é uma nave a qual possui uma estrutura diferente que nos permite identificar qual tipo de nave a componente c é. Vale ressaltar que um vértice $v \in c_i$, não pode pertencer a nenhuma outra componente, a mesma relação vale para as arestas. Além disso, como temos um grafo não-orientado, a componente que representa uma nave é uma componente conexa, ou seja, todo vértice $v \in V$ possui um caminho para $u \in V$, onde $uev \in c$.

O pseudo-código do algoritmo de busca em profundidade 1, apresenta a ideia por trás da implementação realizada. Por meio do pseudo-código é possível ver como é feito o cálculo de componentes os quais irão facilitar na identificação de tipo das naves.

Algorithm 1 Busca em Profundidade

```
1:  procedure DFS(grafo G)
2:      for each v ∈ G do
3:          if v → cor = "BRANCO" then
4:              DFS_VISIT(G, v, c)
5:              componente ← componente + 1
6:              c → id ← componente
1:  procedure DFS_VISIT(grafo G, vertice v, componente c)
2:      v → cor ← "CINZA"
3:      tempo ← tempo + 1
4:      v → d ← tempo
5:      for each u ∈ v → ADJ do
6:          if u → cor = "BRANCO" then
7:              u → componente ← componente
8:              c → num_nodes ← c → num_nodes + 1
9:              DFS_VISIT(G, u)
10:         c → num_edges ← c → num_edges + 1
11:     v → cor ← "PRETO"
12:     v → t ← tempo
```

Inicialmente, tem-se que todos os vértices possuem $cor = "BRANCO"$, mostrando que nenhum deles ainda foi visitado. Durante a estrutura de repetição da linha 2 do procedimento DFS, cada vértice de $cor = "BRANCO"$ é iterado, chamando o procedimento DFS_VISIT. No procedimento DFS_VISIT a cor do vértice é modificada, ficando "$CINZA$", o que significa que o vértice já foi visitado, entretanto ainda não foi processado. Assim, o vértice é processado visitando cada um de seus vizinhos, e definindo em qual componente o vértice está presente, tempo de descoberta e fechamento.

Além disso, uma classe componente armazena informações essenciais da componente c_i, como número de vértice e arestas, o identificador da componente.

Uma nova componente é definida a cada iteração do método DFS, sendo como definido, cada nave é uma componente conexa. Desta maneira, o procedimento $DFS_V ISIT$ sempre visita todos os vértices v que estão presentes em uma componente c_i. Em outras palavras, todos os postos de combate de uma nave.

Com essas informações, e com os dados definidos, é possível passar para etapa de identificação das naves. Existem quatro tipos de naves, as quais possuem diferentes estruturas *Reconhecimento, Frigatas, Bombardeiros e Transportadores*. Abaixo são apresentadas cada a descrição de cada uma das naves e como foi feito a identificação de cada uma das naves.

2.2.1. Reconhecimento

Em naves de Reconhecimento, os tripulantes só podem se teleportar para postos adjacente ao que ele está. Em outras palavras, dado uma componente $c \in G$, o qual é um subgrafo de G, o mesmo será uma nave de Reconhecimento, caso o valor máximo de grau de seus vértices seja 2 para $N - 2$ vértices, sendo N o número total de vértices. E os dois últimos vértices tenham grau igual a 1. A Figura 2 ilustra de maneira melhor a estrutura de uma nave de Reconhecimento.

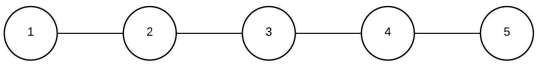


Figura 2. Estrutura de uma nave de Reconhecimento

Assim, dada as características de uma nave de Reconhecimento, para a identificação da mesma, basta apenas olhar o grau de todos os vértices e contando a quantidade de vértices que possuem grau 2 e a quantidade de vértices que possuem grau 1.

2.2.2. Frigatas

Em naves do tipo Frigatas, os postos estão espalhados por toda nave. Essa nave possui como característica principal o fato de possuírem $T - 1$ teleportes possíveis entre os postos, onde T é o número total de postos. Em outras palavras, dado uma componente $c \in G$, a qual é um subgrafo de G, o mesmo será uma nave Frigata, caso o seu número de arestas seja igual ao seu número de vértices subtraído de 1. Ou seja, $N - 1 = M$, sendo N o número de vértices e M o número de arestas. A Figura 3 ilustra a estrutura de uma nave Frigata. Repare como o número de vértices é igual ao número de aresta somado de 1.

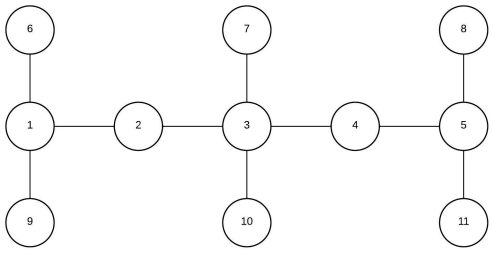


Figura 3. Estrutura de uma nave Frigata

Dada as características da nave, e sabendo que dentro da estrutura componente há o número de vértices e de arestas de cada, para a identificação de naves do tipo Frigata a única necessária a se fazer é comparar o número de vértices subtraído de 1 com o número de arestas. Casos esses valores forem iguais, então a nave é do tipo Frigata.

2.2.3. Bombardeiros

Em naves do tipo Bombardeiros, há duas fileiras as quais podem ter o número de postos de combate diferentes, e sempre, todos os postos de combate da fileira 1, estão conectados com todos os postos de combate da fileira 2, sem que haja ligação entre vértices de mesma fileira. Em outras palavras, dado uma componente $c \in G$, a qual é um subgrafo de G, o mesmo será uma nave do tipo Bombardeiro, se o mesmo for um grafo bipartido. Além disso, para ser uma nave Bombardeira, a mesma precisa que todos os $x \in V$ vértices pertencentes a fileira 1, estejam conectados com todos os $y \in V$ vértices pertencentes a a fileira 2. A Figura 4 ilustra a estrutura do tipo Bombardeiro já mostrando o mesmo como um grafo bipartido.

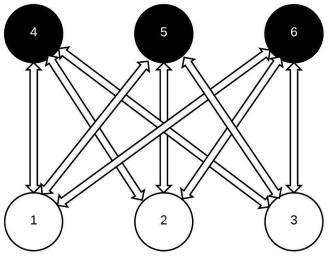


Figura 4. Estrutura de uma nave Bombardeiro

Dado as características da nave, a identificação deste tipo se mostrou a mais custosa, dentre todos as naves dado que temos que avaliar mais o fato do grafo ser bipartido. Para identificação deste tipo de nave foi implementado uma versão do algoritmo de busca em profundidade a qual retorna se um grafo é bipartido ou não 2.

Avaliando o algoritmo 2, é possível ver a implementação do algoritmo de DFS para identificação de grafos bipartidos. O código retorna $False$ quando o grafo não é bipartido, e $True$ caso contrário. O grafo é bipartido, se e somente se, dado um vértice

Algorithm 2 Busca em Profundidade para identificação de Grafos Bipartidos

```
1: procedure DFS_VISIT_BIPARTITE(grafo G, vertice v, int cor)
2:     v → cor ← cor
3:     for each u ∈ v → ADJ do
4:         if u → cor = −1 then
5:             DFS_VISIT_BIPARTITE(G, u, 1-cor)
6:         else
7:             if u → cor == v → cor then
8:                 return False
9:     return True
```

$v \in V$, que tenha uma cor definida como 0 ou 1, tenha todos os seus vizinhos com cor diferente da sua.

Assim, após definirmos que a componente é um subgrafo bipartido é necessária realizar uma última verificação para garantirmos que esse grafo é do tipo bombardeiro. Nessa última verificação, caso o grafo seja bipartido, verifica-se então se a quantidade de vértices da lista de adjacência dos nós brancos, dividido pela quantidade de vértices pretos é igual a quantidade de vértices de cor branca. Em outras palavras, essa lógica define que o número de conexões conexões de um vértice branco deve ser igual ao total de vértices pretos no grafo.

2.2.4. Transportadores

No quarto tipo de nave, as naves de Transporte, os postos de combate estão organizados em duas fileiras as quais sempre possuem o número de postos de combate iguais. Assim, semelhante as naves de reconhecimento, as Transportadoras, só permitem o teletransporte entre postos de combate adjacentes. Em outras palavras, dado uma $c \in G$, o qual é um subgrafo de G, o mesmo será uma nave do tipo Transportadores se todos os vértices tiverem grau igual a 2. Além disso, esta nave se caracteriza por ser um ciclo. A Figura 5 ilustra a estrutura de uma nave do tipo Transportadores. Repare como o mesmo forma um ciclo e como todos os vértices possuem grau igual a 2.

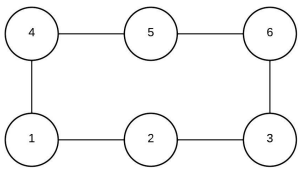


Figura 5. Estrutura de uma nave de Transporte

Dada as características da nave, para identificação da mesma, basta olhar o grau de todos os vértices, caso algum tenha o grau diferente de 2, então, a componente não representa uma nave do tipo Transportadores, caso contrário, representa.

2.3. Cálculo do Tempo de Vantagem

A segunda etapa deste trabalho consiste no cálculo do tempo de vantagem, a qual é definida pela distância mínima que uma frota inimiga está de poder realizar um ataque.

Especificamente, deseja-se encontrar o limite inferior não trivial dessa distância.

A distância mínima de um vértice $\delta(u, v)$, é dada pelo grafo G. Contudo, as distâncias que devem ser calculadas são dadas pelo grafo G', o qual define os teleportes que devem ser realizados por tripulantes de que estão em um vértice $u \in V$ para um vértice $vinV$, dada uma nave $c \in G$. De maneira mais formal temos o tempo de vantagem final é dada pela fórmula 1:

$$min(\sum_{0}^{size(c)} \delta(u, v)) \tag{1}$$

Onde $size(c)$ é a quantidade de componentes $\in G$.

Para realização deste cálculo, foi aplicada neste trabalho a técnica de busca em largura *(Breadth-First Search(BFS))*. A implementação utilizada neste trabalho, assim como a implementação do DFS, é baseada na proposta de [Cormen et al. 2009].

Por meio do BFS é realizado o cálculo da menor distância (δ) entre um vértice $v \in V$, para todos os vértices pertencentes a V. Para este cálculo, poderiam ser utilizadas outras técnicas mais robustas como algoritmos de *Dijkstra*, *Bellman-Ford* e até mesmo abordagens que calculam a distância de todos os vértices para todos. Entretanto, neste problema as arestas não são ponderadas, o cálculo se torna menos custoso utilizando o BFS, além de ser possível a realização de podas, evitando comparações e deixando o algoritmo ainda mais rápido.

Como é necessária a realização, no pior caso, a realização do cálculo de todos os vértices para todos, visando evitar este tipo de cálculo, o qual é muito custoso, foram feitas algumas podas. A poda inicial foi na montagem do G'. Ao realizar a modelagem G', considerou um molde de estrutura gulosa, onde sempre um vértice é adicionado a lista de adjacência do vértice que possui a maior quantidade de vértices em sua lista de adjacência. Desta maneira, ao utilizar esta abordagem consegue-se evitar o recálculo do algoritmo de BFS, tornando o cálculo computacionalmente viável.

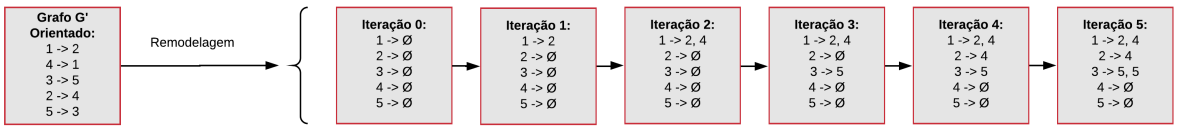


Figura 6. Exemplo da Modelagem com princípio Guloso

A Figura 6 ilustra o funcionamento da técnica proposta, e mostra como ela evita o recálculo do BFS. Repare como os vértices 4 e 5 na última iteração não possuem vértices em sua lista de adjacência, diferentemente de sua modelagem inicial, evitando assim o re-calculo do BFS. Como afirmado os vértices são adicionados sempre no vértice que possui a maior quantidade de vértices em sua lista de adjacência, caso ambos os vértices possuam lista sem valores, então a ligação é armazenada no nodo de menor índice. Vale ressaltar que a modelagem não afeta no tempo de vantagem, dado que a distância é calculado sobre o grafo G, e $\delta(u, v)$ é igual a $\delta(v, u)$.

Além dessa remodelagem feita, foram realizadas duas outras podas que diminuem a quantidade de vezes que o BFS é executado. A primeira poda, assume que se um tempo de vantagem igual a 0 é encontrado, então não é possível definir uma distância menor que essa. Assim, não é necessário continuar calculando a vantagem das outras componentes pertencentes a G.

Por último, foi realizada uma poda sobre o cálculo do tempo de vantagem de uma nave. Dada uma componente $c \in G$, onde c é uma nave, caso o seu tempo de vantagem seja maior em uma iteração i, que o mínimo entre os $\delta(u, v)$ até aquele momento, então não é necessário acabar o cálculo, dado que não possível que ele seja menor que o valor mínimo atual.

3. Análise de Complexidade

Analisa-se nesta seção a complexidade da solução proposta. Essa análise será avaliada por seções, onde avaliando a leitura e modelagem do grafo, a identificação das naves e cálculo do tempo de vantagem.

3.1. Leitura e Modelagem do Grafo

A leitura e a modelagem do grafo são responsáveis pela leitura do arquivo e definição de como o grafo será montado. Essa etapa se torna pouco custosa, pois os dados são armazenados em um *buffer*, o qual possui tamanho $N + M + 1$ linhas, e possui um total de $2 \times (N + M + 1)$ valores, dado que possui dois valores por linha. Para modelagem do grafo G são iteradas os $2 \times M$ primeiros valores do *buffer*, possuindo complexidade $O(2 \times M)$. Já para modelagem do grafo G' são iterados os $2N$ valores restantes do *buffer*, possui complexidade $O(2 \times N)$. Na leitura e modelagem do grafo G' é feita a comparação de qual vértice possui a maior lista de adjacência, tornando a complexidade desta etapa igual a $O(2 \times M + 4 \times N)$, ou simplesmente $O(N + M)$.

3.2. Identificação das Naves

Para descoberta e identificação das naves foram utilizados na implementação, como mostrado na seção 2, o algoritmo DFS em sua versão normal e uma versão para identificação para grafos bipartidos.

Para identificação da quantidade de componentes presentes no grafo G é executado o algoritmo DFS. Como sua implementação segue a proposta de [Cormen et al. 2009], o mesmo, tem complexidade $O(V + E)$. Algumas poucas modificações foram realizadas, entretanto as mesmas não influenciam no desempenho do algoritmo, dado que as mesmas possuem complexidade $O(1)$.

Após realizada a identificação das componentes e de atributos das componentes que auxiliam na identificação do tipo de nave em G , é chamado o método de identificação das naves. O método de identificação percorre todas as componentes de G avaliando os atributos e alocando cada nave para seu tipo específico. O pior caso de custo de identificação de naves possui custo $O(V_{c_i} + E_{c_i})$, onde V_{c_i} e E_{c_i} são os vértices e arestas do í-ésimo componente $c \in G$. Esse pior caso ocorre na identificação das naves Bombardeiros, as quais possuem a necessidade do cálculo do método de busca em profundidade para identificação de grafos bipartidos 2. Vale ressaltar que o número de componentes de G sempre será pelo menos 5 vezes menor que V. Isso ocorre, pois uma nave é composta

por no mínimo 5 vértices, logo, temos que a quantidade de componentes será no máximo $O(\frac{V}{5})$. Por fim, o custo para identificação das naves será no pior caso $O(\frac{V}{5} \times (V_{c_i} + E_{c_i}))$, ou, $O(V_{c_i}^2 + E_{c_i}V)$.

Assim, após a identificação das componentes e das naves, temos que a complexidade é $O(V + E + V_{c_i}^2 + E_{c_i}V))$, que ao fim, se volta a complexidade maior que a de identificação das naves, $(V_{c_i}^2 + E_{c_i}V)$.

3.3. Cálculo Tempo de Vantagem

A etapa do cálculo do tempo de vantagem realiza para cada vértice v em G' o cálculo do caminho mínimo em G por meio do BFS. Assim, no pior caso a complexidade da etapa do cálculo de vantagem é $O(V' \times (V + E))$, sendo V' os vértices de G', e VeE, os vértices de G. No pior caso, cada vértice V' possui apenas um valor em sua lista de adjacência, as distâncias são iguais e maiores ou igual a 1, sendo que desta maneira, nenhuma das abordagens aplicadas iriam auxiliar na redução do número de iterações, sendo a complexidade final no pior caso $O(V' \times (V + E))$.

3.4. Complexidade Final

A complexidade final do método proposto é dada pela complexidade da maior função, a qual realiza o cálculo do tempo de vantagem, tendo complexidade no pior caso $O(V' \times V + E)$. Ressalta-se, entretanto, que nem sempre são avaliados todos os casos devido as podas propostas, o que torna ao final seu tempo de execução em grande parte dos casos menor.

4. Análise Experimental

Nesta subseção é feita uma análise quantitativa dos casos de teste sendo avaliados, além de serem avaliados alguns pontos que auxiliam em um entendimento melhor do comportamento do método proposto. Para as avaliações de tempo foram feitas 10 execuções para cada base de entrada e utilizada a média e o desvio padrão para descrever o comportamento temporal do algoritmo.

Base	# Vértices	# Arestas	# Componentes	Média Tempo Execução	Desvio Padrão	Gasto de Memória (Bytes)
1.in	165	546	10	$0.0068s$	0.0009	160832
2.in	21	20	1793	$0.0073s$	0.0009	95108
3.in	999	992	49	$0.0223s$	0.0049	311556
4.in	2435	10000	3	$0.0239s$	0.0048	1231608
5.in	10000	9524	74	$0.0318s$	0.0061	2355500
6.in	11823	100000	5	$0.2058s$	0.0371	8169224
7.in	42758	100000	1554	$0.2327s$	0.0585	13923428
8.in	22436	100000	58	$0.1596s$	0.0136	9313508
9.in	36407	1000000	3	$1.9075s$	0.0111	68564176
10.in	100000	95626	4	$0.2946s$	0.0116	22473492
pdf1.in	15	14	432	$0.0064s$	0.0011	93624
pdf2.in	19	18	17895	$0.0075s$	0.001	94704

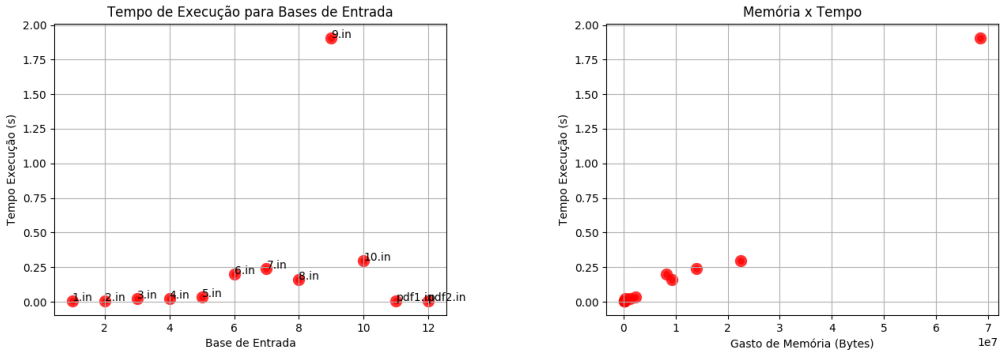
Tabela 1. Dados e tempo de execução de diferentes bases de entrada

Avaliando a Tabela 1, é possível ver dados da execução de diferentes bases de entrada. Por meio da tabela vemos que a entrada $9.in$ é a que o algoritmo tem o pior desempenho, sendo a entrada que fica mais próxima do tempo limite de 2 segundos [1].

[1]Os testes foram realizados em uma máquina pessoal, em computadores do laboratório o valor para entrada 9.in estava próximo a $0.5s$

Os Gráficos 7, mostra o comportamento do algoritmo de diferentes formas em cada base de entrada. O Gráfico 7a, ilustra melhor comportamento do algoritmo para cada uma das bases de entrada. Na Figura 7a fica mais claro que o comportamento para a entrada 9.in é inferior a de outras entradas. Isso se dá pois as podas propostas são possuem o efeito desejado sobre esta base de entrada. Contudo, a base de dados 10.in que é a maior entre todas as bases avaliadas possui um tempo de execução consideravelmente dentro do padrão, dado que é possível realizar a poda, pois seu tempo de vantagem é igual a 0.

O Gráfico 7b, mostra o comportamento do algoritmo com relação ao tempo gasto em uma avaliação e a memória gasta. O maior valor de entrada compreendo a 9.in. Perceba que o tempo de execução do algoritmo é relacionado com o custo de memória.



(a) Comportamento temporal do algoritmo para cada uma das entradas

(b) Relação entre tempo de execução e custo de memória

Figura 7. Avaliações do algoritmo implementado

5. Conclusão

Este trabalho propõe a utilização de um conjunto de algoritmos clássicos em grafos para resolução do problema de *Space Invaders*. O problema consiste na identificação de diferentes tipos de naves que compõe uma frota a qual deseja te atacar. Além de definir quais são os tipos de naves, deve-se definir o tempo mínimo que tem antes que o ataque seja realizado.

Por meio deste trabalho foi possível concluir que a realização de uma modelagem adequada para o grafo permite com que algoritmos simples sejam executados a um custo viável. Por meio das análises foi possível ver que o algoritmo possui um comportamento satisfatório até mesmo para os piores casos, onde a complexidade é $O(V' \times (V + E))$ e respeita a restrição de tempo imposta. Por fim, ressalta-se que o desempenho do algoritmo é satisfatório em grande parte dos casos, sendo a etapa de cálculo de tempo de vantagem a que possui maior complexidade e que dita a complexidade final do algoritmo.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.