O Jogo do Pulo - Algoritmos 1

Matheus Aquino Motta[1]

[1]Bacharelado em Matemática Computacional
DCC - Universidade Federal de Minas Gerais

matheusmotta@dcc.ufmg.br

Abstract. *In this documentation we will explain in detail the approach used to solve the Min Jump Game, where we abstract and model the general idea of a board game in a problem of search in graphs, by using basic observations about the objects and well defined relationships that compose our game. Moreover, we give a brief explanation about the time complexity theoretical and experimental analysis of the developed algorithm and show that we're capable of constructing an efficient solution for the proposed problem from small to large inputs.*

Resumo. *Nessa documentação iremos explicar detalhadamente a abordagem utilizada para solução do problema O jogo do pulo, onde abstraimos a ideia de geral de um jogo de tabuleiro para um problema de caminhamento em grafos, a partir de observações básicas acerca dos objetos e relações que compoem o jogo. Além disso, iremos expor a análise de complexidade e experimental do algoritmo proposto, onde seremos capazes de observar a eficiência da solução apresentada para pequenas e grandes entradas.*

1. Introdução

O problema proposto nesse trabalho consiste em encontrar o vencedor para *O Jogo do Pulo*. O objetivo do Jogo do Pulo é bem simples: dado um tabuleiro de dimensões NxM, jogadores e suas respectivas posições iniciais, atingir a última casa do tabuleiro, i.e, a casa de coordenadas $\{N-1, M-1\}$. Cada casa do tabuleiro possui um valor inteiro que informa o tamanho do pulo que o jogador que estiver partindo dela poderá dar, para cima, baixo, esquerda ou direita. O jogo funciona em turnos, logo em cada rodada todos os jogadores jogam exatamente uma vez, em uma ordem determinada pelo tamanho do pulo realizado na rodada anterior, onde aquele que saltou o menor número de casas irá jogar primeiro na rodada seguinte, e em caso de empates a sequência de jogadas é definida lexicograficamente. Assim, dados J jogadores queremos encontrar o jogador vencedor, assumindo que todos os jogadores sempre irão realizar jogadas ótimas.

A abordagem utilizada para solução do problema consiste em duas observações básicas: A simulação do jogo para cada jogador j pode ser modelada como um caminhamento em grafos, e a ordem das jogadas em uma rodada r depende apenas do valor associado a casa em que o jogador estava na rodada $r-1$, para $r > 1$, e de seu valor lexicográfico. Essas ideias são a base para a implementação e corretude do algoritmo, que consiste em J execuções do algoritmo *Breadth First Search*, e uma ordenação do conjunto de jogadores que alcançam o vértice final $k \subseteq J$, para que assim possamos extrair o jogador $j \in k$ que chega em primeiro lugar.

2. Implementação

Assim como a ideia central do algoritmo baseia-se em duas ideias simples, a implementação do algoritmo consiste em uma adaptação do BFS e de uma função que irá definir a ordenação dos jogadores que alcaçam o vértice $\{N-1, M-1\}$. Para cada jogador $j \in J$ o nosso algoritmo irá executar uma simulação do jogo.

Então seja r a rodada em que j alcança a casa final do tabuleiro, o algoritmo irá extrair como resultado uma tupla $\{pulos, peso\}$, onde o primeiro valor será definido pelo número mínimo de pulos necessário para que j chegue a entrada da matriz $\{N-1, M-1\}$, e o segundo valor será definido pelo inteiro p associado a casa do tabuleiro na qual o jogador j estava na rodada $r-1$, i.e, o valor que define a prioridade de jogada de j na rodada r (tratando casos extremos como o qual o jogador já começa no vértice final). De modo que, ao final, extraidos os resultados da simulação para cada j, o jogador vencedor será aquele que alcança o destino no menor número de pulos, e na ocorrência de um empate, aquele que possua o menor peso, uma vez que, aquele que pula o menor número de casas, joga primeiro na rodada seguinte. E na ocorrência de um empate para ambos os casos, o vencedor é definido lexicograficamente pelo nome associado à ele $A-Z$.

2.1. O Grafo Tabuleiro

Agora que sabemos de um modo geral como o algoritmo funciona, iremos discutir as decisões de implementação individualmente em detalhes.

Seja $board$ a matriz definida no input. Iremos definir $G = (V, E)$ como um grafo direcionado, no qual os vértices $V(G)$ possuem um peso p associado definido pela matriz $board$, i.e, cada vértice v_{ij} será reprensentado pela entrada (i, j) do tabuleiro do jogo, logo $|V(G)| = NM$. E suas respectivas adjacências irão ser direcionadas aos vértices u_{ij} os quais v_{ij} alcança ao realizar um pulo de tamanho p.

Portanto, sejam $v_{ij} = (i, j)$ e $u_{lr} = (l, r)$ vértices, para $0 \leq i, l \leq N-1$ e $0 \leq j, r \leq M-1$, temos que v_{ij} é direcionado à u_{lr} se e somente se $(l, r) = (i \pm p, j)$ ou $(l, r) = (i, j \pm p)$.
Assim, definimos nosso conjunto $E(G)$ e podemos verificar que um vértice será direcionado a no máximo 4 vértices, logo $|E(G)| \leq 4|V(G)|$, haja vista que, os vértices $u_{lr} = (l, r)$ são limitados pelas dimensões da matriz $board$. E com isso percebemos que não é necessário definir explicitamente a adjacência de cada vértice, uma vez que, basta que definamos um vetor de movimentos que associa um valor $\pm p$ as coordenadas de v_{ij} e uma função que verique as restrições estabelecidas pelas dimensões da matriz.

2.2. Busca em largura e simulação do jogo

Agora que definimos o nosso grafo, podemos simular $OJogodoPulo$ por meio de um caminhamento em grafos realizado individualmente para cada jogador $j \in J$, com o algoritmo de Busca em Largura (BFS).

O algoritmo essencialmente consiste em um caminhamento no qual, o caminhante começa pelo vértice *source* e explora todos os vértices vizinhos. Então, para cada um desses vértices mais próximos, exploramos os seus vértices vizinhos inexplorados e assim

por diante, até que ele encontre o alvo da busca.
Analogamente, no nosso caso utilizaremos como vértice $source$ o vértice inicial de cada jogador definido pelo input, e iremos definir uma fila de pesquisa, que irá manter os vértices a serem pesquisados, e vetores que irão manter informação de que se o vértice v_{ij} já foi visitado, e a distância do vértice inicial ao vértices v_{ij}, ou seja

$$\begin{aligned} dist[v_{ij}] = d, &\quad \text{Distância mínima de } source \text{ até } v_{ij} \\ visited[v_{ij}] = 1, &\quad \text{Se } v_{ij} \text{ já foi visitado.} \\ visited[v_{ij}] = 0, &\quad \text{Se } v_{ij} \text{ não foi visitado.} \end{aligned}$$

Assim, iremos caminhar no nosso tabuleiro por meio das adjacências definidas pelo vetor de movimento já mencionado, definido explicatemente como

$$move = \{(p_{ij}, 0), (-p_{ij},), (0, p_{ij}), (0, -p_{ij})\}$$

Pesquisando exaustivamente os vértices da adjacência e incluindo-os na fila de pesquisa, marcando os já transpassados como visitados, até que encontremos o vértice $u_{(N-1,M-1)}$ e tenhamos nossa resposta.

A definição do nosso problema exige o caminho ótimo, ou seja, aquele que leve o nosso jogador j com o menor número de pulos até o vértice destino, o que o nosso algoritmo BFS garante. Porém, seja d a distância mínima percorrida por um jogador até o vértice $u_{(N-1,M-1)}$, não é possível garantir que esse caminho irá dar maior prioridade ao jogador j na ocorrência de um empate em relação ao número de pulos. Haja vista que, é possível que exista mais de um caminho com o tamanho/número de pulos igual a d.

Logo, além de determinar a distância percorrida até encontrar um vértice v_{ij}, e marcar cada vértice pesquisado como visitado, iremos armazenar informação a respeito do vértice w_{ij} predecessor a v_{ij}, ou seja, cada vértice descoberto terá um $parent[v_{ij}] = w_{ij}$. Desse modo, quando encontrarmos um vértice não visitado u_{ij} a partir de v_{ij}, adicionamos u_{ij} à fila de pesquisa e definimos seus parâmetros auxiliares da seguinte forma

$$\begin{aligned} dist[u_{ij}] &= dist[v_{ij}] + 1 \\ visited[u_{ij}] &= 1 \\ parent[u_{ij}] &= v_{ij} \end{aligned}$$

Caso u_{ij} já tivesse sido anteriormente visitado, verificamos se

$$dist[v_{ij}] \leq dist[w_{ij}]$$

onde $w_{ij} = parent[u_{ij}]$.

Caso seja menor, atualizamos os parâmetros de u_{ij} em função de v_{ij}, mas não o adicionamos novamente na fila, haja vista que isso já foi feito anteriormente, e nossas mudanças não irão influenciar nesse propósito. Agora, caso tenhamos uma igualdade, verificamos se $p_{w_{ij}} > p_{v_{ij}}$, onde $p_{w_{ij}}$ e $p_{v_{ij}}$ são os valores associados aos vértice w_{ij} e v_{ij}, que definem o tamanho do pulo, se essa condição for verdadeira, significa que o caminho a partir de v_{ij} possui uma distância $d \leq dist[w_{ij}]$ e $p_{v_{ij}} < p_{w_{ij}}$, o que irá prover ao jogador prioridade de jogada na rodada em que ele estiver no vértice u_{ij}.

Assim, de modo a extrair nossa resposta, caso u_{ij} seja igual ao vértice final, isto é, $(i, j) = (N - 1, M - 1)$, adicionamos v_{ij} a um vetor auxiliar e damos continuidade ao algoritmo para o próximo vértice da fila.

Repetimos esse processo até que não existam mais vértices na fila, e ao final teremos um vetor com os vértices que alcançam o vértice $u_{(N-1,M-1)}$ a menos de um pulo. Isso é necessário, devido ao fato de que os parâmetros dos vértices v_{ij} podem ser atualizados no dercorrer do algoritmo, e queremos apenas os parâmetros ótimos para o jogador em questão. Assim, extraímos desse vetor de k elementos aquele que tenha a menor $dist[v_{ij}]$ e em caso de empate aquele que tenha o menor $p_{w_{ij}}$, onde $w_{ij} = parent[v_{ij}]$, retornando assim, para cada jogador, uma tupla $\{pulos, peso\} = \{dist[v_{ij}] + 1, p_{w_{ij}}\}$ de valor mínimo, e caso não exista, retornamos um valor absurdo que será descartado.

2.2.1. Ordenando e encontrando o vencedor

Após a realização do processo acima para todos os jogadores $j \in J$, teremos um vetor de triplas $\{nome, \{pulos, peso\}\}$ de tamanho k, onde $k \leq |J|$ que iremos ordenar, primeiramente, em função do número de *Pulos* realizados por cada jogador (Onde aquele que realizou menos pulos vence.), em segundo lugar, pelo *Peso*, o que define a prioridade das jogadas dada a ocorrência de um empate no número de pulos (Onde aquele que possui o menor peso vence.) e por fim, caso ambos empatem, a ordem é definida em de forma lexicográfica, isto é, pelo *Nome* dos jogadores empatados (Onde aquele que possui o nome lexicograficamente menor vence.). E assim, teremos o vencedor do *Jogo do Pulo* na primeira posição do nosso vetor, e caso o vetor esteja vazio, printamos que não há vencedores.

3. Análise de Complexidade

Agora que entendemos o funcionamento do algoritmo podemos definir sua complexidade da seguinte forma, sejam as variáveis N, M dimensões da matriz, $|V(G)| = NM$, $|E(G)| \leq 4|V(G)|$ tamanho dos conjuntos que definem o grafo, $j = |J|$ o número de jogadores e $k \leq j$ o número de jogadores que alcançam o vértice final $v_{(N-1,M-1)}$, temos que a complexidade do algoritmo é definida explicitamente por

$$\begin{aligned} f(n) &= \mathcal{O}(j(|V(G)| + |E(G)|) + klogk) \\ &= \mathcal{O}(j(NM + 4NM) + klogk) \\ &= \mathcal{O}(j(5NM)) \\ &= \mathcal{O}(jNM) \end{aligned}$$

Esse resultado é devido ao fato de que $k \leq j \leq 11$, logo o fator ligado a k oriundo da ordenação do vetor de triplas de jogadores que alcaçam o vértice final é assintóticamente dominado, e como $j \leq 11$, podemos tratá-lo como uma constante e portanto a complexidade final é dada por $\mathcal{O}(NM)$.

E trivialmente podemos definir a complexidade de memória alocada como $\mathcal{O}(NM)$, haja vista que a memória utilizada foi toda em função das dimensões da matriz de tamanho NxM vezes uma constante, ou de fatores menores relativos ao número de jogadores j, logo são assintóticamente dominados e temos o nosso resultado.

4. Análise Experimental

Agora para verificar as conclusões teóricas acerca da complexidade do algoritmo, testes foram realizados e podemos observar o gráfico do tamnho de tempo em função da entrada N, onde N é a dimensão da matriz NxM, considerando M = N. Onde foram gerados valores aleatórios para compor a matriz e executamos o algoritmo 10 vezes para N variando de 1 até 2000 e extraimos o desvio padrão. (Outros gráficos foram adicionados a pasta de testes do Trabalho Prático.)

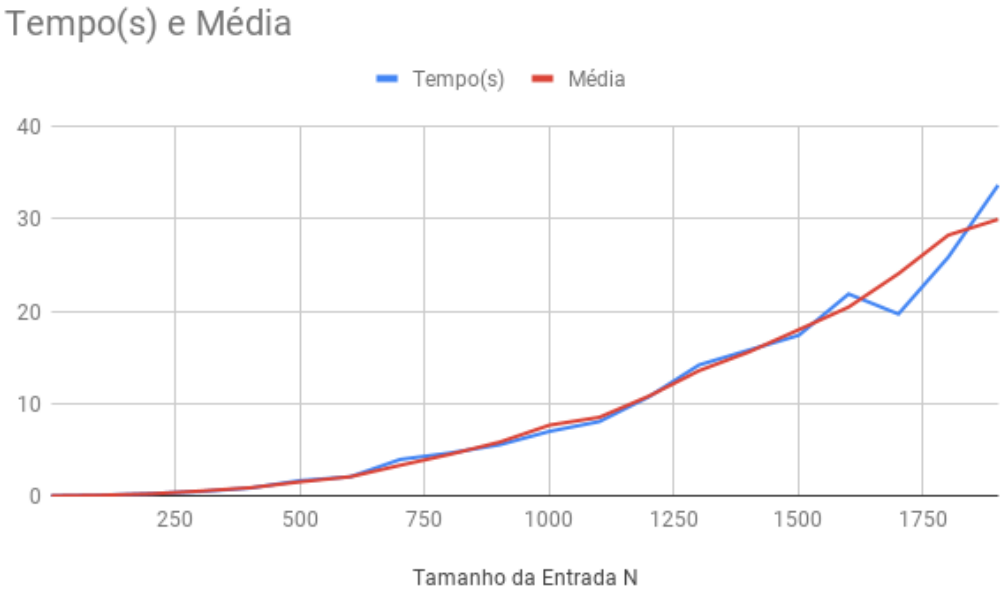

Figure 1. Tempo de execução e média do tempo de execução do algoritmo em função de N, onde N é a dimensão da matriz NxM, considerando M = N.

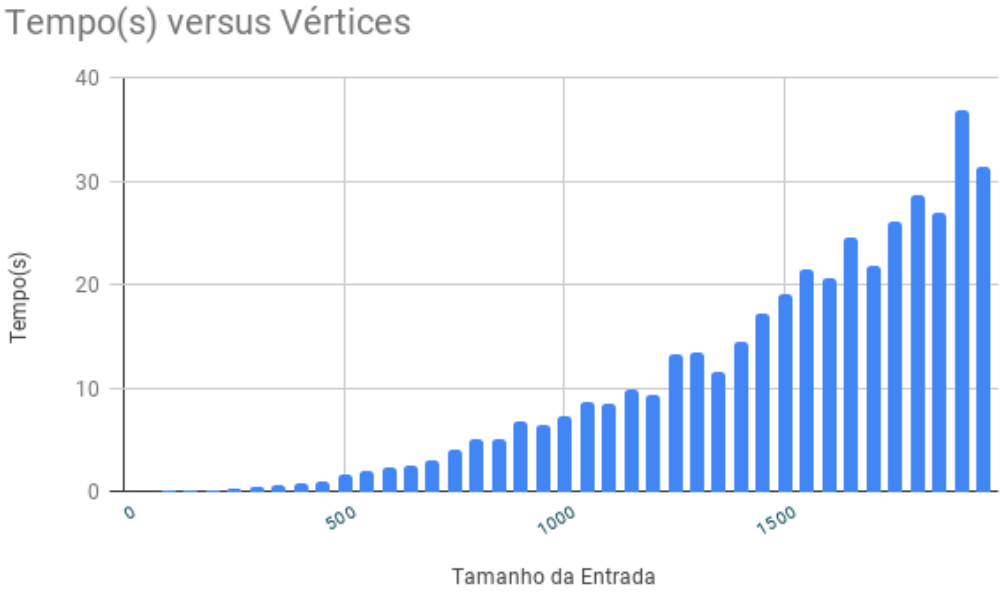

Figure 2. Tempo de execução em função do número de vértices.

Podemos verificar que o Desvio Padrão aumenta quando o valor de N aumenta, mas mais importante, podemos visualizar o comportamento quadrático da função, o que corrobora as nossas assumpções teóricas acerca da complexidade do algoritmo implementado.

Figure 3. Desvio padrão do tempo(s) de execução do algoritmo.

Então, podemos concluir que análise experimental possui resultados assim como o esperado pela análise de complexidade teórica.

Os testes foram realizados em um Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz 8GB RAM.

5. Instruções de compilação e execução

Para a execução do programa basta a utilização do *Makefile* presente no arquivo e a execução do comando *make* no diretório.
O algoritmo foi implementado e testado em um ambiente linux Ubuntu

Distributor ID: Ubuntu
Description: Ubuntu 18.04.2 LTS
Release: 18.04
Codename: bionic

Utilizando o compilador *g++ -std=c++11*.

6. Conclusões

A partir da construção desse trabalho foi possível observar formas de como resolver problemas diversos utilizando modelagem em grafos, abstraindo a ideia central do problema para um conjunto de vértices e arestas a partir de uma relação bem definida entre os objetos que compoem o cenário abordado.