Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Algoritmos 1

Trabalho Prático 1
Jogo do Pulo

Aluno: Lucas Augusto Araújo Aguiar
Professora: Olga Nikolaevna Goussevskaia

Belo Horizonte, 01 de Setembro de 2020

1 - Introdução

Neste trabalho, recebi a função de realizar ‘previsões’ para o jogo do pulo. O jogo do pulo consiste num jogo em que, em um tabuleiro MxN, cada quadrado tem um valor que diz ao jogador quantas casas ele pode movimentar, sendo esse movimento em uma única direção e sentido. Dessa forma, dado os jogadores e suas posições iniciais, eu tive que prever em quantos passos um jogador ganharia, sendo a condição de vitória aquela em que o jogador consegue chegar na última posição do tabuleiro, caso MxN, a posição (M-1)x(N-1). Por exemplo, um tabuleiro 1x3 em que a posição (0, 0)=1, (0, 1)=1 e (0,2) o ponto de vitória. Caso o usuário comece no ponto (0, 0), ele terminará ganhando em duas rodadas, visto que poderá ir para (0, 1) e, logo em seguida, para (0, 2).

2 - Implementação

Classes, structs e métodos utilizados:

a) Board: classe responsável por montar o tabuleiro. De modo geral, utilizada apenas para manuseamento de posicionamento e valor por posição. Tal classe possui como atributos: a dimensão x do tabuleiro (x), a dimensão y do tabuleiro(y) e um vetor de vetores (board), representando uma matriz e, assim, o tabuleiro.

   Nessa classe, temos os seguintes métodos:

   - Board(x, y): representa o construtor, onde passo os parâmetros x e y e, assim, declaro a matriz dando os ‘resizes’ necessários.
   - setX(x): função responsável por ditar o valor da variável x.
   - getX(): função responsável por retornar o valor da variável x.
   - setY(y): função responsável por ditar o valor da variável y.
   - getY(): função responsável por retornar o valor da variável y.

- setPositionValue(x, y, value): função responsável por ditar o valor da posição (x, y) na matriz.
- getPositionValue(x, y): função responsável por retornar o valor da posição (x, y) na matriz.

b) Player: classe responsável por representar os jogadores. Tal classe possui os atributos: o identificador (id), um par de inteiros que representa sua posição (position) e um vetor que a quantidade de passos que ele deu em cada jogada possível (steps).

Nessa classe, temos os seguintes métodos:

- setX(x): função responsável por ditar o valor da sua posição em relação ao eixo x.
- getX(): função responsável por retornar o valor da sua posição em relação ao eixo x.
- setY(y): função responsável por ditar o valor da sua posição em relação ao eixo y.
- getY(): função responsável por retornar o valor da sua posição em relação ao eixo y.
- setId(id): função responsável por ditar o valor do id do player.
- getId(): função responsável por retornar o valor do id do player.
- setSteps(steps): função responsável por adicionar um valor a um novo passo do player.
- getLastStep(): função responsável por retornar o valor do último passo do jogador.
- getFirstMoviment(): função responsável por retornar o valor do primeiro passo do jogador.

c) Game: classe responsável por manusear as outras classes e a regra do jogo, assim como toda lógica. Tal classe possui como atributos: o tabuleiro (board),

um vetor de jogadores (players) e um vetor de vetor de pares (list), o qual representa nossa lista de vértices adjacentes.

Nessa classe, temos os seguintes métodos:

- Game(board): construtor da classe, a qual já inicializa com o board criado na main.
- addPlayer(player): função responsável por adicionar um player no vetor de players.
- getPlayer(id): função responsável por encontrar um player pelo id e retorná-lo.
- createListAdj(): função responsável por criar a lista de adjacência do jogos, a qual é feita da seguinte forma: eu determino que o vetor terá MxN vetores dentro dele, de modo que o primeiro elemento de cada um desses vetores será uma posição referente ao tabuleiro. Dessa forma, o primeiro elemento do primeiro vetor da lista de vetores é o (0, 0), o primeiro elemento do segundo vetor da lista de vetores é o (0, 1) e assim sucessivamente. A partir desse ponto, a função passa por cada um dessas posições iniciais e analisa quais as possíveis posições essa posição inicial pode chegar, inserindo-as nesse vetor. Por exemplo.: supondo que a posição (0, 0) tenha o valor 1 e o tabuleiro seja 1x3. Dessa forma, o vetor[0] será [(0,0), (0,1)], e assim sucessivamente para cada vetor da lista de vetores, uma vez que o primeiro elemento funciona como um vértice de um grafo e o restante dos elementos são ligações desse principal.
- bfs(): função responsável por realizar a busca de níveis na minha lista de adjacências, a qual funciona da seguinte forma: eu possuo uma fila (q), um map onde sua chave é um par que representa uma posição no tabuleiro e seu valor é a distância dessa posição da posição de partida e se ela foi visitada pela minha busca ou não(visited) e um map com os possíveis vencedores do game onde a chave é o id do jogador e o valor é a quantidade de rodadas que esse jogador levou para chegar na posição

vencedora(possiblesWin). Dessa forma, eu realizo um loop para cada jogador existente e inicializo a queue com a posição inicial desse jogador. A partir disso, entro em um outro loop em que, enquanto a fila não estiver vazia, ele não irá parar e, dentro desse outro loop, eu passo procuro em qual vetor do vetor de vetores seu primeiro elemento corresponde à posição atual da fila. Ao encontrar, marco como 'visitada' essa posição e coloco na fila todas as posições que estão nesse mesmo vetor, verificando se elas já foram visitadas, uma vez que, caso tenham sido, eu não insiro na fila. Nesse tempo, ao marcar que a posição foi visitada, eu guardo também em quantas rodadas ela foi visitada, pegando as rodadas da posição anterior a ela e adicionando 1. Tal função retorna o map possiblesWin.

- defineWinner(possiblesWin): função responsável pelo output do programa, definindo o vencedor, caso haja, e o número de rodadas gastas para realizar isso, além de verificar, caso haja empate de rodadas, qual deu o menor penúltimo passo ou qual deu o menor primeiro passo. Ela é realizada com dois loops, passando pelos possíveis vencedores retornado pelo bfs() e pelos players do game em questão. Dessa forma, ele verifica, para cada possível vencedor, qual levou a menor quantidade de rodadas para conseguir vencer e todas as outras verificações supracitadas.

d) Main: na main, apenas fiz tratamento de entrada e chamei as funções do game.

e) Escolha de tipos de variáveis e estruturas: Por tratar bastante de pares de posições (x, y), utilizei pair ao longo do código, sempre que precisava marcar uma posição ou quando queria que duas informações quaisquer fossem guardadas em conjunto. Utilizei vector para a lista de adjacências por conhecer a quantidade exata de outros vectors necessários. Utilizei queue para fazer a fila devido ao manuseamento facilitado para essa função e um map para marcar os possíveis vencedores pois cada jogador precisa de um número de rodadas.

3 - Instruções de compilação e execução

Para compilar o programa, basta acessar a pasta pelo terminal e digitar ‘make’. Após isso, para executar o programa, basta acessar a pasta build, ainda pelo terminal, e digitar ‘./tp1’. Ademais, para verificar os testes presentes na pasta tests basta digitar ‘make test’(caso adicione testes, remodele o for do arquivo .sh) e para limpar os arquivos temporários criados basta digitar ‘make clean’.

4 - Análise de complexidade

Análise assintótica:

- Board() apresenta complexidade O(M*N) onde M e N são as dimensões do tabuleiro, uma vez que cria cada posição do tabuleiro. Dessa forma, a complexidade é O(n^2).
- Game() possui complexidade O(M*N) onde M e N são as dimensões do tabuleiro, uma vez que percorre cada posição do tabuleiro. Dessa forma, a complexidade é O(n^2).
- createListAdj() possui complexidade O(M*N) onde M e N são as dimensões do tabuleiro, uma vez que percorre cada posição do tabuleiro e insere em cada vetor de vetores, sendo tal ação O(1), onde o jogador começa na casa vencedora ou começa na posição em que não pode se mover. Dessa forma, a complexidade é O(n^2) para o pior caso, onde todo vértice pode ser acessado.
- bfs() possui complexidade O(P*(M*N+E)), onda P é o número de players, M a dimensão x do tabuleiro, N a dimensão y do tabuleiro e E o número de ligações, sendo algo entre 0 e 4. Dessa forma, a complexidade é O(n^3).
- defineWinner() possui complexidade O(P*W), onde P é o número de jogadores e W o número de jogadores que chegam à posição final. Nesse caso, o pior caso

é O(n^2), onde todos os jogadores chegam na posição final e o caso médio é O(nlogn).

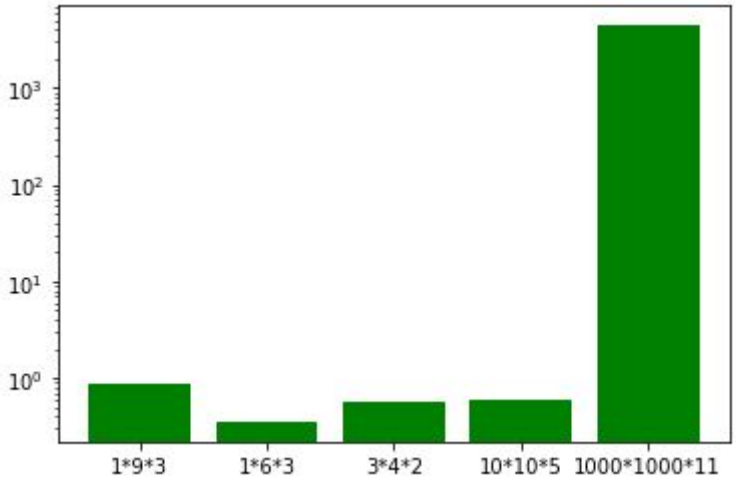
Análise de custo de memória:

Analisaremos o custo de memória por variáveis e estruturas relevantes.

- board: N*M*sizeOf(int), onde M e N são as dimensões x e y do tabuleiro.
- game->list: M*N*sizeOf(std::pair<int, int>), onde M e N são as dimensões x e y do tabuleiro.
- game->visited: M*N*sizeOf(std::pair<int, int>), onde M e N são as dimensões x e y do tabuleiro.

Análise experimental:

Analisei o tempo por meio da biblioteca ‘chrono’, apenas pegando o ‘now’ no começo e no fim do programa e comparando ambos. Para plotar, utilizei python.



O y é o tempo em log, visto que o plot normal seria de difícil visualização devido ao tamanho do y para entradas grandes. A partir do gráfico, podemos verificar que o tempo do algoritmo cresce consideravelmente de acordo com sua entrada.

```
Média 1*9*3:
0.81
Desvio padrão:
0.10
Média 1*6*3:
0.38
Desvio padrão:
0.11
Média 3*4*2:
0.84
Desvio padrão:
0.19
Média 10*10*5:
0.61
Desvio padrão:
0.19
Média 1000*1000*11:
4369.00
Desvio padrão:
103.29
```

5 - Conclusão

O trabalho foi muito produtivo em questão de aprendizado, uma vez que pude utilizar ponteiro único, coisa que nunca havia feito antes, mesmo que tenha tido pouca influência no projeto como um todo, foi interessante de ser utilizado. Além disso, pude aprender melhor sobre o bfs. A criação da lista de adjacência foi simples, uma vez que não possui uma complexidade mais elevada. Ademais, o bfs também foi fácil de ser implementado, uma vez que seu código também é simples. No entanto, o nível de complexidade do trabalho pra mim foi determinar em quantas rodadas o jogador conseguiu chegar na posição final, uma vez que como passava por todo nível do grafo, inicialmente, a minha quantidade de rodadas sempre dava superior à resposta desejada. De resto, o trabalho foi interessante de ser implementado e eu gostei bastante.

6 - Bibliografia

https://pt.wikipedia.org/wiki/Busca_em_largura,
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/,
https://en.cppreference.com/,
Aulas da Olga.