UNIVERSIDADE FEDERAL DE MINAS GERAIS
DCC – DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

TRABALHO PRÁTICO 1

Disciplina: Algoritmos 1

Turma: TN

Professora: Olga Nikolaevna Goussevskaia

Aluno: Daniel Souza de Campos

Curso: Ciência da Computação

Agosto/2020

Sumário

1 Introdução

Existe um jogo chamado **O Jogo do Pulo** que se passa em um tabuleiro n-dimensional em que o objetivo dos seus **K** jogadores é, partindo de uma casa/posição nesse tabuleiro, chegar até uma posição final estipulada. No nosso caso, o tabuleiro é bidimensional **M** por **N** e a **posição final** que todos os jogadores querem chegar é a posição **(M,N)**. Por exemplo, em um tabuleiro com 3 linhas e 4 colunas, a posição final será a posição na 3° linha e 4° coluna.

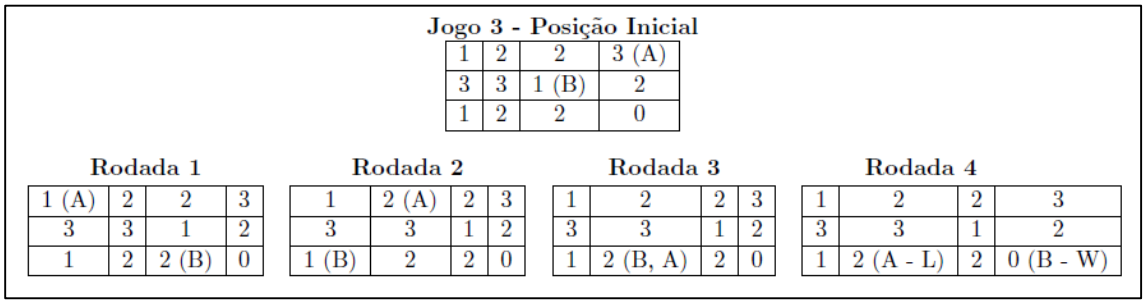
Os jogadores se movimentam apenas nos eixos horizontal e vertical (sem diagonal) de acordo com o valor do tamanho do **pulo P** da posição que estiverem no momento. Por exemplo, caso o jogador A esteja na posição (2,3) com valor de pulo P=1 em um tabuleiro com 3 linhas e 4 colunas, ele só poderia se movimentar para as posições (1,3), (3,3), (2,2) e (2,4). Uma regra importante é que o jogador é obrigado a se mover exatamente P posições, quando possível, nem mais nem menos.

No início do jogo, a ordem que os jogadores escolhem suas casas é a ordem que irão jogar. Na próxima rodada, a ordem de jogada se dará de forma crescente de acordo com o valor de pulo de cada jogador na rodada anterior. Por exemplo, se o jogador A, na primeira rodada, se movimentar primeiro com um pulo Pa=3 e o jogador B, na primeira rodada, se movimentar depois com um pulo Pb=1, na segunda rodada, o jogador B será o primeiro a se movimentar.

Caso dois ou mais jogadores tenham pulado com um valor de P igual na última rodada, a ordem de jogada será de acordo com a ordem em que esses jogadores jogaram na primeira rodada.

Todos os jogadores tentam realizar o caminho ótimo até a posição final.

Desse modo, o **primeiro jogador a conseguir chegar na posição final será declarado o vencedor** e todos os outros perdedores. Caso ninguém consiga chegar na última posição, seja por terem entrado em um loop ou por terem caído em uma posição com valor de pulo P=0, o jogo deverá ser declarado **sem vencedores**.



Jogo 3 - Posição Inicial

1	2	2	3 (A)
3	3	1 (B)	2
1	2	2	0

Rodada 1

1 (A)	2	2	3
3	3	1	2
1	2	2 (B)	0

Rodada 2

1	2 (A)	2	3
3	3	1	2
1 (B)	2	2	0

Rodada 3

1	2	2	3
3	3	1	2
1	2 (B, A)	2	0

Rodada 4

1	2	2	3
3	3	1	2
1	2 (A - L)	2	0 (B - W)

Figura 1: Exemplo de partida

Nesse jogo, existem dois jogadores A e B jogando em um tabuleiro 3 por 4 sendo que, depois de 4 rodadas, o jogador B foi o vencedor.

2 Objetivo

O objetivo do trabalho é desenvolver um programa que, dado todas as entradas necessárias para se iniciar um jogo do **Jogo do Pulo**, calcule qual será, se houver, o jogador vencedor do jogo.

Além disso, a ideia é que, durante o desenvolvimento do programa, seja empregada alguma técnica de pesquisa em grafos para que se ache o melhor caminho entre cada jogador e a posição final.

3 O Programa

3.1 Recursos para o desenvolvimento

O programa foi desenvolvido na linguagem **C++** utilizando o editor de texto **Sublime Text**. Para controle de versionamento e armazenamento do trabalho foi utilizado o GitHub (disponível em: https://github.com/Pendulun/TP1ALG1). Para a verificação de vazamentos de memória foi utilizado o **Valgrind**. Para compilação foi utilizado o **g++** e o **Ubuntu 18.04 LTS** para **Windows 10** para conseguir executar o valgrind.

A máquina utilizada para o desenvolvimento, execução e recuperação do tempo de execução dos testes possui 8Gb de memória RAM e processador Intel Core i5-7200U 2.5 GHz.

3.2 Padrão de entrada e saída

De acordo com a especificação do problema, o programa deve ser capaz de ler um arquivo contendo todas as informações necessárias para se criar uma instância do jogo.

O padrão é o que segue:

Exemplo das linhas da entrada

```
N M // dimensão do tabuleiro;
K // quantidade de jogadores
X11 X12 .. X1M //tabuleiro
X21 X22 .. X2M
.
.
.
XN1 XN2 .. XNM
Ax Ay // posição inicial do jogador A
Bx By
.
.
.
Kx Ky // posição inicial do jogador K
```

Figura 2: Padrão de entrada de informações

E para a saída apenas indicar o nome do jogador vencedor e a rodada em que o jogo terminou.

Exemplo das linhas da saída - Jogo com vencedor

```
X // Jogador vencedor
R // Número da rodada final
```

Exemplo das linhas da saída - Jogo sem vencedores

```
SEM VENCEDORES
```

Figura 3: Padrão de saída do programa

3.3 Divisão de pastas e arquivos

O programa foi dividido em três pastas: **src**, **include** e **build**. Na pasta **src** estão todos os arquivos **.cpp**, na pasta **include** estão todos os arquivos **.hpp** e na pasta **build** estão todos os arquivos **.o**. Além disso, existe o **makefile** para compilação do programa. O arquivo executável, depois de rodar o makefile, será gerado fora dessas três pastas no mesmo nível que o arquivo makefile.

3.4 Implementação do programa

A parte principal do programa é como resolver o problema de achar caminhos para cada jogador, se houver, para a posição final. Para fazer isso, o programa é dividido em 3 passos.

3.4.1 Grafo em lista de adjacência

O primeiro passo é **criar um grafo** para o tabuleiro que reflete as conexões possíveis entre todas as posições de acordo com o pulo P de cada uma dessas posições. Como o intuito é utilizar o algoritmo de pesquisa em largura no grafo para descobrir os caminhos de cada jogador até a posição final, se faz necessário uma **lista de adjacência** para indicar qual posição se liga em outras posições. Isso segue o exemplo dado no livro de ***Kleinberg e Tardos, Algorithm Design*** **página 88** que afirmam que a lista de adjacência ocupa **O(m+n)** memória, **sendo m o número de arestas e n o número de nós ou vértices**.

Uma **lista de adjacência** é um vetor de tamanho n de nós, no nosso caso **N linhas vezes M colunas posições**, de listas de nós/posições. Na implementação “normal” da lista de adjacência, cada **posição u** no vetor se refere ao **nó u+1** da instância do problema e essa posição u contém uma lista de outros nós aos quais o no u+1 tem uma aresta partindo de u e que chega nesses outros nós. Basicamente, **todo nó v tal que a aresta (u+1,v)** existe. Porém, na implementação do programa, **fazemos o contrário**. Na lista de adjacência, para toda aresta (u,v), colocamos o nó u na lista da posição relativa ao nó v. Isso significa que, **ao invés de guardarmos para quais nós um nó consegue ir, guardamos quais nós conseguem chegar a um dado nó**.

Por exemplo, no jogo da Figura 1, a lista de adjacência ficaria assim:

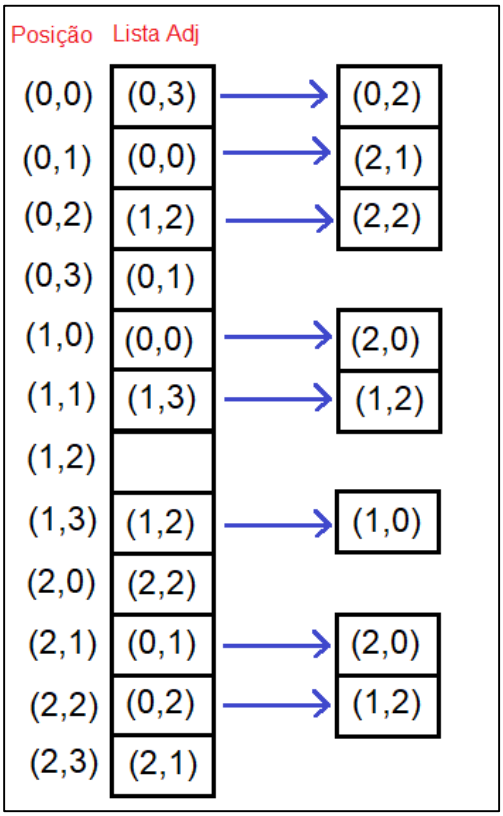


Figura 4: Lista de Adjacência

Podemos perceber, por exemplo, que a posição (1,2) no tabuleiro não é alcançada por ninguém, enquanto a posição final (2,3) só pode ser alcançada pela posição (2,1) e a posição (0,0) é alcançada tanto pela posição (0,3) quanto (0,2).

Por que fazer a lista ao contrário? Bom, dessa maneira, precisaremos fazer **apenas uma execução da pesquisa em largura partindo da posição final** para achar o **componente conectado contendo essa posição final**, ou seja, todos os nós que estão ligados à posição final. Assim, caso uma posição p em que um jogador começa nessa posição esteja no componente conectado da posição final, com certeza esse jogador consegue chegar à posição final. Consequentemente, jogadores cuja posição inicial não esteja nesse componente conectado não conseguem chegar à posição final.

Essa lista pode ser implementada em uma passada por todos os nós existentes, portanto, **O(M*N)**.

3.4.2 Pesquisa em largura (BFS)

O próximo passo é realizar o **algoritmo de pesquisa em largura**. No mesmo livro de **Kleinberg e Tardos, Algorithm Design página 90**, é mostrado um pseudocódigo sobre o algoritmo de pesquisa em largura. Esse algoritmo parte de um nó raiz e descobre todos os nós alcançáveis por esse nó, ou então,

o seu componente conectado. Como já dito antes, estamos fazendo uma versão contrária que irá nos dizer, partindo da posição final/de objetivo dos jogadores, quais jogadores tem um caminho para essa posição final. Desse modo, não precisamos rodar o algoritmo para cada jogador, mas sim, apenas uma vez.

No algoritmo descrito, é utilizado um **vetor de tamanho N*M chamado Descobertos**. Ele serve para guardar quais nós já foram descobertos pelo algoritmo. Além disso, existe também uma **estrutura Árvore** que guarda as arestas formadas por pares de nós analisados e nós que foram descobertos por esse nó analisado. Funciona assim, se, ao analisar os vizinhos um nó v em uma camada x e descobrirmos que dois de seus vizinhos, u e y, não foram descobertos ainda, marcamos u e y como descobertos no vetor Descoberto, os adicionamos na próxima camada de análise e adicionamos as arestas (v,u) e (v,y) na árvore de pesquisa em largura.

Na nossa versão, **unimos a Árvore e o vetor Descobertos em uma única estrutura também de nome Árvore de tamanho N*M**. Essa estrutura é um **vetor de pares** de valores de x e y. **Nela, guardaremos para o nó v, o par x e y do nó u que o achou**. Dessa forma, se a partir de v acharmos os nós u e w, nas posições específicas de u e w guardamos o os valores de xv e yv do nó u no tabuleiro bidimensional. No início do algoritmo, **iniciamos todos os pares de todos os nós com os valores (-1,-1)**. Desse modo, como um tabuleiro não tem posições negativas, fica fácil saber quem já foi descoberto ou não.

Outra estrutura de dados utilizada pelo algoritmo base é uma **lista de listas de nós L** que é responsável por guardar as camadas e seus respectivos nós. Essa estrutura também é implementada no programa desenvolvido e, toda vez que um nó v é descoberto por um nó u que faz parte da camada **L[i]**, adicionamos esse v **a L[i+1]**.

Uma última observação é que, no algoritmo descrito no livro, o BFS recebe um **nó s raiz** mas, no nosso caso, sabemos que o nó raiz deve ser o relativo a última posição do tabuleiro **(N-1,M-1)** (pois iniciamos a contagem no programa a partir do 0 para colunas e linhas).

Um pseudocódigo da pesquisa em largura é o seguinte:

```
BFS():
    Defina um vetor Árvore[N*M] de pares de inteiros e inicialize todos os pares como (-1,-1);
    Defina uma lista L de listas contendo apenas uma lista L[0] com o nó relativo à posição final;
    Defina um contador de camadas i igual a 0;
    Enquanto L[i] não estiver vazio:
        Inicialize uma nova lista L[i+1] vazia;
        Para cada nó u pertencente a L[i]:
            Considere toda aresta (v,u) na lista de adjacência;
            Se o par da Árvore[N*M] relativo a v for igual a (-1,-1):
                Defina o par de v na Árvore como (Xu,Yu);
                Adicione o nó v na lista L[i+1];
            FimSe;
        FimPara;
    Aumente i por 1
    FimEnquanto;
```

Esse algoritmo, como também explicado no livro dito anteriormente na página 91, **executa em tempo O(m+(N*M))** pois serão no máximo n iterações do loop “For” e no máximo 2m, **O(m)**, iterações considerando todas as arestas de (v,u). Logo, O(m) + O(n) = O(m+n).

Para exemplificar a Árvore gerada, vamos pegar de exemplo a Árvore do jogo mostrado na figura 1:

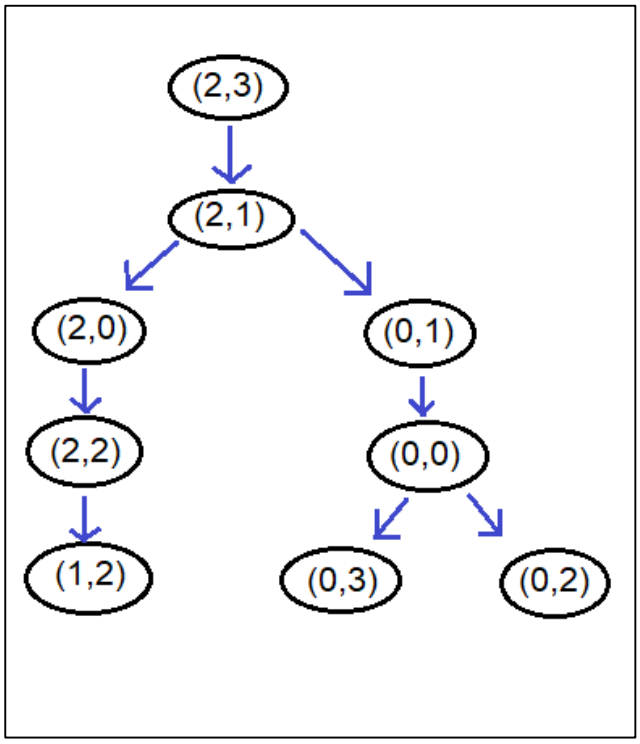


Figura 5: Árvore de pesquisa gerada

3.4.3 Achando o vencedor

Para achar o vencedor, devemos, primeiramente, definir quais dos jogadores são capazes de chegar à posição final.

Para fazer isso, precisamos iterar na estrutura que guardou todos os jogadores, no nosso caso uma lista. Essa iteração tem tamanho O(K), sendo K o número de jogadores.

Analisamos então, para cada jogador, **se o par na estrutura Árvore descrita na seção anterior é diferente da 2-tupla (-1,-1)** – ou como está no código, se a primeira posição é igual a -1. Caso seja igual, significa que a posição de início do jogador não foi encontrada quando realizamos o BFS a partir do nó relativo á posição final do tabuleiro e o jogador não consegue chegar no objetivo. Descartamos esse jogador.

Caso o par mencionado for diferente, isso significa que ele foi achado por algum outro nó no BFS.

Agora é fácil descobrir o caminho que deve ser feito pelo jogador para que ele chegue na posição final. **Se todo nó u que foi achado, foi achado por apenas ou outro nó v, então esse nó u tem apenas um caminho para o nó raiz w**. Logo, basta pegarmos o nó que achou o jogador e o que achou esse e recursivamente até chegarmos no nó raiz. Isso pode ser realizado seguindo os pares salvos na Árvore relativos a todo nó no caminho u-w.

A todo pulo entre nós, salvamos o peso desse pulo que pode ser calculado comparando os valores x e y do nó filho atual e com o nó pai. Do mesmo jeito, vamos contando quantos pulos o jogador deve realizar para chegar no objetivo. **O tamanho do caminho a ser feito pelo jogador é igual ao número da rodada que ele chega nesse nó**.

Quando conseguirmos o tamanho do caminho que um jogador deve realizar para chegar no objetivo**, salvamos o Jogador** com o tamanho do caminho e o peso do penúltimo pulo em uma **lista**. Essa lista **se parece com uma fila de prioridades** pois, a toda inserção, comparamos o tamanho do caminho do jogador a ser inserido com **no máximo K-1** jogadores nessa lista para que eles sejam ordenados, respectivamente: pelo tamanho do caminho, pelo peso do penúltimo pulo e pela ordem na primeira rodada. Esses dois últimos como critério de desempate.

Essa parte do algoritmo irá executar então, para cada Jogador, K iterações, verificar se ele foi achado. Para todo jogador, conseguimos o seu caminho, N*M iterações no máximo. Então, adicionamos esse jogador na nossa lista final de chegada dos jogadores, mais K-1 iterações. Temos portanto: $K*(N*M)*(K-1) = (K^2+K)*(N*M) = O(K^2)*O(N*M) = O(K^2*N*M)$.

3.4.4 Tempo de execução total

Na seção 3.4.1, dissemos que a implementação da lista podia ser feita em O(M*N). Na seção 3.4.2, dissemos que o algoritmo de pesquisa em largura executava em O(m+(M*N)). Na seção 3.4.3, dissemos que, para achar o vencedor, essa parte do programa executava em $O(K^2*N*M)$. Logo, o tempo de execução final será: $O(M*N)+ O(m+(M*N))+ O(K^2*M*N) = O(m+(M*N)*(K^2+1+1)) =$ **$O(m+(M*N)*K^2)$**.

4 Testes realizados

Para realização da medição do tempo, foram feitas 10 contabilizações de tempo de execução do programa para cada um dos cinco testes disponibilizados no moodle. Segue os resultados:

Tabela 1: Tempos de execução para o EX1

Tempo de Execução do Programa para o teste EX1						
Teste	Linhas	Colunas	Jogadores	Vencedor?	N° Execução	Tempo(Microsegundos)
EX1	1	9	3	Sim	1	1016
					2	719
					3	685
					4	487
					5	401
					6	399
					7	419
					8	420
					9	397
					10	396
			Média	533,9	Desvio Padrão	197,865333

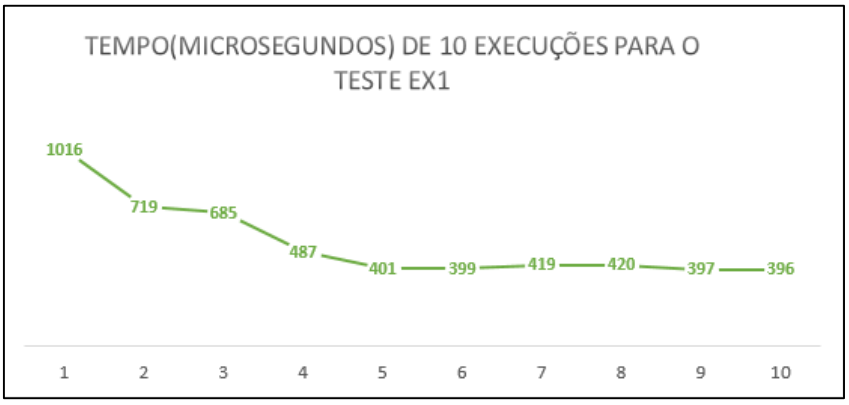


Figura 6: Gráfico do tempo de execução de EX1

Tempo de Execução do Programa para o teste EX2						
Teste	Linhas	Colunas	Jogadores	Vencedor?	N° Execução	Tempo(Microsegundos)
EX2	1	6	3	Não	1	790
					2	564
					3	567
					4	469
					5	495
					6	407
					7	421
					8	411
					9	408
					10	345
			Média	487,7	Desvio Padrão	121,4323268

Tabela 2: Tabela do tempo de execução do teste EX2

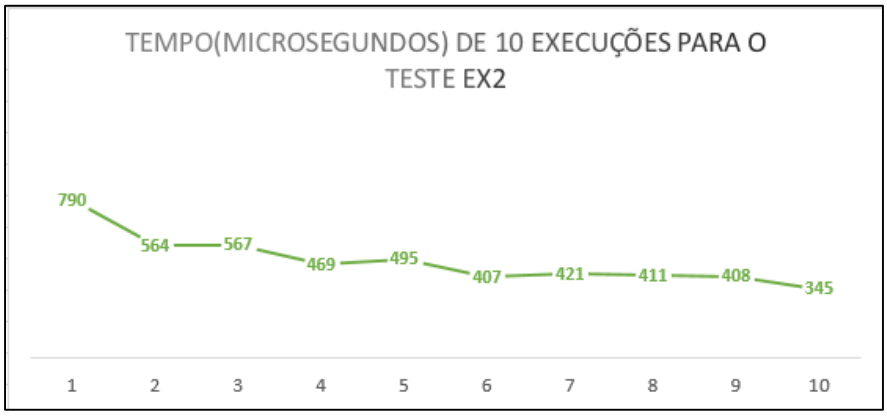


Figura 7: Gráfico do tempo de execução para o teste EX2

Tempo de Execução do Programa para o teste EX3						
Teste	Linhas	Colunas	Jogadores	Vencedor?	N° Execução	Tempo(Microsegundos)
EX3	3	4	2	Sim	1	850
					2	471
					3	441
					4	455
					5	432
					6	438
					7	433
					8	433
					9	431
					10	1177
			Média	556,1	Desvio Padrão	240,3977745

Tabela 3: Tabela para o tempo de execução do teste EX3

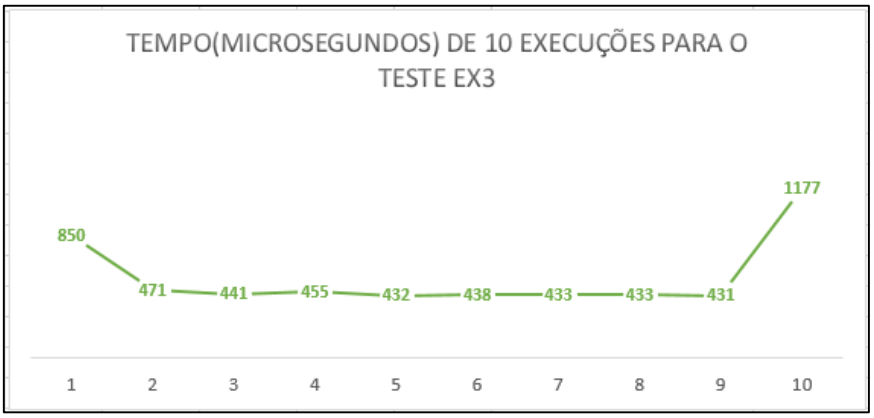


Figura 8: Gráfico para o tempo de execução do teste EX3

Tempo de Execução do Programa para o teste EX4						
Teste	Linhas	Colunas	Jogadores	Vencedor?	N° Execução	Tempo(Microsegundos)
EX4	10	10	5	NÃO	1	870
					2	549
					3	561
					4	552
					5	544
					6	518
					7	497
					8	385
					9	455
					10	936
			Média	586,7	Desvio Padrão	166,7849214

Tabela 4: Tabela do tempo de execução para o teste EX4

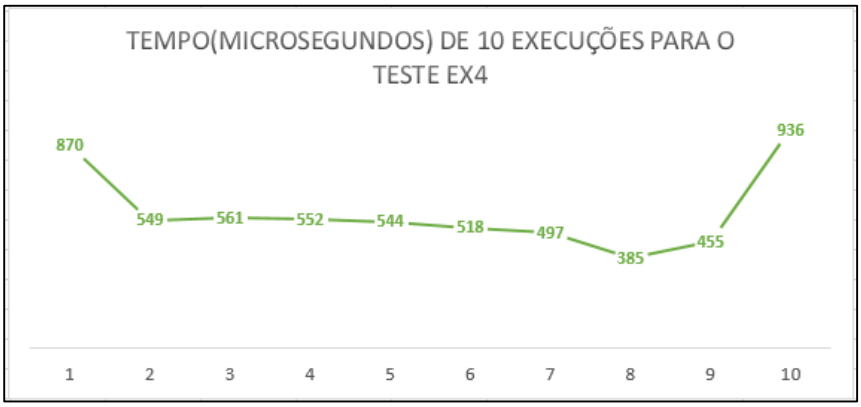


Figura 9: Gráfico para o tempo de execução do teste EX4

Tempo de Execução do Programa para o teste EX5						
Teste	Linhas	Colunas	Jogadores	Vencedor?	N° Execução	Tempo(Microsegundos)
EX5	1000	1000	11	Sim	1	1844403
					2	2394367
					3	1975444
					4	1960167
					5	2002236
					6	1970075
					7	2024289
					8	2065851
					9	1989594
					10	1977719
			Média	2020414,5	Desvio Padrão	135716,4127

Tabela 5: Tabela para o tempo de execução do teste EX5

Figura 10: Gráfico para o tempo de execução do teste EX5

5 Conclusão

Este trabalho tinha o objetivo de empregar os algoritmos vistos em aula para pesquisa de caminhos em grafos em uma situação do mundo real com o chamado **Jogo do Pulo**. Para tal, foi escolhido um algoritmo de pesquisa em largura tendo como base aquele descrito no livro ***Kleinberg e Tardos, Algorithm Design*** mas com alguma alterações propícias à resolução do problema.

Acredito que tal objetivo tenha sido cumprido e que o programa desenvolvido apresentou boa dificuldade de ser realizado. Tais dificuldades apareceram na fase de elaboração de outras formas de implementação dos algoritmos base para que se adequassem melhor ao problema a ser resolvido. Além disso, a programação utilizando a linguagem C++ também foi bem desafiadora, tanto para tentar empregar boas práticas quanto para garantir que o programa não tivesse vazamento de memória.