Primeiro trabalho prático de Algoritmos I

Universidade Federal de Minas Gerais

Ábner de Marcos Neves

2018054605

1. Introdução

O jogo do pulo é um jogo de tabuleiro (neste trabalho, bidimensional) que tem como objetivo atingir a última casa do tabuleiro, isto é, a casa *N-1xM-1* num tabuleiro de dimensões *NxM*.

O tabuleiro é tal que em cada casa contém um número *c* que define quantas casas o jogador deve andar a partir dele. Não menos, não mais, mas exatamente *c*. O deslocamento deve ser feito horizontal ou verticalmente em qualquer sentido, ou seja, nas direções +*x*, -*x*, +*y* ou -*y*.

A ordem dos jogadores para a primeira rodada é definida no início e, nas rodadas posteriores, joga primeiro quem 1) tiver andado menos casas na rodada anterior e, se esse critério não for suficiente, 2) quem tiver jogado antes na primeira rodada.

O vencedor é aquele que primeiro chegar na casa *N-1xM-1* do tabuleiro. Se ninguém conseguir chegar nela, o jogo termina sem vencedores.

2. Algoritmo

O jogo do pulo pode ser modelado como um problema de grafos. Para isso, podemos considerar que os vértices são as casas do tabuleiro e uma aresta *(u, v)* existe se é possível ir da casa *u* para a *v*, isto é, se *v* está a uma distância *c* de *u*, sendo *c* o valor presente na casa *u*.

2.1. Estruturas de dados

A princípio, durante a leitura da entrada, o tabuleiro é armazenado em uma matriz com as mesmas dimensões que ele. Cada posição da matriz guarda o valor *c* da posição correspondente no tabuleiro.

Também durante a leitura da entrada, são criados objetos da classe Jogador com os dados lidos (nome e posição inicial). Os jogadores são, então, inseridos numa *std::priority_queue*, que funciona como um *heap*. Essa estrutura de dados foi escolhida por ser compatível com a ideia de que determinados jogadores têm prioridade para a próxima jogada. Para que isso fosse levado em conta, fiz um *comparator* que obedecesse aos critérios da ordem de jogada do jogo. Assim, cada vez que um jogador mudasse de posição, ele seria rearranjado na fila para a próxima jogada de acordo com as regras.

Para a classe Grafo, a implementação assume a forma de uma lista de adjacências, tomando proveito dos *containers* da biblioteca padrão *vector* e *list*. Os índices do *vector* correspondem aos vértices do grafo e cada posição contém uma *list* com os índices dos vizinhos que estão a uma distância *c* daquele vértice.

A fim de manter uma relação consistente entre os índices bidimensionais das posições do tabuleiro com os índices unidimensionais do *vector*, esses últimos são calculados por $M*i + j$, em que M é a quantidade de colunas do tabuleiro e i e j são os índices da posição em questão.

2.2. Procedimentos

A base para rodar o jogo é a busca em largura (BFS) no grafo criado a partir do tabuleiro. Seguindo a ordem dos jogadores, o algoritmo faz para cada jogada uma BFS a partir da posição do jogador.

O critério adotado para escolher para qual das possíveis posições o jogador deve ir é: a primeira posição possível em que ele ainda não esteve. Se há uma posição possível e ela é a posição final do tabuleiro, o algoritmo é terminado e retorna esse jogador como vencedor. Se ela não é o final do tabuleiro, o jogador é movido para ela assim mesmo e ele é elegido para jogar na próxima rodada, pois ainda pode ter chances de ganhar. Se não houver uma posição a que ele possa ir, ele é, então, retirado do jogo e não participará das próximas rodadas.

Se nenhum jogador tiver chegado no final do tabuleiro, a fila de jogadores acabará sendo esvaziada ao decorrer das rodadas e, assim, o jogo acaba sem vencedores.

3. Análise de complexidade

No armazenamento, esse algoritmo utiliza uma matriz NxM, uma fila de prioridade com K jogadores e um grafo implementado como lista de adjacências que conterá M vértices e menos de 4M arestas. Assim, a função de complexidade espacial se forma como

$$E = O(N*M) + O(K) + O(M) = O(N*M + K)$$

No pior caso, o tabuleiro possui o valor 1 em todas as suas posições, permitindo que, de qualquer vértice, se vá para qualquer um dos seus 4 vizinhos. Isso geraria um grafo com quase 4M arestas. Ainda seria menos que 4M devido às posições nos limites do tabuleiro, que têm menos de 4 vizinhos possíveis. Assim, a função de complexidade temporal de uma BFS nesse grafo é

$$C_{BFS} = O(M + 4M) = O(M)$$

No pior caso, cada jogador poderia visitar todas as posições do tabuleiro antes que alguém chegasse no final. E, como é feita uma busca em largura a partir da posição do

jogador antes de cada jogada, a BFS seria invocada N*M vezes, fazendo com que o algoritmo tenha uma função de complexidade em tempo total de

$$C_{Total} = N*M * O(M) = O(N*M^2)$$

4. Conclusão

Percebo que meu algoritmo não é o esperado. A escolha da próxima posição de cada jogador leva a um caminho diferente na maioria dos casos que testei. Isso se deve à minha interpretação errônea da suposição expressa no enunciado de que "*os jogadores adotarão sempre a melhor estratégia*". Fez-me pensar que *qualquer estratégia* seria considerada a melhor e adotei a de ir para a primeira casa possível ainda não visitada.

Assim, desenvolvi meu trabalho processando rodada por rodada em vez de implementar a estratégia de fruição da busca em largura que era esperada.

Apesar disso, foi um trabalho útil para compreender melhor como funciona a busca em largura num grafo. Só parece que não tive muita visão do seu potencial para resolver esse problema.