Trabalho Prático 1 - Algoritmos I

Luiz Philippe [1]

[1] Universidade Federal de Minas Gerais – Departamento de Ciência da Computação

luizphilippep@gmail.com

Abstract. *The jump game revolves around a search problem in targeted graphs without weights. From this graph we will have a destination vertex v_f and a set of origin vertices $R = \{r_0, r_1 ... r_{k-1}\}$. The objective of the algorithm is, for each vertex of origin, we want to find the fastest way to the destination efficiently and in a timely manner.*
Palavras-chave: *Algorithms, Graph, Breadth First Search, C ++.*

Resumo. *O jogo do pulo gira em torno de um problema de busca em grafos direcionados sem pesos. A partir desse grafo teremos um vértice de destino v_f e um conjunto de tamanho k de vértices de origem $R = \{r_0, r_1 ... r_{k-1}\}$. O objetivo do algoritmo é, para cada vértice de origem, queremos encontrar o caminho mais rápido ao destino de forma eficiente e em tempo hábil.*
Palavras-chave: *Algoritmos, Grafo, Busca em Largura.*

1. Problema

O problema se desenrola da seguinte forma: dado um tabuleiro, jogadores e suas posi¸ções iniciais, atingir a última casa do tabuleiro (em um tabuleiro de tamanho MxN, essa seria a casa $N-1xM-1$).

O tabuleiro pode conter qualquer número de dimensões, dependendo do modo do jogo e da realidade em que está inserido. Neste trabalho iremos abordar o modo bidimensional do jogo. Cada célula do tabuleiro contém um número que informa quantas casas o jogador deve pular partindo dela. Note que é o número exato de casas, e não o máximo partindo dela. O tabuleiro possui números inteiros positivos, e pode conter casas com o número zero (no caso do número da casa ser um zero, o jogador nunca consegue sair dela).

Os personagens movimentam-se apenas em sentidos paralelos aos limites do tabuleiro. No caso de um tabuleiro bidimensional, por exemplo, o personagem só pode andar nas dire¸ções x ou y, ou seja, movimentos diagonais não são permitidas. Além disso, o movimento pode se dar em qualquer sentido, desde que seguindo a restri¸ção anterior. Novamente, no caso bi-dimensional, o personagem pode se movimentar nas dire¸ções +x, -x, +y ou -y.

A ordem de jogadas e a casa inicial dos jogadores é definida no come¸ço do jogo. No nosso caso, essa ordem e as casas iniciais serão dadas como entrada do problema.

Definidas as posi¸ções iniciais, vamos come¸çar o jogo. Seguindo a ordem posteriormente sorteada, cada jogador pula o número de casas escrito na sua posi¸ção, em qualquer sentido permitido e desejado. A ordem de jogadas nas rodadas seguintes é definida pelo

tamanho do pulo anterior. A regra é que tem maior prioridade aquele que pulou a menor quantidade de casas na rodada anterior e, em caso de empate, tem prioridade o jogador que jogou antes na primeira rodada.

O jogo pode terminar de duas maneiras:

- A vitória de algum jogador
- A impossibilidade de vitória

No primeiro caso, o jogo termina quando o primeiro jogador atinge a última casa do tabuleiro. Já no segundo caso, o jogo acaba quando fica definido que nenhum jogador conseguirá vencer nunca o jogo (jogo sem vencedores).

Um jogo sem vencedores pode ocorrer quando nenhum jogador consegue chegar na ultima posi¸ção do tabuleiro. Isso pode acontecer ao cair em uma casa marcada com o número 0 (e ele não conseguiria sair de lá), ou ao ficar rodando sempre pelas mesmas casas, sem conseguir sair delas.

2. Implementação

A implementação do código foi feita em inglês devido ao benefício de se tratar de um idioma que não utiliza acentos.

Utilizaremos uma matriz $m \times n$ para armazenar os valores de pulo do tabuleiro e um grafo direcionado de $m * n$ nós para representar os possíveis saltos para cada posição do tabuleiro. Nesse grafo, cada nó representa uma célula no tabuleiro, e uma aresta de um vértice v a um vértice w indica que é possível realizar um salto de v para w.

Para a solução do problema foi implementada uma lista de adjacências para a representação do grafo, isso porque para cada posição no tabuleiro, existem apenas, no máximo, quatro saltos possíveis, logo, trata-se de um grafo esparso, e com a representação por lista, economizamos espaço e tempo armazenando somente as arestas existentes do grafo.

Após a construção do grafo a partir da matriz de saltos, selecionamos o nó de origem de cada jogador (os nós pertencentes ao conjunto R mencionado no resumo deste documento) e aplicamos um algoritmo de busca em largura para encontrar o caminho mais rápido deste nó até o vértice destino. Armazenamos em um tipo de dado adequado o caminho percorrido pelo jogador e em seguida avaliamos os caminhos para determinar um vencedor.

Para a determinação do vencedor, temos duas abordagens candidatas. Podemos simplesmente iterar sobre nossos jogadores e selecionar deles, aquele cujo caminho envolve o menor número de saltos (aplicando os critérios de desempate descritos no problema). Ou podemos optar por realizar uma simulação rodada por rodada do jogo até que um jogador seja vencedor. Nesse caso, seguiremos a segunda abordagem pelas suas vantagens quando se trata da visualização da solução.

A seguir, o detalhamento das entidades do algoritmo.

2.1. Tabuleiro

Nossa representação de um tabuleiro contém os seguintes elementos:

- Um "mapa" M, uma matriz $m \times n$ que armazena os valores dos saltos de cada célula do grafo.
- Um grafo $G(V, E)$, uma lista de adjacências representando os possíveis saltos.
- Uma matriz $M^{'}$ $m \times n$ de células já visitadas. Para otimização da solução, inicialmente todas as posições da matriz indicam uma célula não visitada, quando um jogador ocupa uma célula, ela é marcada como visitada. A vantagem que tiramos disso vem de que um jogador que visita uma célula já visitada em uma rodada anterior não tem chances de vitória na partida.

2.2. Jogador

Cada um dos jogadores é representado por um número identificador, sua ordem inicial (geralmente dada pelo indicador), a posição que ocupa e um status "vitória possível" ou "vitória impossível" (a princípio a vitória de qualquer jogador é possível a menos que ele esteja em uma célula de número de salto 0). Utilizamos o algoritmo de busca em largura (ou BFS , breadth first search) para encontrar o melhor caminho da posição ocupada por esse jogador até o nó de destino. Armazenamos esse caminho em uma pilha, ou seja, a célula que será visitada após o primeiro salto se encontra no topo da pilha e o nó de destino se encontra na base. Caso a célula do topo da pilha não seja alcançável pela célula que o jogador ocupa, ou seja, $(v_1, v_2) \notin E$. para v_1 sendo o vértice inicial do jogador e v_2 o vértice do topo da pilha, não existe um caminho para o jogador até a célula de destino, portanto, o jogador é marcado como "vitória impossível".

2.3. Jogo

O jogo é composto por um tabuleiro, um conjunto de jogadores representado po um vetor, um jogador vencedor pertencente ao conjunto e um número de rodadas. O jogo acontece rodada por rodada. Seguindo a ordem determinada, os jogadores movem-se um por vez. Ao fim de cada rodada, a ordem dos jogadores é reavaliada a partir dos critérios especificados e uma nova rodada recomeça. Se um jogador alcança a posição final do tabuleiro, a rodada é imediatamente interrompida e o jogo termina. Se todos os jogadores presentes no jogo estão marcados como "vitória impossível" o jogo termina sem vencedores.

3. Instruções de compilação e execução

O makefile preparado para a execução do código tem as seguintes opções:

- make: Compila os arquivos de código fonte e armazena a saída da compilação em uma pasta "./build/".
- make run: Executa o arquivo gerado pela compilação.
- make mem: Executa o arquivo gerado pela compilação em modo de depuração de memória, assim podemos nos certificar que não existe nenhum vazamento acontecendo durante a execução.
- make test: Executa os script de testes.
- make clean: Limpa a pasta contendo as saídas da compilação.

4. Análise de complexidade

Para analisar a complexidade assintótica da resolução do problema, adotaremos as variáveis m sendo a primeira dimensão do tabuleiro e n a segunda, p o número de jogadores da partida, $v = mn$ o número de nós do grafo relacionado ao tabuleiro e e o número de arestas.

A entrada é lida com uma complexidade de $2mn + p$, já que percorremos toda a matriz duas vezes (preenchendo os valores de pulo e em seguida montando o grafo) e em seguida instanciamos cada um dos jogadores da partida.

Com as entradas configuradas, cada um dos jogadores executa um algoritmo BFS para encontrar seu caminho otimizado até o destino. Sabemos que a complexidade do BFS é dada por $O(v+e)$, e sabemos que cada um dos mn nós tem de 0 a 4 arestas, portanto, nosso valor de e será dado pela ordem de $4mn = O(mn)$, assim podemos afirmar que $O(v+e) = O(mn+mn) = O(mn)$.

Finalmente, rodamos uma simulação das rodadas da seguinte forma: para cada um dos jogadores que podem vencer a partida, até que algum deles chegue ao fim do tabuleiro, realizamos um movimento do seu caminho armazenado e calculamos a ordem de jogadas da próxima rodada com complexidade $O(p * log(p))$. No pior dos casos, um jogador que pode vencer a partida percorrerá todas as casas do tabuleiro até alcançar a última casa, assim, temos que a complexidade dessa etapa será de $O(mnp * logp)$.

Obs.: poderíamos reduzir a complexidade dessa última etapa optando pelo caminho mencionado na seção de implementação desse documento. Assim, como não teríamos que simular cada uma das rodadas, apenas percorrer todo o vetor de jogadores e determinar aquele que alcançou primeiro a última posição, realizaríamos todo o processo com uma complexidade de apenas $O(p)$.

Considerando todas as etapas da solução, temos uma complexidade assintótica descrita por $O(mn+p) + O(mn) + O(mnp * logp) = O(mnp * logp)$. Nas instâncias do problemas com as quais trabalharemos na nossa solução, o valor máximo de p será 11, por isso, trataremos o número de jogadores como uma constante e podemos concluir que a complexidade é de $O(mn)$. Assim, podemos dizer que a complexidade é linear em relação ao número de vértices do grafo, equivalentemente, em relação ao número de células no tabuleiro. O gráfico abaixo ilustra essa noção:

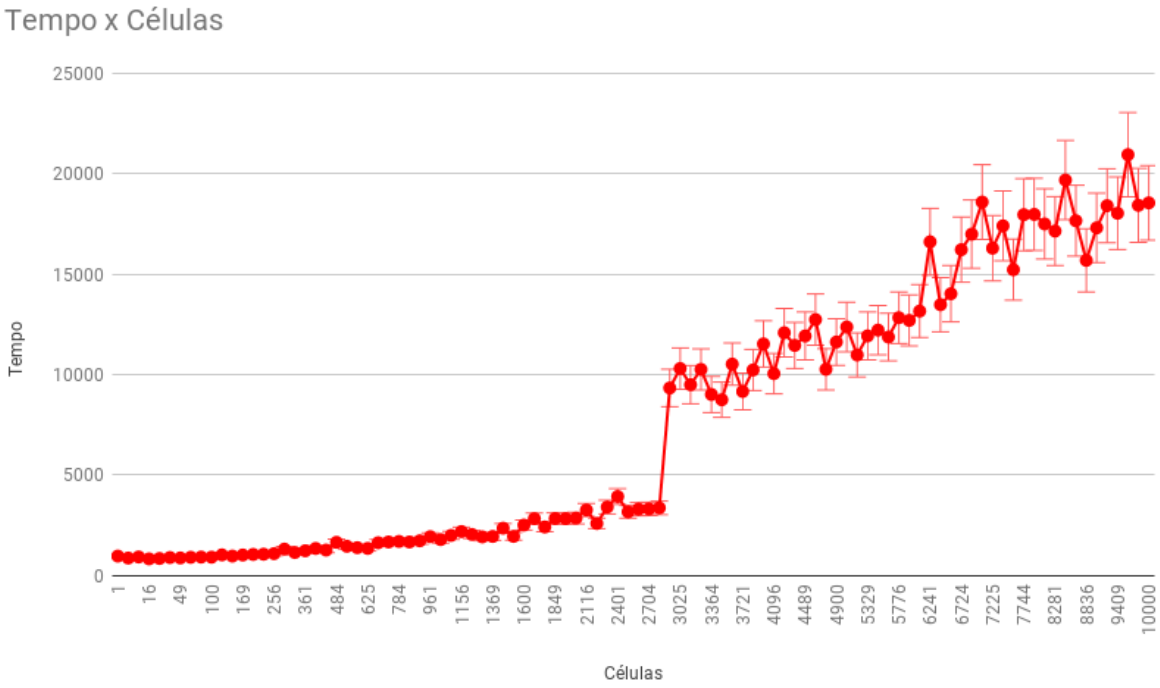

Figura 1. **Gráfico de tempo de execução (microsegundos) em função dos nós do grafo. Para cada valor de números de nós foram feitas 10 medidas, o gráfico apresenta a mediana dessas medidas.**

5. Conclusão

Conclui-se que, em grafos esparsos, onde podemos aproximar o número de arestas a uma constante, o algoritmo de busca em largura torna-se especialmente eficiente. As observações em relação a quantidade máxima de ligações que um vértice do grafo pode ter nos permitiu otimizar nossa solução para uma complexidade linear.