Trabalho Prático 1
Algoritmos 1

Daniel Ferreira Abadi[1] **- 2018088062**

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

1. Introdução

O problema abordado consiste em simular o Jogo do Pulo, um jogo de tabuleiro. O mesmo tem um objetivo bem simples: dados os jogadores e suas posições iniciais, quem chegar na última posição primeiro ganha. Cada casa do tabuleiro tem um número natural que define quantas casas o jogador tem que pular, sendo obrigado a pular o número integral. Caso algum jogador caia em uma casa com o número 0, o mesmo perde o jogo.

No começo do jogo os jogadores são sorteados e suas posições iniciais são determinadas. A ordem na primeira rodada é a ordem do sorteio, e nas próximas rodadas os jogadores que pularam menos casas tem a prioridade na próxima rodada.

2. Implementação

Para representar o tabuleiro e suas casas foi utilizado o conceito de grafo direcionado. Cada jogador é representado como uma estrutura de dados, e cada vértice do grafo também é uma estrutura de dados. Escolhi esse tipo de implementação pois é o que acho mais organizado, apesar de que há maneiras mais eficientes de representar tais aspectos do problema.

2.1. Jogador

Cada jogador tem um vértice e um próprio grafo, pois cada um começa de um ponto diferente do tabuleiro. Há também duas variáveis que auxiliam na decisão de quem ganha o jogo. E seu único método é seu inicializador, que cria o grafo e atribui todas as variáveis.

2.2. Vértice

Essa estrutura representa as casas do tabuleiro. Ela tem um valor, que representa o número da casa, um id, que ajuda na sua identificação dentro do programa, e um Vector de outros vértices, que são as possíveis casas que se consegue chegar dado o valor.

A estrutura possui 1 método: cria_vizinhos, que recebe o endereço de memória de um vértice e o coloca no Vector.

2.3. Grafo

A estrutura Grafo representa o caminho que o Jogador percorre pelo tabuleiro. Nesse caso a estrutura não tem realmente um grafo, mas foi a melhor forma que encontrei para nomeá-la. Essa estrutura possui 3 métodos, sendo o inicializador que recebe apenas um inteiro, uma função chamada desempata, que será explicada logo abaixo, e a função principal de todo o programa, O BFS, que recebe o endereço de memória de um Vértice e o id do Vértice final.

Uma variável chamada jogadas é inicializada com o valor -1 pois caso o BFS não atinja a última casa do tabuleiro, esse valor não mudará, assim auxiliando na decisão de quem vence o jogo. Essa variável é posteriormente passada para o jogador.

2.3.1. BFS

O algoritmo BFS implementado foi baseado na versão do livro Algorithm Design, o livro texto da disciplina. Nele uso uma fila de Pair contendo o endereço de memória de um Vértice e um inteiro que representa o valor do Vértice ”pai”, que auxilia no desempate do jogo. Há também 2 vetores de inteiros, de tamanho n*m, que servem para guardar a camada em que cada Vértice foi encontrado e para marcar se o Vértice já foi encontrado.

Para cada Vértice que chega ao destino final é guardado em um vector de Pair's a camada atual do Vértice e o valor do ”pai”do mesmo.

2.3.2. Desempate

Durante a produção do código percebi que há um ”corner case”muito específico que diz respeito a melhor escolha possível do Vértice a ser pulado. Por exemplo, seja N o pulo que dou para a última posição do tabuleiro, o BFS não me garante que no pulo N-2 eu vá para a casa com menor valor que também chegue ao Vértice final. Ele garante apenas o menor caminho, mas não o melhor caminho com aquele número de pulos baseado no valor da casa.

Para resolver o problema, criei o método desempate que utiliza o vector de Pair's citado na subseção acima. Ele simplesmente escolhe o Pair que tem o menor número representando a camada. Se tiverem valores iguais, o mesmo seleciona o que tem o menor número que representa o valor do Vértice pai, visto que os jogadores que deram o menor pulo têm a preferência.

3. Instruções de compilação e execução

Vá até a pasta Daniel_Ferreira_2018088062, pelo terminal, e digite “make” para compilar. Para executar o programa digite “./tp1”. Há também a opção “make test” que executa o programa utilizando os casos teste dados.

4. Análise de complexidade de tempo e espaço

Neste tópico irei focar as análises nas partes mais relevantes do programa e que tenham maior impacto em sua complexidade final.

4.1. Main

No início do programa temos a inicialização de uma matriz m*n que guarda os Vértices e a inicialização de um vetor que guarda o endereço de memória do tipo Jogador.

Para a matriz de Vértices temos 3 loops, que servem para alocar os valores, preencher o Vector de vizinhos de cada Vértice e para desalocar os mesmos. Já para o vetor de Jogadores temos algo muito semelhante, 3 loops que servem ara inicializar os mesmos, verificar quem foi o ganhador e para desalocá-los. Apenas o main já tem complexidade O(m*n), sendo m*n a quantidade de Vértices.

4.2. Vértice

Como dito antes, a estrutura possui um Vector que guarda os seus vizinhos, que recebe o endereço de memória que aponta para o tipo Vértice. Então, no pior caso todos os Vértices tem uma aresta entre si, logo o Vector teria uma complexidade espacial de O(m*n) e, a função que adiciona elementos no Vector é de complexidade temporal constante O(1), não adicionei aqui o loop necessário para percorrer a matriz de Vértices pois o mesmo está incluso na subseção acima.

4.3. Grafo

A estrutura em si não possui nada relevante em termos de complexidade espacial. O importante aqui é o algoritmo BFS que possui dois vetores do tipo inteiro de tamanho m*n, logo a complexidade espacial é de O(m*n). Há também uma fila de Pair's que contém o endereço de memória do tipo Vértice e um inteiro. No pior caso para a fila temos uma complexidade espacial de O(m*n) que aconteceria caso o primeiro Vértice tivesse uma aresta para todos os outros.

Já em termos de complexidade temporal, o BFS tem complexidade O(m*n + A), sendo m*n o número de Vértices e A o número de arestas do grafo. Isso pois o algoritmo percorre todos os Vértices uma única vez e, para cada aresta, ele verifica se aquele Vértice foi encontrado. Em caso negativo, o Vértice vizinho é adicionado na fila e tem seu correspondente no vetor alterado para 1. Essas duas últimas operações tem complexidade constante O(1).

5. Análise experimental

O gráfico abaixo mostra a relação entre tempo e tamanho do tabuleiro, com o número de jogadores fixo em 11. Decidi que o melhor seria manter o número de linhas e colunas iguais, para representar um "pior"caso. O valor de cada casa e o ponto inicial de cada jogador é sempre aleatório.

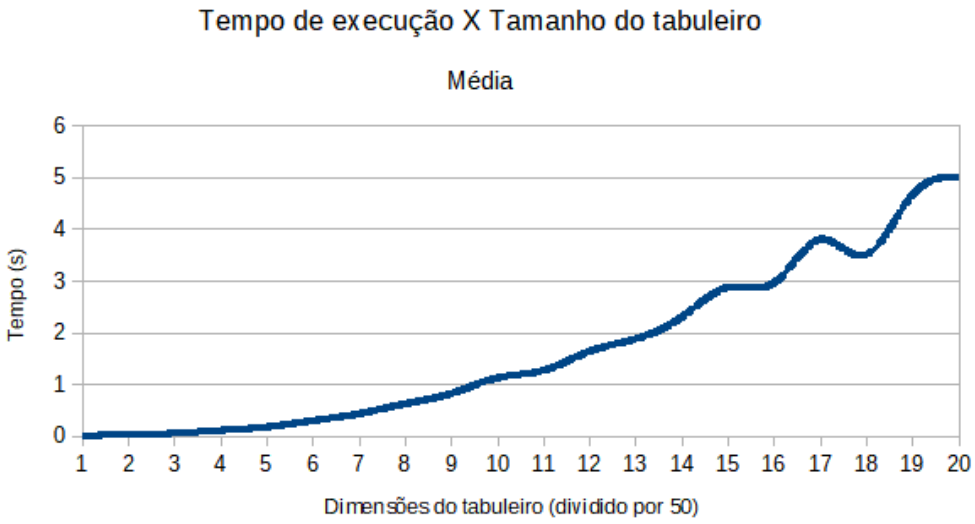

Figura 1. Gráfico

Para cada tamanho de tabuleiro foram feitas 10 amostras e a média foi calculada e plotada no gráfico acima. O gráfico mostra que o tempo de execução parece crescer em um ritmo linear, com nenhuma mudança brusca a partir de um certo valor. O que sugere que o programa realmente segue os valores descritos em sua análise de complexidade temporal. Para jogos sem vencedores o programa executa no mesmo período de tempo, pois o mesmo apenas não atinge a última casa do tabuleiro.

O gráfico abaixo mostra o desvio padrão das amostras geradas para todos os tamanhos de tabuleiro.

Figura 2. Gráfico

6. Conclusão

Acredito que o problema foi resolvido de uma forma bem elegante e que o propósito do trabalho foi cumprido, que era aprender melhor sobre o BFS. Acredito que a complexidade temporal total é definida pelo BFS, que é O(m*n + A), sendo m e n o número de Vértices e A o número de arestas, e a complexidade de espaço seja O(m*n).

Ao que me parece, a complexidade temporal teórica é bem mostrada olhando-se os valores da análise experimental. Não parece ter um crescimento quadrático. O BFS se mostra eficaz dando o tamanho das entradas e o tempo de execução, dado que 1 milhão de Vértices a execução não passa de 5.3 segundos em média.

Uma observação sobre o trabalho é que quando se roda, com valores acima de 700, utilizando o Valgrind o mesmo aponta um ”warning”de troca de stacks, mas acredito que isso não afeta o programa Para rodar o Valgrind normalmente, sem que apareçam esses warnings, basta usar a flag --max-stackframe=8000032”. Os ”frees”de ponteiros ocorrem normalmente, sem nenhum leak.

A única referência utilizada foi o livro texto da matéria [Kleinberg and Éva Tardos 2006].

Referências

Kleinberg, J. and Éva Tardos (2006). *Algorithm Design*. Pearson/Addison-Wesley, 1th edition.