UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

DEPARTAMENTO DE INFORMÁTICA APLICADA

Implementação de uma IA para o jogo Batalha de Peões

Bruno Iochins Grisci - 208151
Jorge Alberto Wagner Filho - 205975
26 de junho de 2013

1 INTRODUÇÃO

Neste relatório, iremos dar detalhes da criação e implementação do trabalho final da disciplina de Inteligência Artificial, um programa capaz de jogar satisfatoriamente o jogo Batalha de Peões, uma variação do xadrez tradicional. Serão apresentados o problema, as estruturas de dados utilizadas e os algoritmos propostos para a solução.

2 O JOGO DA BATALHA DE PEÕES

Batalha de Peões é uma versão simplificada do jogo de Xadrez[1]. Consiste de um tabuleiro de 64 casas e dez peças para cada jogador: oito peões e duas torres, dispostas na mesma formação inicial do Xadrez. As peças mantém as mesmas regras de movimentação do Xadrez, incluindo o salto duplo inicial e o en passant[2] para os peões. Cada jogador pode mover uma peça por vez dentro do conjunto de movimentos válidos, e a condição de vitória é eliminar todos os peões adversários ou então levar um de seus peões até a última linha do tabuleiro. Foi especificado que o tempo máximo para a decisão de uma jogada é de 5 segundos.

3 AMBIENTE DE TRABALHO

Para o desenvolvimento do programa utilizamos a linguagem Java com a IDE NetBeans e sistema operacional Windows 7 e Mac OS X 10.8.

4 ESTRUTURAS DE DADOS E REPRESENTAÇÃO DO JOGO

Ao implementarmos um jogo, as estruturas de dados que utilizamos são extremamente importantes. Adotar uma representação ineficiente pode comprometer os resultados mesmo de um excelente algoritmo. As nossas peças são uma classe Java que guarda alguns métodos e atributos específicos. Para nosso trabalho, usamos duas representações distintas.

4.1 INTERFACE GRÁFICA

Para a apresentação do jogo na tela, usamos um vetor ordenado de JLabels, com índices de 0 a 63, cada um referente a uma das casas do tabuleiro, começando no canto inferior esquerdo e crescendo para a direita, de forma que para avançar uma casa para frente é necessário somar oito ao índice atual. Esta representação facilitou a implementação da interface gráfica do jogo, que escreve nos JLabeles caracteres unicode específicos para peças de Xadrez. Para atualizar o tabuleiro bastava apagar a peça de um JLabel e escrevê-la no próximo. As peças são representadas em um vetor de vinte posições onde índices pares correspondem a um time e índices ímpares a outro, sendo as dezesseis primeiras posições os peões e as quatro últimas as torres. Este vetor armazena o índice de um outro vetor, de casas, que armazena o índice do vetor de peças, facilitando o acesso direto à localização de uma peça no tabuleiro. Esta implementação pode ser vista mais claramente na Figura 4.1.

4.2 LÓGICA

A representação para a lógica do programa precisava ser mais eficiente que um vetor ordenado para economizarmos memória e principalmente não perdermos tempo gerando e analisando os movimentos do jogo. A solução adotada foi o emprego de BitSets do Java para implementarmos bitboards. BitSets são vetores de bits, muito eficientes e que permitem utilização de expressões lógicas.

Nosso jogo usa diversos BitSets para representar o tabuleiro. Cada um possui 64 posições, para as casas do tabuleiro, sendo que valem 1 as casas ocupadas por uma peça e 0 as vazias. Evidentemente é preciso distinguir os jogadores e os tipos de peça, por isso são usados quatro BitSets distintos: peões brancos, peões pretos, torres brancas e torres pretas. A Tabela 4.1 ilustra visualmente o BitSet dos peões brancos em sua configuração inicial, sendo a posição do canto inferior esquerdo a de índice 0 e a do canto superior direito a de índice 63.

As duas representações são constantemente sincronizadas uma com a outra para refletir na tela o que a lógica está retornando como jogada e vice-versa.

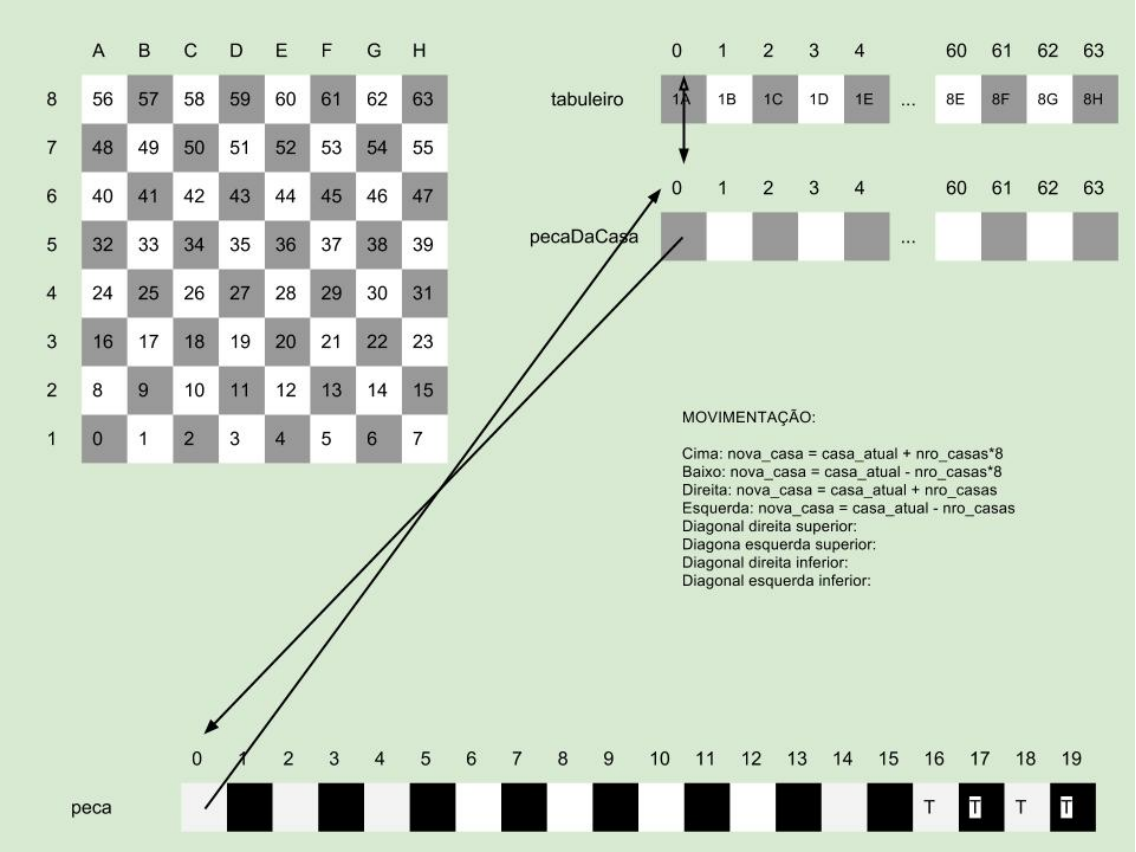

Figura 4.1: Esboço dos vetores para a representação gráfica do tabuleiro

5 NOSSA INTELIGÊNCIA ARTIFICIAL

Com o estado do jogo representado através de BitSets, o programa deve ser capaz de procurar movimentos para as peças e fazer a melhor jogada possível com o objetivo de vencer a partida.

5.1 GERAÇÃO DOS MOVIMENTOS

O programa precisa gerar todos os movimentos possíveis do jogo, ou seja, deslizamento horizontal e vertical das torres, captura da torre, avanço simples do peão, avanço duplo inicial do peão, en passant, captura diagonal do peão, respeitando as restrições de não "pular"peças e os limites do tabuleiro.

Como estamos usando BitSets, é apenas uma questão de operá-los. Para detectarmos as bordas criamos dois BitSets, um com a primeira coluna com todos os elementos em 1 e o restante em 0, e um com a última coluna com todos os elementos em 1 e o restante em 0. Para sabermos se a peça está em uma destas colunas, portanto, basta fazer a operação lógica and entre o seu BitSet e o BitSet dos limites. As bordas superior e inferior podem facilmente ser identificadas pelo valor do índice da casa.

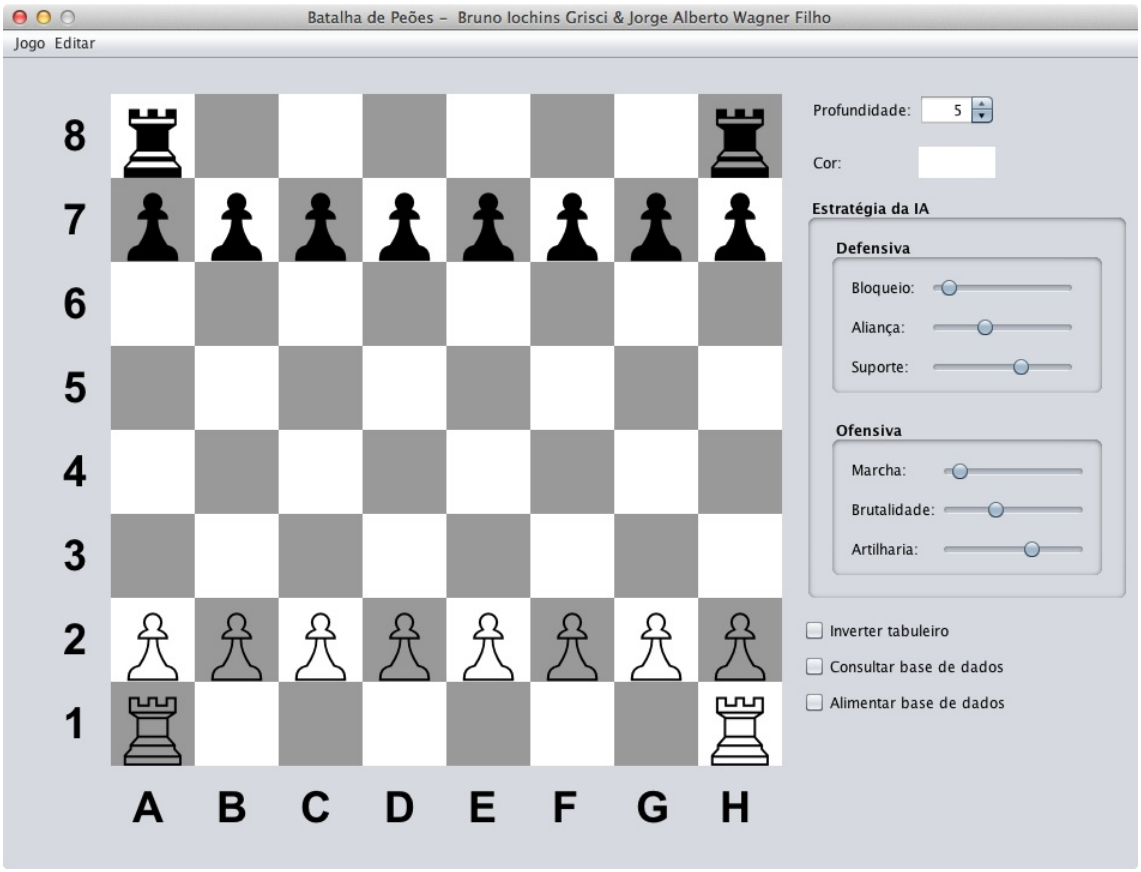

Figura 4.2: Interface Gráfica

É trivial encontrar BitSets para diversas situações. Caso queira saber se o peão branco na posição n pode fazer a captura na diagonal, por exemplo, basta fazer a operação or entre os BitSets das torres e peões pretos e então ver se as posições n+7 ou n+9 contém 1. Para o salto duplo do peão, precisamos testar se ele está em sua posição inicial m e se a posição n+16 está com 0 no BitSet originado da operação or entre todos os BitSets, pois não podem haver peças na casa destino do avanço do peão. Da mesma forma, é trivial encontrar situações de en passant. O único diferencial para este caso consiste na dependência não apenas dos BitSets do estado atual, mas também do estado anterior a ele, tendo em vista as regras que regem esta jogada. Para cada jogada obtemos todos os movimentos possíveis para cada peça. Embora possa parecer muito custoso, como BitSets permitem percorres o vetor de 1 em 1, pulando posições com 0, economizamos recursos computacionais.

5.2 Função avaliadora

A função avaliadora é um dos elementos mais importantes de todo o programa. É ela quem determinará o valor de um dado estado do jogo, pautando as decisões do algoritmo minimax.

Tabela 4.1: Representação do tabuleiro em BitSet para os peões brancos em sua posição inicial

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0

Utilizamos a seguinte função:

avalIA = marcha×$\sum_{peoes} linha$+indice do peão mais avançado+brutalidade×NroDePeoes+ artilharia × NrodeTorres

avalHumano = detencao × $\sum_{peoes}(8 - linha)$ + índice do peão mais avançado + aliança × NroDePeoes + suporte × NrodeTorres

aval = avalIA - avalHumano

com os seguintes valores default:

$$\begin{cases} \text{marcha = detenção = 1} \\ \text{brutalidade = aliança = 7} \\ \text{artilharia = suporte = 13} \end{cases}$$

Esta configuração dá ênfase à utilização das torres e avanço dos peões e foi decidida após testes com outros valores. Para estados de vitória ou derrota, é retornado um valor muito alto para forçar a escolha.

5.3 Algoritmo Minimax

O algoritmo Minimax percorre toda a árvore de estados possíveis a partir do estado atual até a profundidade especificada pelo jogador, sempre buscando minimizar as avaliações das jogadas do adversário e maximizar as próprias jogadas. Trata-se de um algoritmo relativamente simples de ser implementado a partir do momento em que se tem como gerar todos os movimentos possíveis a partir de um dado estado, como visto na seção Geração dos movimentos, e avaliar o valor de cada estado, como visto na seção Função avaliadora.

5.4 Otimizações

Para otimizar o algoritmo Minimax permitindo alcançar profundidades maiores dentro do tempo máximo estipulado de 5 segundos, implementamos a Poda Alfa-Beta. Esta otimização, largamente conhecida e frequentemente associada ao Minimax, interrompe a recursão sempre que possível evitando analisar sub-árvores que garantidamente não tenham possibilidade de apresentar valores melhores dos que os já encontrados.

Para evitar a repetição de buscas muito comuns, implementamos uma base de dados de estados que, por meio de uma tabela Hash, mapeia um estado origem para um estado destino e salva a profundidade do Minimax usada nesse cálculo. Isso possibilita manter uma profundidade padrão menor para os cálculos, mas ao mesmo tempo permitindo usar o resultado de uma busca com profundidade maior automaticamente para este estado caso ela já tenha sido pré-computada anteriormente, buscando executar uma jogada mais inteligente quando possível sem nunca exceder o tempo limite estabelecido. A implementação de uma base de dados também abre a possibilidade de buscar conhecer os estados mais frequentes com a IA jogando contra si mesma ou outras IAs, por exemplo, e pré-computar os resultados do Minimax com profundidades grandes para estes estados, ou pré-computar os resultados do Minimax para os estados mais comuns de abertura ou encerramento de jogo.

6 Recursos do programa

- Seleção da cor do time do jogador (branco sempre começa)
- Seleção da profundidade do minimax
- Escolha dos valores das constantes da função avaliadora
- Espelhamento do tabuleiro
- Deleção de peças
- Recortar e colar peças
- Desfazer e refazer movimentos
- Alimentar e consultar o banco de dados

7 Resultados

Nosso programa foi capaz de atingir até o nível 7 de recursão no algoritmo minimax com poda alpha-beta sem estourar o limite de 5 segundos proposto para uma jogada. Naturalmente, quanto mais avançada está a partida e menor o número de peças no tabuleiro, mais rápido ele se torna. Para as duas fases do torneio utilizamos níveis 5 e 6 para garantir que não estouraríamos o tempo.

Para colocar nossa IA à prova, participamos de um torneio de duas fases com os demais trabalhos da turma. Jogamos com os grupos duas vezes, uma como equipe branca (a que começa) e uma como equipe preta. Os resultados podem ser vistos nas Tabelas 7.1 e 7.2 e nos gráficos 7.1 e 7.2. É possível notar que tivemos uma pequena variação negativa quando fomos os segundos a mover a peça, mas isso talvez se deva à vantagem que o primeiro jogador sempre leva no Xadrez. Acreditamos que os resultados mostram uma avaliação positiva da nossa implementação e estratégia.

Tabela 7.1: Resultados do primeiro turno do Torneio

Cor	Total de jogos	Vitórias	Empates	Derrotas
Branco	11	10	1	0
Preto	11	8	2	1
Total	22	18	3	1

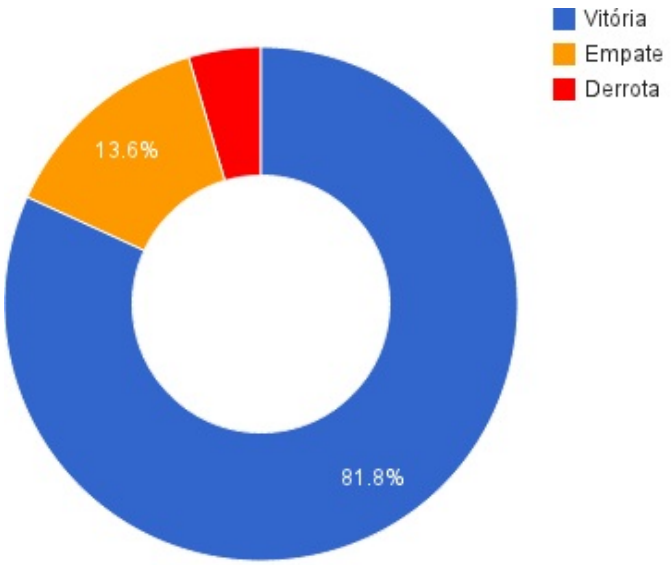

Figura 7.1: Gráfico do aproveitamento percentual do primeiro turno do campeonato

REFERÊNCIAS

[1] Chess http:en.wikipedia.orgwikiChess

[2] En passant http:en.wikipedia.orgwikiEn_passant

[3] Michael Goeller, Pawn Battle Rules and Strategies 2007

Tabela 7.2: Resultados do segundo turno do Torneio

Cor	Total de jogos	Vitórias	Empates	Derrotas
Branco	15	12	1	2
Preto	15	11	2	2
Total	30	23	3	4

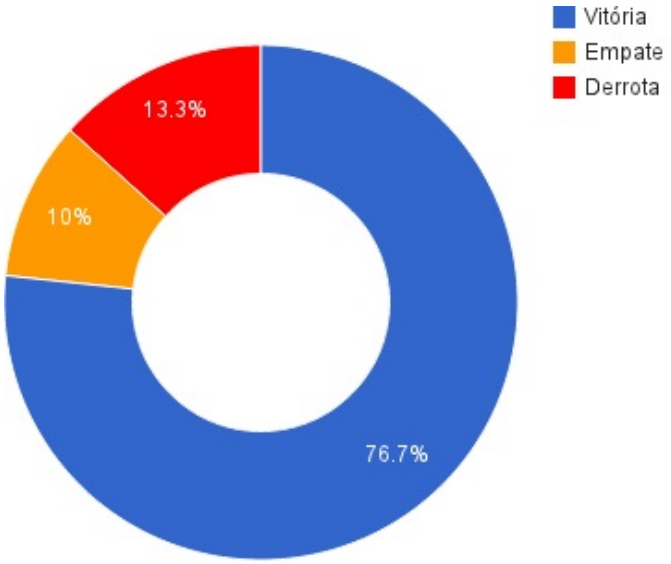

Figura 7.2: Gráfico do aproveitamento percentual do segundo turno do campeonato

[4] Joey K Black, Overview of a Simple Chess Implementation

[5] David Ravn Rasmussen, Parallel Chess Searching and Bitboards 2004