IA para o jogo Fanorona modificado

Cristiano Medeiros Dalbem - 173362
Luca Couto Manique Barreto – 173051

Trabalho Final de Inteligência Artificial - INF01048

Instituto de Informática
Universidade Federal do Rio Grande do Sul

25 de novembro de 2010

1 Introdução

Fanorona é um jogo de tabuleiro criado em Madagascar. Dentre suas versões, a mais comum é jogada em um tabuleiro 9x5[1]. Para o trabalho desta disciplina utilizamos uma versão simplificada desta, que utiliza um tabuleiro 7x5. Entre as características do jogo, a mais peculiar é a jogada de captura, que consiste em um movimento para uma casa vazia, em frente a uma peça adversária, e resulta na captura de todas as peças inimigas adjacentes naquela direção.

Para o grupo, o jogo se mostrou divertido. Porém, tratar casos especiais, como jogadas obrigatórias e múltiplas, exigiu uma atenção maior à parte de implementação de regras do que à parte de inteligência artificial propriamente dita.

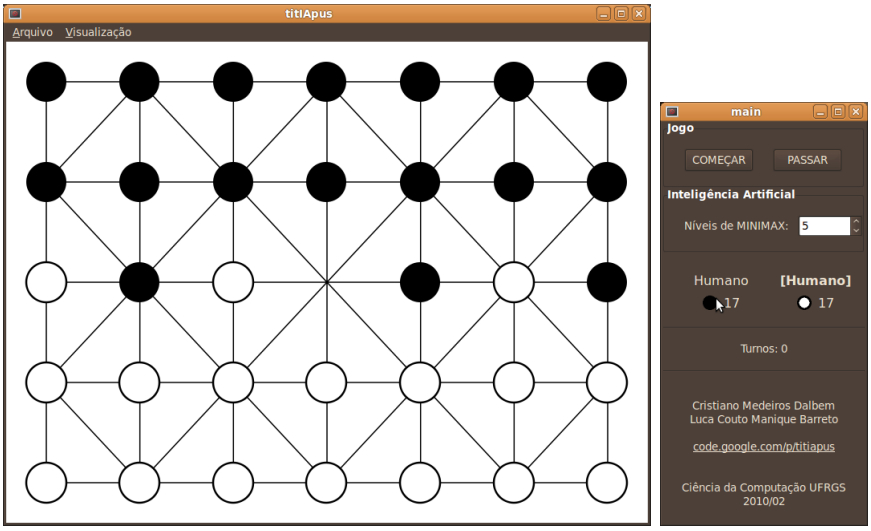

Figura 1: Tela principal.

2 Características Gerais

O programa foi desenvolvido na linguagem C++, utilizando-se apenas da biblioteca padrão STL[3] e da biblioteca gráfica GTK[4], em conjunto com o software Glade[5], para a criação da interface gráfica.

2.1 Arquitetura do Software

Estes são os principais módulos:

Estado é a classe que representa um estado de jogo. Contem a configuração das peças no tabuleiro, um vetor das jogadas feitas no turno atual e várias funções que calculam e listam as jogadas obrigatórias e as jogadas possíveis. Estado é uma classe minimizada na engenharia do código, já que ela será replicada nos nodos do Minimax.

Interface encapsula funções das callbacks da GTk, referentes aos cliques na tela. Além disso, as funções de desenhos e marcações gráficas.

Jogador é a classe genérica para Jogadores. É a classe pai dos jogadores Humano e jogadores Máquina.

Jogo Guarda numero de jogadas já feitas, estado atual do jogo e funções de jogo como passar(), comecar(), novo(), executarTurno() e executarJogada().

utils tem funções de utilidade e debug.

2.2 Sistema Jogo/Jogador

O controle do jogo se dá pelo sistema de comunicação Jogo/Jogador através do "loop"do Jogo. Uma partida é uma instância da classe Jogo, que possui dois ponteiros para instâncias da classe Jogador. Estas recebem o estado do jogo (tabuleiro) e decidem e comunicam ao jogo sua jogada. A classe Jogador possui como especializações (classes filhas) JogadorHumano, cujos métodos de decisão de jogada são baseados na interface gráfica, e JogadorComputador, cujos métodos aplicam as funções de Minimax e avaliação de utilidade.

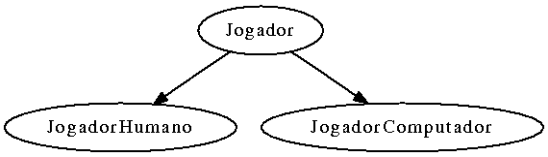

Figura 2: Relacionamento de herança.

3 Estrutura de dados do Tabuleiro

A representação do tabuleiro é uma parte extremamente importante no programa. Algoritmos de decisão de jogada precisarão criar uma extensa árvore de estados (configurações do tabuleiro), o que consome proporções

enormes de memória e tempo de processamento de crescimento exponencial. Portanto, nos preocupamos em fazer a classe ser compacta e possuir métodos de acesso eficientes.

3.1 Classe Estado

A classe definida pelo grupo consiste em uma matriz de bits, onde cada casa é representada por 2 bits. Esta representação nos permite definir até quatro estados para cada casa. Os estados definidos são: casa vazia, peça branca, peça preta e casa reservada (utilizado para indicar se a casa já foi visitada durante uma jogada múltipla).

Em baixo nível, a implementação consiste em um vetor de tamanho 5 (número de linhas) de short int's (inteiros de 16 bits), onde cada short int é manipulado por operações bitwise (bit-a-bit) a fim de se obter o valor de um par de bits (entre 0 e 6, o número de colunas). Assim, cada matriz ocupa 10 bytes (16x5 = 80 bits).

Além da matriz, um estado possui ainda duas variáveis responsáveis por guardar a quantidade de peças de cada jogador. Cada variável ocupa um byte, totalizando assim 12 bytes por estado de jogo.

3.2 Manipulação

3.2.1 Acesso a casas

O acesso (leitura e escrita) a casas do tabuleiro, como já citado, é dado a partir de operações bitwise. Em resumo, após selecionar a linha desejada (no vetor de short int's), a coluna é encontrada com operações de shift e isolada com aplicações de máscaras AND.

3.2.2 Busca por jogadas possíveis

Ao buscar por estados de jogo possíveis (i.e. estados atingíveis), a classe deve primeiro checar a existência de jogadas obrigatórias. Em caso de existência destas, o próximo passo é determinar quais são as jogadas múltiplas possíveis (a grosso modo, isso consiste em aplicar a busca de jogadas obrigatórias em cada peça avaliada) . Caso contrário, buscam-se as jogadas que não geram captura de peça.

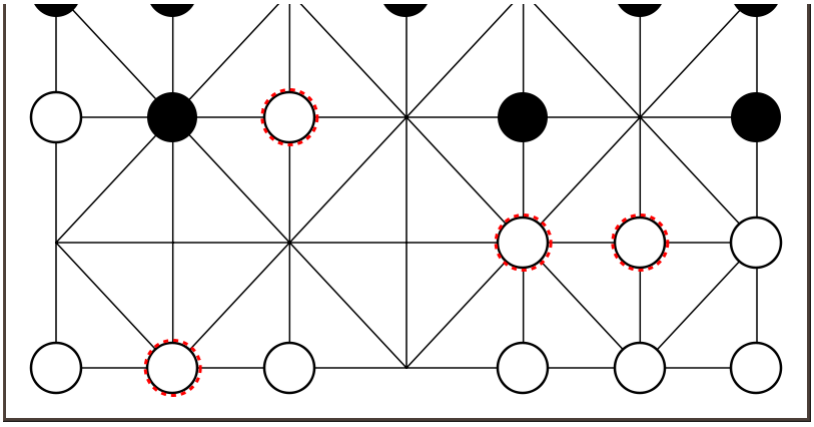

Figura 3: No exemplo, jogo mostra quais peças possuem jogadas obrigatórias. Essa funcionalidade é extremamente útil para jogar Humano contra Humano.

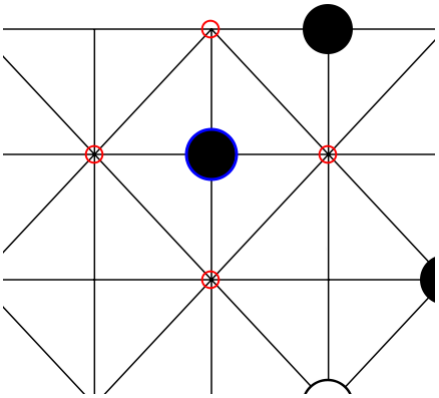

Figura 4: Jogadas possíveis para a peça selecionada.

4 Função de Avaliação

A função de avaliação é outro componente crucial do sistema. Por ser aplicada largamente, não se pode exigir operações complexas, nem abusar de muitos aspectos de avaliação. Ao mesmo tempo, os pesos devem ser corretamente balanceados, para evitar a super/subestimação de um estado. O projeto do grupo incluía a implementação de Redes Neurais Artificiais para balancear estes pesos, mas contratempos impediram esta tarefa de ser concluída.

4.1 Aspectos e Pesos

Com medo de deixar a função de avaliação pesada, o grupo preferiu deixá-la extremamente simples, atribuindo um único aspecto (com peso 1): a razão de peças do jogador e peças no tabuleiro.

5 Minimax

5.1 Profundidade alcançada

Em testes realizados os computadores dos próprios membros do grupo, a implementação conseguiu atingir, dentro do tempo limite (5s por turno), a profundidade de 9 níveis. Entretanto, não podendo contar com estes no local do campeonato, a profundidade máxima atingida nos computadores do laboratório foi de 6 níveis, mas que ficavam muito pesado nos momentos finais do jogo, então jogamos com 5 níveis.

float

5.2 Otimização utilizada

Duas otimizações foram aplicadas à implementação do algoritmo Minimax. A primeira foi a poda alfa-beta (alpha-beta pruning), que afetava diretamente na execução do programa. A segunda, afetando apenas quesitos relacionados com tempo de programação, foi a implementação da versão Negamax.

```
function negamax(node, depth, α, β, color)
    if node is a terminal node or depth = 0
        return color * the heuristic value of node
    else
        foreach child of node
            α := max(α, -negamax(child, depth-1, -β, -α, -color))
            {the following if statement constitutes alpha-beta pruning}
            if α≥β
                return α
        return α
```

Figura 5: Pseudo-código do Negamax. [2]

6 Conclusões

Ficamos bastante descontentes com os resultados da nossa IA no campeonato, mas apesar disso estamos orgulhosos do Software desenvolvido. Na realidade gostaríamos de ter gasto muito mais tempo pensando em funções de avaliação mais elaboradas, e otimizamos nosso desenvolvimento utilizando boas práticas de programação justamente pra que pudéssemos atingir esse objetivo.

Achamos que a complexidade do trabalho exigiu demais da nossa programação com interfaces e de menos no que realmente interessava no âmbito da cadeira, que seria implementar um Minimax eficiente e inteligente. Obtivemos êxito na eficiência, mas fracassamos na inteligência, o que é uma pena, pois nossos resultados no campeonato não representaram nosso esforço.

Em última análise, foi bastante construtivo e divertido desenvolver esse trabalho prático, mas gostaríamos de ter gasto mais tempo com técnicas de IA, como Programação Genética por exemplo.

Referências

[1] Wikipedia: Fanorama - website: http://en.wikipedia.org/wiki/Fanorona

[2] Wikipedia: Minimax - website: http://en.wikipedia.org/wiki/Minimax

[3] STL Containers - C++ Reference - website: http://www.cplusplus.com/reference/stl

[4] GTK+ - website: http://www.gtk.org

[5] Glade - A User Interface Designer - website: http://glade.gnome.org