Universidade do Porto

Faculdade de Engenharia

FEUP

Hex

Relatório Final

Inteligência Artificial

3º ano do Mestrado Integrado em Engenharia Informática e Computação

Elementos do Grupo:

Bruno Nova – 080509109 – ei08109@fe.up.pt

Rolando Pereira – 080509150 – ei08150@fe.up.pt

22 de Maio de 2011

Objectivo

Este trabalho têm como objectivo criar uma implementação, em Java, do jogo Hex.

Esta implementação servirá como um suporte prático para melhorar a aprendizagem de conceitos apresentados nas aulas teóricas da disciplina, nomeadamente as estratégias de pesquisa considerando adversários. Em especial, este trabalho serviu para um estudo do algoritmo Minimax com cortes Alfa-Beta.

Para facilidade de utilização do programa, foi também desenvolvida duas interfaces para o jogo: uma interface em modo de texto e uma interface gráfica utilizando a API Swing do Java.

Para finalizar, foram também adicionadas várias características para melhorar a qualidade do jogo, tais como diferentes modos de jogo, diferentes modos de dificuldade e diferentes tamanhos de tabuleiro.

Descrição

Funcionalidades

O programa desenvolvido neste trabalho contêm várias funcionalidades tais como a escolha do tamanho do tabuleiro no qual irá ser efectuado o jogo.

Existem também três tipos de jogo disponíveis:

- Humano contra Humano
- Humano contra Computador
- Computador contra Computador

É também possível modificar a dificuldade da inteligência artificial, tendo que conta que o único parâmetro diferente entre as várias dificuldades é o nível de profundidade do algoritmo Minimax que é usado. Dito isto, as três dificuldades existentes são:

- Fácil, correspondendo ao algoritmo Minimax com profundidade 2
- Médio, correspondendo ao algoritmo Minimax com profundidade 3
- Difícil, correspondendo ao algoritmo Minimax com profundidade 4

Em termos de código, a variáveis que determinam a profundidade do algoritmo Minimax para cada dificuldade encontram-se no ficheiro $ComputerAI.java$, sendo elas:

```
static private final int EASY_DEPTH = 2;
static private final int MEDIUM_DEPTH = 3;
static private final int HARD_DEPTH = 4;
```

Estrutura do programa

O programa desenvolvido encontra-se dividido em vários módulos, estando estes separados por ficheiros de código Java.

O módulos desenvolvidos foram os seguintes:

- *Hex* – Ponto de entrada do programa. Contêm a função *main* do programa;
- *HexConsole* – Módulo da interface de consola;
- *HexGraphical* – Módulo da interface em módo gráfico;
- *Board* – Módulo do tabuleiro de jogo. Contêm o algoritmo para detectar caminhos no mesmo;
- *ComputerAI* – Módulo da inteligência artificial.

As relações entre estes módulos podem ser vistos na figura 1.

melhorar esta seccao

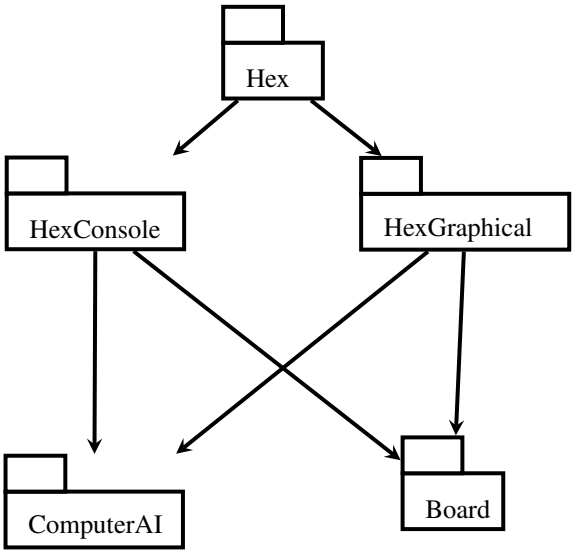

Figura 1: Diagrama dos módulos do programa e as suas relações

Esquemas de Representação de Conhecimento

Descreva os esquemas de Representação de Conhecimento que utilizou no trabalho. Tente, sempre que possível, indicar as vantagens da representação que escolheu em relação a representações alternativas que poderiam ter sido usadas. Refira nesta subsecção detalhes relevantes da implementação

Análise da complexidade dos algoritmos usados

Faça aqui uma análise da complexidade dos algoritmos que usou no trabalho.

Board.checkLineThroughBoard

Devido ao polimorfismo da linguagem Java, existem três funções com o nome checkLineThroughBoard na class Board.

Para ser possível diferenciar entre elas, cada função será numerada:

1. – checkLineThroughBoard(int[][] boardToCheck, int color)
2. – checkLineThroughBoard(int[][] boardToCheck, int y, int x, int color)
3. – checkLineThroughBoard(int[][] boardToCheck, int y, int x, int color, List<int[]> passedPositions)

A função ① encontra-se dividida em duas partes.

O primeiro ciclo *for*, mostrado na listagem 1, serve para verificar em existe pelo menos uma peça da cor pedida em cada linha do tabuleiro. Caso isso não ocorra, não pode haver um caminho que vá do topo do tabuleiro até ao fundo do mesmo.

mudar Listing para Listagem

```
for (int[] boardLine : boardToCheck) {
    boolean existsColoredPiece = false;

    for (int boardPlace : boardLine) {
        if (boardPlace == color) {
            existsColoredPiece = true;
        }
    }

    if (!existsColoredPiece) {
        return false;
    }
}
```

Listing 1: O primeiro ciclo *for* da função ①

Facilmente se mostra que este código tem uma complexidade $O(n^2)$:

$$\begin{aligned} n*(1+n*(1+1)+1+1) = \\ n*(1+2n+2) = \\ n*(2n+3) = \\ 2n^2+3n \rightarrow O(n^2) \end{aligned}$$

O resto da função ①, (ver código da listagem 2), chama a função ② para cada elemento do topo do tabuleiro para tentar descobrir caminhos que começem nesse elemento e consiguem atingir o fundo do tabuleiro.

```
for (int x = 0; x != boardToCheck[0].length; x++) {
    if (boardToCheck[0][x] == color) {
        if (checkLineThroughBoard(boardToCheck, 0, x,
            color)) {
            return true;
        }
    }
}
```

Listing 2: O segundo ciclo *for* da função ①

Observa-se que a complexidade deste código é $O(n * ②)$

A função ② simplesmente cria uma lista vazia (do tipo *LinkedList*) que servirá para armazenar as posições do tabuleiro que a função ③ irá processar, para impedir que a função ③ entre em ciclo infinito.

A função ③ é uma função recursiva. Como tal teremos de começar a análise da sua complexidade pelo seu caso-base, que pode ser visto na listagem 3

```
// Final do passo recursivo: Chegar ao fundo do tabuleiro
if (y == boardToCheck.length-1)
    return true;
```

Listing 3: Caso-base da função ③

Facilmente se vê que esta parte do código tem uma complexidade $O(1)$.

A parte recursiva da função envolve a procura de peças que estejam adjacentes à casa que está a ser analisada que tenham a mesma cor e que não estejam presentes na lista *passedPositions*.

Essa parte recursiva requer o uso de uma função auxiliar *listContainsBoardPlace* uma lista de posições do tabuleiro *list* e uma posição do tabuleiro *boardPlace* e verifica se *boardPlace* está contido em *list*. Esta função tem uma complexidade $O(n)$, pois tem de percorrer a lista *list* apenas uma vez.

O algoritmo recursivo da função ③ (código na listagem 4 simplesmente chama-se a si mesma para as casas que se encontram à esquerda, direita, baixo e diagonal baixo esquerda, desde que essas casas tenham a mesma cor que a da casa actual. Este comportamento pode ser visto na figura 2.

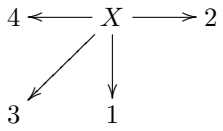

Figura 2: As casas que são visitadas recursivamente pela função ③. Os números indicam a ordem de visita.

```
if (boardToCheck[y+1][x] == color) {
    int[] boardPlace = {y+1, x};

    if (!listContainsBoardPlace(passedPositions,
        boardPlace)) {
        passedPositions.add(boardPlace);

        if (checkLineThroughBoard(boardToCheck,
            boardPlace[0], boardPlace[1], color,
            passedPositions)) {
            return true;
        }
    }
}
if (x != boardToCheck[0].length-1 && boardToCheck[y][x+1]
    == color) {
```

```
    int[] boardPlace = {y, x+1};

    if (!listContainsBoardPlace(passedPositions,
        boardPlace)) {
         passedPositions.add(boardPlace);

         if (checkLineThroughBoard(boardToCheck,
             boardPlace[0], boardPlace[1], color,
             passedPositions)) {
              return true;
         }
    }
}
if (x != 0 && boardToCheck[y+1][x-1] == color) {
    int[] boardPlace = {y+1, x-1};

    if (!listContainsBoardPlace(passedPositions,
        boardPlace)) {
         passedPositions.add(boardPlace);

         if (checkLineThroughBoard(boardToCheck,
             boardPlace[0], boardPlace[1], color,
             passedPositions)) {
              return true;
         }
    }
}
if (x != 0 && boardToCheck[y][x-1] == color) {
    int[] boardPlace = {y, x-1};

    if (!listContainsBoardPlace(passedPositions,
        boardPlace)) {
         passedPositions.add(boardPlace);

         if (checkLineThroughBoard(boardToCheck,
             boardPlace[0], boardPlace[1], color,
             passedPositions)) {
              return true;
         }
    }
}
```

Listing 4: Parte recursiva do algoritmo ③

Na análise do algoritmo recursivo, considera-se que S corresponde ao tamanho do tabuleiro que está a ser analisado e que F corresponde à função ③, que recebe um tuplo (n, m) em que n corresponde a uma posição y do tabuleiro e m corresponde a uma posição x do tabuleiro. F recebe também um conjunto l que contêm os tuplos (n, m) que já foram visitados.

$$
\begin{aligned}
F(S-1,m,l) &= 1 \\
F(n,m,l) &= F(n+1,m,l) \quad \text{sse} \quad (n+1,m) \notin l \\
&\quad + F(n,m+1,l) \quad \text{sse} \quad (n,m+1) \notin l \\
&\quad + F(n+1,m-1,l) \quad \text{sse} \quad (n+1,m-1) \notin l \\
&\quad + F(n,m-1,l) \quad \text{sse} \quad (n,m-1) \notin l \\
F(n,0,l) &= F(n+1,m,l) + F(n,m+1,l) \\
F(n,S-1,l) &= F(n+1,S-1,l) + F(n+1,S-2,l) \\
&\quad + F(n,S-2,l)
\end{aligned}
$$

Ambiente de desenvolvimento

Este trabalho foi realizado na linguagem de programação Java[1], tendo o código sido desenvolvido e testado em máquinas correndo o sistema operativo Linux[2] , usando o compilador *javac* e os ambientes de programação Emacs[6] e Netbeans[7].

e windows?

Foram usados também alguns pacotes extra de "software" que permitiram melhorar a qualidade do código. Estes foram:

O código final tem de ser entregue com Makefile? nesse caso tem que se colocar também aqui o Make

- Sistema de cobertura de código Emma[3]
- Sistema de compilação de código Ant[4]
- Sistema de testes unitários Junit[5]

Avaliação do programa

Descreva a forma de avaliação do trabalho desenvolvido e os testes realizados.

Caso tenha feito simulações coloque nesta secção uma descrição das condições em que foram feitas as simulações e indique os correspondentes resultados obtidos.

Conclusões

Escreva aqui as conclusões que achar devidas. Diga como o programa poderia ser melhorado. Que funcionalidades adicionais deveria ter ou que sofisticações gostaria de ver implementadas caso tivesse tempo para tal. Como poderia aumentar a eficiência do programa, torná-lo mais rápido ou restringir os gastos de memória.

forma de melhorar o programa: melhorar a funcao heuristica do algoritmo minimax, implementar esta [8] versão do algoritmo

Recursos

Indique os recursos usados na realização do trabalho: bibliografia e software.

Referências

Software

[1] Oracle, “Oracle Technology Network for Java Developers”, 2011, <http://www.oracle.com/technetwork/java/index.html>.

[2] linux.

[3] Vlad Roubtsov, “EMMA: a free Java code coverage tool”, 2005, <http://emma.sourceforge.net>.

[4] The Apache Software Foundation, “Apache Ant - Welcome”, 2010, <http://ant.apache.org>.

[5] Object Mentor, “Welcome to JUnit.org! | JUnit.org”, 2011, <http://www.junit.org>.

[6] Free Software Foundation, “Emacs 23.2.1”, <http://www.gnu.org/software/emacs>

[7] Oracle Corporation, “Netbeans 7.0”, <http://netbeans.org>

Internet

[8] Vadim V. Anshelevich, “A hierarchical approach to computer Hex”, in *Elsevier Science B.V*, 2002, visto a 15/05/2011, <http://home.earthlink.net/~vanshel/VAnshelevich-ARTINT.pdf>.

[9] “Basic (strategy guide) - HexWiki”, 2011, visto a 07/04/2011, <http://www.hexwiki.org/index.php?title=Basic_%28strategy_guide%29#The_two-bridge>

Anexos

Manual do utilizador

Coloque aqui um manual (sucinto) de utilização do seu programa.

Exemplo de uma execução

Apresente aqui um exemplo de execução do programa, ilustrado por uma sequência de "printscreen"s relativos a uma sessão de utilização do programa.