Relatório do EP5

Juliano Garcia de Oliveira Nº USP: 9277086

28 de Novembro, 2016

1. Especificação do EP

O EP5 consiste em implementar o jogo de tabuleiro Hex, que é um jogo de estratégia para dois jogadores, jogado em um tabuleiro de hexágonos. O tamanho do tabuleiro definido no EP é de 14×14, e as regras do jogo são simples: O jogador com as peças *brancas* tenta formar um caminho da linha 0 até a linha 13 $(n-1)$, e o jogador com as peças **pretas** tenta formar um caminho da coluna 0 até a coluna 13 $(n-1)$. A cada turno, cada jogador pode colocar uma peça no tabuleiro, e o jogo termina quando um dos jogadores construiu o caminho necessário para a vitória. Foi **provado por John Nash** que o jogo sempre tem um vencedor, nunca sendo possível um empate (para qualquer tamanho de tabuleiro).

Neste EP deve ser implementado o jogo, tal que o programa faça papel de um dos jogadores. Ele recebe entradas com os movimentos do oponente, e imprime as jogadas com os movimentos que o algoritmo calculou para ele. Ao final, deve imprimir qual jogador ganhou a partida.

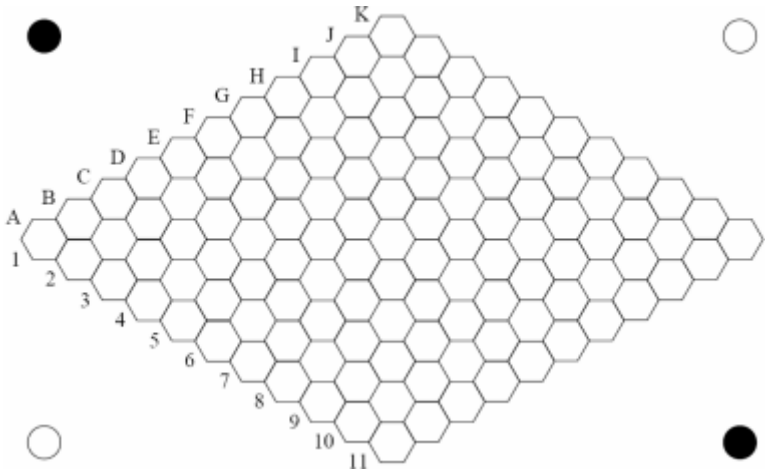

Figure 1: Jogo Hex

2. Estrutura do EP

O fluxo do meu EP segue uma sequência lógica bem definida, e está simplificada abaixo:

- Recebe entrada inicial e inicializa o tabuleiro;
- Verifica condições para a *pie rule* e a executa se necessário;

- Entra no *game Loop*:

    1. Se ninguém venceu, vai para 2. Se não, vai para 7;
    2. O computador decide qual movimento irá fazer;
    3. Executa o movimento;
    4. Pede a entrada do oponente, lendo-a da entrada padrão (*stdin*);
    5. Executa o movimento;
    6. Volta para 1;
    7. Imprime quem ganhou;

- Termina a execução.

3. Estruturas de Dados

Neste EP,eu quis implementar estruturas de dados diferentes, além de usar um pouco mais a o conceito de tabela de símbolos explorado no EP4. Ao pesquisar artigos e dicas sobre como implementar a resolução de um jogo, encontrei várias estruturas de dados interessantes para utilizar no EP em questão.

Abaixo estão explicadas as estruturas de dados e o seu propósito no EP, junto com o nome que utilizo no código em si. Informações mais detalhadas sobre cada uma pode ser encontrada nos comentários do próprio código fonte do EP.

3.1 *Jogo*

Estruturas de dados relacionadas ao próprio jogo Hex.

- Cores (*color*): Uma cor é um inteiro que representa as cores. Pode ser *WHITE*, *BLACK*, *NONE* ou em um caso especial, *INVALID*;
- Hexágono (*Hexagon*): O hexágono é a menor unidade do tabuleiro, e armazena somente uma cor;
- Tabuleiro de Hexágonos (*hexBoard*): É o tabuleiro do jogo, além de fornecer o game state. O tabuleiro propriamente dito é um vetor com os Hexágonos. o *game state* armazena o número do turno atual e o jogador atual, além do tamanho do tabuleiro.

Esta estrutura de dados (*hexBoard*) possui várias operações, como duplicar um tabuleiro, destruir, transformar seus dados em uma *string*, imprimir o estado atual do jogo, obter as bordas, etc. Cada hexágono do tabuleiro tem um número ou *id*, sendo que $id \in \{0, 1, 2, ..., n+4\}$. Os últimos 4 hexágonos são adicionais e representam as bordas do tabuleiro. Os índices $[i, j]$ de um certo hexágono com número *id* na matriz da especificação do tabuleiro é dada por:

$$f(id) = \begin{cases} i = & \text{id } \mod 14 \\ j = & \lfloor \text{id } \div 14 \rfloor \end{cases}$$

3.2 *Dijkstra* O algoritmo de decisão implementado calcula caminhos para determinar qual a jogada irá fazer. Para determinar isto, ele usa um algoritmo para achar o caminho com a menor distância até a vitória. Esse algoritmo é o Dijkstra, bastante usado em problemas de *path finding* (labirintos, etc).

- Dijkstra Storage (*DjkStorage*): Armazena informações após rodar o algoritmo Dijkstra, como o tabuleiro atual, os *nodes* do caminho escolhido, as distâncias, o *node* inicial e o *final*, etc;
- Dijkstra Path (*DjkPath*): Representa um caminho já calculado, e é construído a partir de um *DjkStorage*. O caminho é o menor caminho do *node* inicial até o *final*, e armazena o valor de cada hexágono no caminho, além da "distância" total;

  O algoritmo de Dijkstra usa uma *bitmask* para especificar quais os caminhos permitidos. Essa *bitmask* é composta de 3 posições, $b_1b_2b_3$. b_1 Significa se é permitido a cor preta, b_2 a cor branca e b_3 hexágonos sem cores. Por exemplo, a *bitmask* 101 permite cores pretas e hexágonos sem cor, porém não permite brancos. Um Dijkstra que usasse essa *bitmask* faria um caminho com hexágonos que só satisfazem essa propriedade.

3.3 *Transposition Table*
A *transposition table* é uma tabela de símbolos que armazena tabuleiros com configurações diferentes, e o seu respectivo valor. O valor de um tabuleiro é usado no algoritmo de decisão da jogada implementado. Como o algoritmo de decisão calcula várias jogadas possíveis, pode ser que em um momento ele chegue em uma configuração que já foi calculada anteriormente.

Para não ter que recalcular, armazenamos na tabela de símbolos o tabuleiro e o valor já calculado. Isto é uma forma de Programação Dinâmica, mais conhecida como **Memoization**, que é bastante aplicada em algoritmos recursivos (como é o meu caso).

Formalmente, a *transposition table* é um mapeamento $T : \mho \to \Psi$, de um conjunto de configurações de tabuleiro $\mho$ para um conjunto de valores (pesos) Ψ.

- *Transposition table* (*hashTable*): Por questões de performance e otimização, a *transposition table* foi implementada usando uma *Hash Table*. Na matéria de Técnicas de Programação I, o professor explicou a técnica de **Linear Probing** para construir uma *Hash Table* facilmente com uma lista ligada. Usei uma implementação de *Hash Table* disponível em domínio público por Keith Pomakis (1998), e simplifiquei um pouco. A implementação da Hash table feita por Keith Pomakis está disponível no site dele.

  O algoritmo de *Hashing* que eu utilizo é o **djb2**, para as *strings* (é simples de entender e é bom para cadeia de caracteres).

3.4 *Árvores de jogo*
No algoritmo de decisão, é necessário avaliar as posições válidas de movimentos disponíveis, junto com os diversos tabuleiros finais que os movimentos levam. Essa informação é organizada em uma estrutura chamada *game tree* ou árvore de jogo.

A árvore de jogo é uma árvore que possui uma raiz (o estado inicial do tabuleiro), e os nós são diferentes configurações do tabuleiro, e as arestas / ligações representam um movimento válido na árvore. A informação é armazenada de maneira hierárquica, sendo que o um *jogo* é um caminho de uma folha até a raiz da árvore. Porém, na maioria dos algoritmos e implementações, quase não se usa uma árvore de jogo completa, pois gastaria muito tempo e espaço. No meu caso, defini o máximo de profundidade da árvore como 4.

O algoritmo usado para decidir a jogada usa essa estrutura para simular os jogos e atribuir pesos a cada um deles, escolhendo um deles no final de acordo com o peso. Porém, é importante observar

que essa estrutura foi implementada **implicitamente** através de **recursão**, ou seja, a árvore é criada através de chamadas recursivas, e é percorrida com uma *dfs(Depth-first Search)*.

4. Algoritmo

4.1 *Minimax*

O algoritmo principal usado para escolher o movimento é o *MTDf(Memory-enhanced Test Driver with node n and value f)*, que é um tipo de algoritmo *Minimax*. Primeiramente, o algoritmo *Minimax* é um algoritmo de busca exaustiva na árvore de jogo usando uma *dfs*, e retorna um valor para a raiz da árvore de jogo. O algoritmo possui duas etapas, a *maximização* e a *minimização*. A fase de maximização ocorre em todas as configurações do tabuleiro na qual é a vez do primeiro jogador, e a minimização ocorre em todas as configurações do tabuleiro que é a vez do segundo jogador.

A busca na configuração de tabuleiro na fase de maximização retorna o maior valor (peso) calculado para as configurações dos nós sucessores desta. Já a busca na configuração de tabuleiro na fase de minimização retorna o menor valor (peso) calculado aos sucessores. Estes valores são então levados para a posição anterior do tabuleiro, e assim por diante, até chegar à raiz da árvore.

Se em algum momento da busca o primeiro jogador venceu, então é retornado o maior valor possível (no meu caso, este valor é 3000). Se o segundo jogador venceu, o valor é o menor possível (é minimizado).

Abaixo temos um exemplo de uma árvore de jogo com as etapas de maximização e minimização.

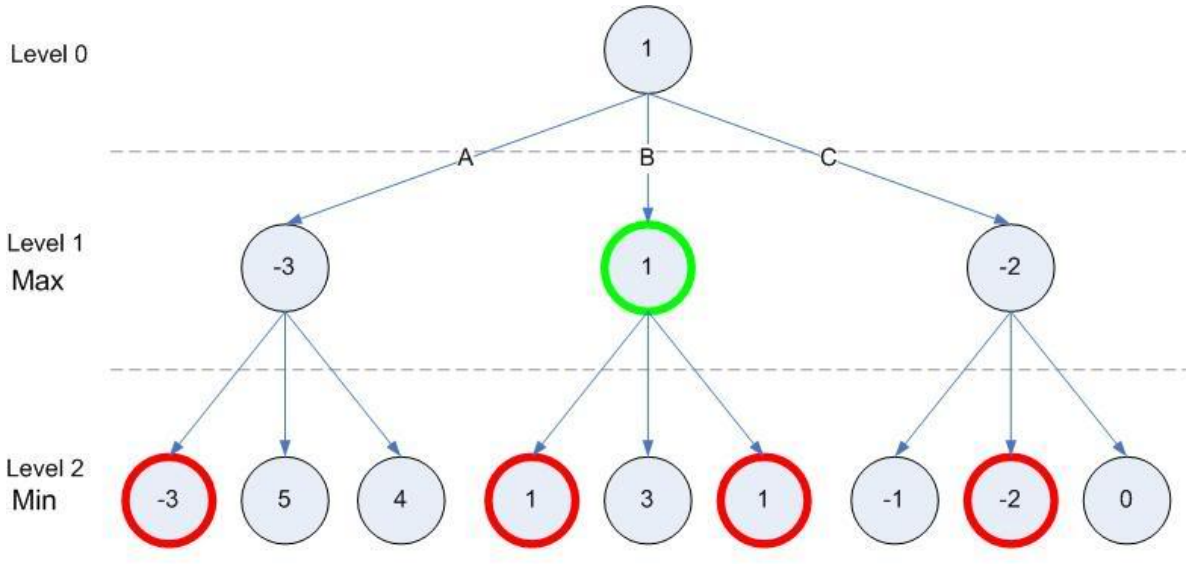

Figure 2: Minimax

4.2 *Algoritmo Alpha-beta*

O algoritmo *alpha-beta* é usado para otimizar o algoritmo *Minimax*. Neste algoritmo, para cada nó da árvore de jogo, temos dois valores, α e β. O α é o limite inferior (*lower bound*), que deve ser maximizado no algoritmo de busca, e o β é o limite superior (*upper bound*), que deve ser minimizado no algoritmo de busca. Esse algoritmo é útil pois permite eliminar ramos da árvore, ou seja, são configurações que não precisam ser avaliadas recursivamente. Isso é chamado de *alpha-beta prunning*, e cada um deles (o α e o β) tem uma condição diferente para acionar esse "corte de ramo". Mais detalhes sobre o algoritmo pode ser visto em *Minimax and Alpha-beta prunning, Cornell University*.

4.3 *A estratégia*

No meu EP, ao fazer uma jogada o programa faz algumas escolhas. Inicialmente, se for uma primeira jogada, dependendo da cor do jogador, o algoritmo tenta escolher a peça do meio, ou então faz uma jogada aleatória no tabuleiro.

O algoritmo principal usa o *MTDf* para maximizar / minimizar as jogadas, então executa o *alpha-beta* um certo número de vezes (o padrão são 4), e vai trocando os valores, como é especificado no algoritmo MTDf. No final, ele retorna o melhor valor encontrado.

No final das contas, tudo depende do valor que damos ao tabuleiro. Eu construo os valores com base na distância do caminho até a vitória. Quanto maior for a distância até a vitória, menor é a pontuação de um determinado tabuleiro. Essa distância é calculada usando o algoritmo de *Dijkstra*, e é feita para cada tabuleiro da Árvore de jogo.

É neste momento que se usa a *transposition table*: Se o peso de um determinado tabuleiro já foi calculado, então não preciso rodar o algoritmo de Dijkstra novamente. Como a tabela de símbolos é uma *Hash Table*, o tempo de inserção e busca é quase constante ($\mathcal{O}(1)$), sendo bastante útil. O uso da *transposition table* no *alpha-beta* é chamado de *cache* no meu EP.

A *transposition table* armazena os tabuleiros através de uma função de conversão para *string* que está disponível no arquivo *hexBoard.c*. A *string* gerada é algo como "000100200100000000000001010002...000001", onde 01 é uma peça da cor branca, 02 da cor branca, e 00 um local vazio.

No final, depois das etapas de maximização e minimização, o algoritmo devolve o número do hexágono que julgou ter a melhor chance. O tamanho máximo da árvore influencia tanto no tempo quanto na "inteligência" do algoritmo, e foi fixada uma profundidade máxima de 4 níveis, que dá resultados aceitáveis sem demorar tanto.

5. Resultados

A estratégia de verificar o caminho até a vitória não é muito complexa, mas dá alguns resultados aceitáveis. Contra um jogador aleatório, o programa ganha facilmente, o que não acontece contra jogadores que usam táticas diferentes.

O algoritmo não dá resultados muito inteligentes, sendo fácil uma pessoa que conhece um pouco da regra do jogo ganhar dele. Um dos maiores pontos fracos é que o meu algoritmo não tenta bloquear o outro jogador, mas sim tenta fazer o seu próprio caminho. Outro ponto fraco é que ele não determina as "pontes", que fariam com que ele tivesse uma melhor estratégia.

O ponto forte é que ele tenta acabar o jogo rapidamente por um caminho determinado pelo algoritmo de *Dijkstra*, sendo que mesmo se alguém o bloquear, ele consegue na maioria das vezes sair do bloqueio.

A falha mais fácil que consegui encontrar foi fazer o caminho o mais longe possível dos hexágonos do computador, assim somente nos turnos finais o computador tenta mudar sua rota, mas já é tarde demais!

OBS: Em relação ao tempo, foi fixado um limite para não ultrapassar o proposto ($0.3s$), então quando o algoritmo chega em $0.285s$ de execução, a recursão é interrompida manualmente. Como eu uso a variável *CLOCKS_PER_SECOND*, talvez esse tempo varie dependendo do computador.

Encontrei várias estratégias e outros modos de implementar, mas por falta de tempo não dá para colocar todas no meu programa. Achei o EP bem interessante para aprender a implementar tipos diferentes de estruturas de dados que nunca tinha ouvido falar mas não são tão difíceis de entender e implementar.

Fontes:

Algorithmic Approaches for Playing and Solving Shannon Games, Rasmussen. Queensland University of Technology, 2007. Disponível em http://eprints.qut.edu.au/18616/1/01Thesis.pdf

A Move Generating Algorithm for Hex Solvers. Rasmussen, Rune K. and Maire, Frederic D. and Hayward, Ross F., 2006. Disponível em http://eprints.qut.edu.au/5121/1/5121_1.pdf

Multigame — An Environment for Distributed Game-Tree Search. Romein, J., 2001. Disponível em https://www.cs.vu.nl/en/Images/romein_thesis_tcm210-92624.pdf