Trabalho Prático 1 - Algoritmos 1

Breno C. Pimenta
RA: 2017114809

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Brasil

brenocpimenta@gmail.com

1. Introdução

O problema pode ser modularizado em duas partes. A primeira parte consiste do mapeamento de um grafo direcional para uma estrutura de dados.

A segunda parte consiste de um algoritmos capaz de encontrar o menor caminho entre os dois nós desse Grafo. Porém esse algoritmo deve ser customizado para que consiga atender também a necessidade de turnos do jogo, pois alpem do menor caminho deve ser atendido também uma ordem regularizada a partir de valores de cada nó da estrutura.

2. Implementação

Para a construção da solução do problema foi seguido a modularização, como detalhado a cima. Primeiramente, houve a construção da estutura do grafo composta pelas classes **Graph** e **NodeGraph** que se encontram na pasta *src/graph,* sendo este módulo completamente independente dos restantes. O TAD **NodeGraph** corresponde a um nó do grafo, porém, além de ser um nó com o valor atribuído e ponteiros para os nós subsequentes a ele, também realiza uma interface com o que seria o tabuleiro imaginário do jogo, faz isso através do armazenamento da sua posição no jogo e se é ou não a posição final do mesmo. O TAD **Graph** é responsável por inicializar e gerenciar os nós. A inicialização se dá através da linearização da representação do tabuleiro (entrada) e a partir dela cria cada nó verificando seus valores, posições e filhos (possíveis movimentos no tabuleiro). Como já dito sobre o nó, o grafo realiza a interface com o tabuleiro através de um método que nos permite acessar um nó através de uma posição fictícia no tabuleiro.

Existe um módulo intermediário composto pela classe **Player**, esta estrutura representa cada jogador, onde possui posição inicial no tabuleiro, nome do jogador, se consegue ou não finalizar o jogo e caso consiga, armazena um vetor que representa o caminho ideal. Da mesma forma que o módulo inicial, este módulo é completamente independente dos demais e foi criado para atender ao princípio da responsabilidade única e facilitar o entendimento e distribuição do código. Este módulo está armazenado em *src/player* e é composto por este único TAD.

O último módulo é composto pelas classes **Heap** e **NodeHeap**, que se encontram na pasta *src/heap*, este módulo encapsula os outros dois módulos a cima, pois seu objetivo é calcular caminhos no Grafo para cada jogador, realizando assim a interface entre os módulos. O **Heap** funciona como uma árvore, possuindo seus nós com filhos e pai. No entanto o heap é uma estrutura temporária que será criada a cada pesquisa que deve ser

feita no grafo. Cada nó da árvore é um TAD **NodeHeap** que ao ser instanciado representa um nó do grafo, a diferença é que nesta estrutura este nó terá um pai com o intuito de poder ser mapeado os níveis de uma árvore. Funciona da seguinte forma, o nó pai da árvore é o nó que representa a posição inicial do jogador recebido como parâmetro. A cada iteração será criada uma nova camada dessa árvore utilizando as ligações que esse nó teria no grafo. Logo esta nova camada com os nós que representam posições no grafo também terão suas ligações mapeadas e assim criando uma nova camada na árvore e assim sucessivamente. Quando ao mapear uma nova camada é encontrado o nó do grafo que representa a posição final do tabuleiro o algoritmo retorna o nó pai que levou à aquela posição e este nó, intância de um **NodeHeap** terá mapeado o caminho mais curto até a posição inicial do jogador. Este sistema apesar de utilizar de uma estrutura de árvore, sua lógica é baseada em uma pesquisa BFS, onde cada nova camada que é criada corresponde a uma iteração da BFS. Vale ressaltar que a cada iteração é verificado se aquele caminho já foi ou não utilizado ou se há outro jogador que iniciará naquela posição, caso haja, essa posição não será tomada, pois caso essa posição leve a um caminho menor este futuro jogador será o ganhador.

E finalmente existe o arquivo **main.cpp** responsável pela ordem de execução e em caso de empate de números de movimentos, o arquivo gera uma comparação simples entre os valores que os jogadores passaram de forma inversa aos turnos. Quando um ou mais jogadores tiverem valores menores que os demais, o demais serão exluídos e a comparação segue para turnos anteriores até encontrar um ganhador.

3. Instruções de compilação e execução

Para que se realize os comandos de compilação e execução é impressindível que esteja-se no diretório raiz do projeto, nesse caso no diretório **breno_pimenta**. Caso o arquivo esteja zipado, primeiro realize o unzip e depois entre no diretório. Os comandos e suas respectivas ações são:

- **make**: apenas compila.
- **make run**: compila e executa o programa.
- **make clean**: deleta arquivos compilados e executáveis.

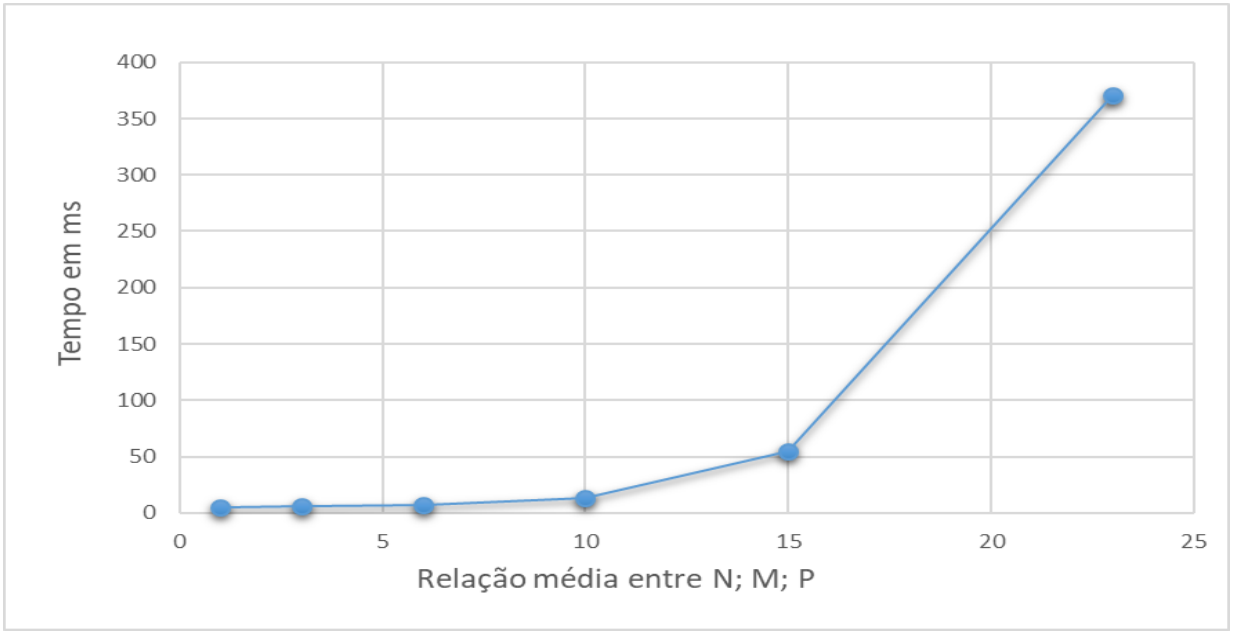
4. Análise de Complexidade

A metodologia para a análise da complexidade se dará da seguinte forma: será realizado o estudo primeiramente em cada módulo, sendo cada ação dos módulos analisadas separadamente e sem seguida as complexidades serão comparada entre si. Será tomado como referência n o número de colunas do tabuleiro, m o número de linhas do tabuleiro e p o número de jogadores.

1. **Módulo Graph -> Criação do Grafo:** Será percorrido os n*m nós de entrada e para cada nó serão realizadas até 4 verificações de posições subsequentes tendo uma complexidade temporal de O(n*m). O grafo terá uma instância do TAD **NodeGraph** para cada entrada do tabuleiro, tendo assim uma complexidade espacial de O(n*m). Será usada apenas uma instância do Grafo durante toda a execução.
2. **Módulo Heap -> Criação do Heap:** Para cada jogador teremos a partir de uma célula, até 4 outras células a serem mepeadas, porém estas células já mapeadas não serão acessadas novamente por este jogador. Sabemos também que as células iniciais dos outros jogadores não seram acessadas, logo haverá até n*m – (p-1) células mapeadas. Porém para o próximo jogador vale a mesma regra, não é armazenado as posições utilizadas pelo primeiro jogador para a comparação com o segundo, vide item cinco conclusão, portanto a complexidade temporal se dará por (n*m – (p-1)) * p, como n*m > p, pois caso contrário ou haverá jogadores na mesma posição e não será necessários duas execuções similares ou o jogador terá a posição inicial igual a posição final gerando uma complexidade de O(1), podemos assumir então que n*m*p sempre será maior que p^2, logo a complexidade temporal será O(n*m*p) (novamente vide conclusão para melhor entendimento). Já a complexidade espacial de uma árvore terá em seu pior momento uma representação completa do tabuleiro menos a posição final, tendo assim até n*m instâncias do TAD **NodeHeap** criadas, ou seja uma complexidade espacial de O(n*m), como a cada execução a estrutura é limpa da memória, existirá até no máximo uma árvore por vez.

Portanto no pior dos casos para a complexidade temporal teremos O(n*m) + O(n*m*p) resultando assim em uma complexidade temporal de O(n*m*p) para o algoritmo. E para a complexidade espacial no pior dos casos teremos O(n*m) + O(n*m), resultando em O(n*m).

Tabela 1. Tempo de execução da busca com aumento do tamanho do tabuleiro e número de jogadores.



5. Conclusão

Ao planejar o projeto imaginou-se que a construção da estrutura de representação do grafo seria o maior desafio. No entanto a estrutura de busca, apesar de se parcer um tanto trivial na teoria, se mostrou com uma implementação trabalhosa, principalmente diante da gestão de ponteiros. Um fator que vale mencionar foi a maior facilidade de gerência de memória neste projeto do que em projetos anteriores, proporcionadas por estruras de dados mais bem definidas permitidas pelo escopo.

Deve ser notado que é possível otimizar o algoritmo de busca, basta realizá-lo para os jogadores em paralelo, assim evitando acessar posições que outros jogadores acessaram em turnos mais recentes, diminuindo a complexidade temporal do algoritmo. Foi um erro inicial do planejamento do projeto não ter sido implementado dessa forma e infelizmente com o tempo que restou do prazo, a partir da verificação do problema, não foi suficiente para solucioná-lo.

6. Bibliografia

ZIVIANI, N. Projeto de Algoritmos com implementações em pascal e c. 3ª edição. Cengage Learning.

CORMEN, T. H. Et al. Algoritmos: Teoria e Prática. 3a edição. Elsevier, 2012

C++ reference. Disponível em: < https://en.cppreference.com/w/>. Acesso em: 08 nov. 2019.