UNIVERSIDADE
DE ÉVORA

Relatório
Trabalho III de Inteligência Artificial

Desenvolvimento de um Agente Inteligente para o Jogo Quatro em Linha

Lucas Werle Melz Nº44224

27 de março de 2023

Introdução

O objetivo deste trabalho é implementar variações do algoritmo minimax para criar um agente inteligente capaz de jogar o jogo “quatro em linha” ou “conecta quatro”. O jogo consiste num tabuleiro de 6 linhas e 7 colunas no qual os jogadores podem inserir peças em qualquer das posições do tabuleiro. Aquele que alcançar uma sequência de 4 peças numa linha, coluna ou diagonal é vitorioso. Utilizou-se a linguagem Java para a implementação do agente inteligente. Implementou-se o algoritmo minimax com corte em profundidade e em um segundo momento foi implementada a otimização *alpha-beta pruning*. Essas duas variações do algoritmo foram testadas com uma função de utilidade simples, que retorna 0 para estados não terminais ou empate e 1 ou -1 para casos terminais com vitória, e uma função de avaliação mais complexa, que permite estimar a probabilidade de vitória em estados não terminais.

Considerando que o tabuleiro possui 42 posições a serem preenchidas, a profundidade máxima da árvore de pesquisa no espaço de estados usando o algoritmo minimax é 42. O número de configurações possíveis do tabuleiro é 3 elevado a 42, considerando que cada posição pode estar vazia, ocupada por uma peça branca ou preta. Neste sentido, é computacionalmente inviável desenvolver um agente inteligente para o jogo utilizando o algoritmo minimax sem estabelecer um corte de profundidade.

No código fonte do projeto podem ser encontrados quatro diretórios: *minmax.simple, minmax.alphabeta.simple, minmax* e *minmax.alphabeta*. Os diretórios que terminam em *simple* utilizam uma função de utilidade simples que retorna 1 ou -1 nos casos de vitória e 0 em todos os outros casos. Os outros dois diretórios utilizam uma função de avaliação que retornará entre -4 a 4, considerando possíveis sequências de peças em linhas, colunas ou diagonais. Por exemplo, se for encontrada no tabuleiro uma sequência de peças ‘W . . W’, isto é, duas peças brancas com duas posições vazias entre elas, e se não houver uma outra sequência de quatro posições no tabuleiro com nenhuma peça preta e 2 ou mais peças brancas, a função de avaliação retornará 2. Neste sentido, obter-se-á uma avaliação objetiva de quais configurações são mais vantajosas para o jogador, encontrando a melhor sequência de posições com ausência de peças inimigas e com maior número possível de peças do jogador atual. Os diretórios que possuem *alphabeta* no nome contém implementações do algoritmo minimax que utilizam *alpha-beta pruning.* Para executar quaisquer um dos projetos, basta acessar o diretório em questão e correr o seguinte comando no terminal:

```
javac *.java && java main
```

Desenvolvimento

1. **a) Escolha uma estrutura de dados para representar os estados do jogo.**

O tabuleiro será representado por um array bidimensional de dados do tipo *char*, de forma que posições vazias são representadas por um ponto e posições preenchidas são representadas pela letra 'W' ou 'B' (*white* ou *black*). Para representar um estado, criou-se uma classe chamada *State*, que para além do tabuleiro, guarda também a informação sobre o número de linhas e colunas do tabuleiro e métodos que permitem gerar os estados sucessores, imprimir o tabuleiro, verificar se um estado é terminal ou não e as funções de utilidade/avaliação.

```
import java.util.*;
import java.awt.Point;

class State implements Cloneable {

   int rows, cols;
   char[][] board;

   public State(int n_rows, int n_cols) {

       this.rows = n_rows;
       this.cols = n_cols;
       this.board = new char[n_rows][n_cols];

       // Fill the board with blanks

       for (int i = 0; i < n_rows; i++) {
           Arrays.fill(this.board[i], '.');
       }
   }

   @Override
   public boolean equals(Object obj) {
            // (see source code for implementation details)

   }
```

```
    @Override
    public int hashCode() {
        // (see source code for implementation details)
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // (see source code for implementation details)

    }

    public List<Point> getPossibleMoves() {
        List<Point> actions = new ArrayList<>();
        for (int i = 0; i < this.rows; i++) {
            for (int j = 0; j < this.cols; j++) {
                if (this.board[i][j] == '.') {
                    actions.add(new Point(i, j));
                }
            }
        }
        return actions;
    }

      public State generateSuccessor(char agent, Point action) throws
CloneNotSupportedException {
        State newState = (State) this.clone();
        newState.board[action.x][action.y] = agent;
        return newState;
    }

    public void printBoard() {
        // (see source code for implementation details)
    }

    public boolean isGoal(char agent) {
        // (see source code for implementation details)
    }

    public double utilityFunction(char agent) {
        // (see source code for implementation details)
}
```

1. **b) Defina o predicado terminal (estado) que sucede quando um estado é terminal.**

Na classe *State*, foi definido o método *boolean isGoal (char agent)*, que determina se um estado é vitorioso ou não para o agente dado. Esse método recebeu duas implementações distintas, uma para quando fazemos uso da função de avaliação otimizada e outra para a versão com a função de utilidade simples. Abaixo segue a versão da função *isGoal* com a função de avaliação mais complexa, que deve retornar 4.0 ou -4.0 se um estado for vitorioso. Essa função de avaliação será explicada posteriormente.

```
public boolean isGoal(char agent) {
    double utilityValue = utilityFunction(agent);
    return Math.abs(utilityValue) == 4.0;
}
```

Na outra implementação, a função *isGoal* foi definida da seguinte forma:

```
public boolean isGoal(char agent) {

    String find = String.valueOf(agent).repeat(4);

    // Check rows
    for (char[] row : this.board) {
        if (new String(row).contains(find)) {
            return true;
        }
    }

    // Check columns
    for (int j = 0; j < this.cols; j++) {
        StringBuilder col = new StringBuilder();
        for (int i = 0; i < this.rows; i++) {
            col.append(this.board[i][j]);
        }
        if (col.toString().contains(find)) {
            return true;
        }
    }

    // Check diagonals
    return checkDiagonals(find);
}
```

Em casos de empate, a verificação é feita na função *main* do programa, que contém a lógica de execução das jogadas para cada um dos jogadores. A situação de empate é verificada se a função *state.getPossibleMoves()* retorna um array vazio.

```
            public    static    void    main(String[]    args)    throws
CloneNotSupportedException {
        // . . . (User inputs depth)

        Minimax minimaxAgent = new Minimax(depth);
        State state = new State(6, 7);

        while (true) {
            // White player's move
            Point whiteMove = minimaxAgent.getMove(state, true);
            state = state.generateSuccessor('W', whiteMove);
            state.printBoard();

            if (state.isGoal('W')) {
                System.out.println("White wins!");
                break;
            }

            // Black player's move
            Point blackMove = minimaxAgent.getMove(state, false);
            state = state.generateSuccessor('B', blackMove);
            state.printBoard();

            if (state.isGoal('B')) {
                System.out.println("Black wins!");
                break;
            }

            // Check for draw
            if (state.getPossibleMoves().isEmpty()) {
                System.out.println("It's a draw!");
                break;

            }
        }
    }
```

1. **c) Defina uma função de utilidade que para um estado terminal deve retornar o valor do estado (ex: -1 perde, 0 empata, 1 ganha).**

A função de utilidade e as funções auxiliares (tais como *isGoal*, *checkDiagonals*, etc.) foram definidas como métodos da classe *State*. A implementação de *isGoal* pode ser consultada na página 5 deste relatório. Segue a implementação da função de utilidade:

```
public double utilityFunction() {
        if (this.isGoal('W')) {
            return 1000.0;
        }
        if (this.isGoal('B')) {
            return -1000.0;
        }
        return 0.0;
    }
```

1. **d) Use a implementação da pesquisa minimax dada na aula prática para escolher a melhor jogada num estado. Teste a sua descrição do jogo com vários estados.**

A implementação do algoritmo minimax (com e sem *alpha-beta pruning*) pode ser consultada nos ficheiros *Minimax.java* em cada um dos subdiretórios do projeto. Cada um dos subdiretórios (que possuem diferentes implementações do algoritmo e das funções de utilidade/avaliação conforme explicado na introdução) possui um ficheiro *Tests.java*, que testa o algoritmo com 10 diferentes configurações do tabuleiro. Para executar os testes, basta correr o comando *javac Tests.java && java Tests.*

1. **e) Implemente a pesquisa Alfa-Beta e compare os resultados (tempo e espaço).**

A comparação do desempenho dos diferentes algoritmos será feita através da tabela apresentada na resposta da pergunta **1.h**.

1. **d) Defina uma função de avaliação que estime o valor de cada estado do jogo. Use os dois algoritmos anteriores com corte em profundidade e compare os resultados (tempo e espaço).**

Definiu-se uma função de avaliação que considera todas as possíveis sequências de quatro posições consecutivas em linhas, colunas e diagonais, contando quantas dessas sequências contêm peças do jogador em questão e não contêm peças do oponente. A função retorna um valor que reflete a melhor sequência encontrada, com valores positivos indicando posições favoráveis para o jogador maximizante (peças brancas) e valores negativos indicando

posições favoráveis para o jogador minimizante (peças pretas). A implementação foi feita através de métodos instanciados na classe *State*.

```
public double utilityFunction(char agent) {
        double rowsEval = evaluateRows(agent);
        double colsEval = evaluateColumns(agent);
        double diagEval = evaluateDiagonals(agent);

        return agent == 'W' ?
        Math.max(Math.max(rowsEval, colsEval), diagEval)
        :
        Math.min(Math.min(rowsEval, colsEval), diagEval);
    }


    public double evaluateRows(char agent) {
        double maximumEvaluation = 0;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j <= cols - 4; j++) {
              char[] sequence = Arrays.copyOfRange(board[i], j, j + 4);
              int positionsCurrentPlayer =
                  countOccurrences(sequence, agent);

              int positionsNextPlayer =
                  countOccurrences(sequence, agent == 'W' ? 'B' : 'W');

                if (positionsNextPlayer == 0) {
                   maximumEvaluation =
                   Math.max(maximumEvaluation, positionsCurrentPlayer);
                }
            }
        }
        return agent == 'W' ? maximumEvaluation : -maximumEvaluation;
    }
```

```
public double evaluateColumns(char agent) {
    double maximumEvaluation = 0;

    for (int j = 0; j < cols; j++) {
        for (int i = 0; i <= rows - 4; i++) {
            char[] sequence = new char[4];
            for (int k = 0; k < 4; k++) {
                sequence[k] = board[i + k][j];
            }
            int positionsCurrentPlayer =
              countOccurrences(sequence, agent);
            int positionsNextPlayer =
              countOccurrences(sequence, agent == 'W' ? 'B' : 'W');

            if (positionsNextPlayer == 0) {
               maximumEvaluation =
               Math.max(maximumEvaluation, positionsCurrentPlayer);
            }
        }
    }
    return agent == 'W' ? maximumEvaluation : -maximumEvaluation;
}

public double evaluateDiagonals(char agent) {
    double maximumEvaluation = 0;
    // Evaluate diagonals from top-left to bottom-right
    for (int i = 0; i <= rows - 4; i++) {
        for (int j = 0; j <= cols - 4; j++) {
            char[] sequence = new char[4];
            for (int k = 0; k < 4; k++) {
                sequence[k] = board[i + k][j + k];
            }
            int positionsCurrentPlayer =
              countOccurrences(sequence, agent);
            int positionsNextPlayer =
              countOccurrences(sequence, agent == 'W' ? 'B' : 'W');
```

```
                if (positionsNextPlayer == 0) {
                    maximumEvaluation =
                    Math.max(maximumEvaluation, positionsCurrentPlayer);
                }
            }
        }

        // Evaluate diagonals from top-right to bottom-left
        for (int i = 0; i <= rows - 4; i++) {
            for (int j = 3; j < cols; j++) {
                char[] sequence = new char[4];
                for (int k = 0; k < 4; k++) {
                    sequence[k] = board[i + k][j - k];
                }
                int positionsCurrentPlayer =
                  countOccurrences(sequence, agent);
                int positionsNextPlayer =
                  countOccurrences(sequence, agent == 'W' ? 'B' : 'W');

                if (positionsNextPlayer == 0) {
                    maximumEvaluation =
                    Math.max(maximumEvaluation, positionsCurrentPlayer);
                }
            }
        }
        return agent == 'W' ? maximumEvaluation : -maximumEvaluation;
    }

    public int countOccurrences(char[] row, char target) {
        int count = 0;
        for (int i = 0; i < row.length; i++) {
            if (row[i] == target) {
                count++;
            }
        }
        return count;
    }
```

A comparação do desempenho do algoritmo minimax usando a função de avaliação referida e a função de utilidade simples será feita a seguir no contexto da resposta à questão **1.h**.

1. **g) Implemente um agente inteligente que joga o 4 em linha: 1 - Joga uma peça, atualiza e mostra o tabuleiro. 2 - Lê a jogada do adversário, atualiza e mostra o tabuleiro. Volta a 1 até o jogo terminar.**

Para visualizar o algoritmo minimax a jogar o jogo de forma automática, basta executar a função *main* de qualquer uma das implementações.

1. **h) Apresente uma tabela com o número de nós expandidos para os diferentes estados do jogo (10 no mínimo) com os vários algoritmos.**

Para verificar o desempenho do algoritmo nos casos de teste utilizados para a construção da tabela a seguir, basta executar o programa *Tests* em cada um dos subdiretórios do projeto.

Nós expandidos (profundidade da pesquisa = 5).

Estados	Minimax c/ função de utilidade simples	Minimax alpha-beta c/ função de de utilidade simples	Minimax c/ função de avaliação	Minimax alpha-beta c/ função de avaliação
Tabuleiro 1	440 131	**7 981**	676 620	14 916
Tabuleiro 2	14 843 390	**69 779**	74 832 086	102 787
Tabuleiro 3	396 076	**2 926**	396 076	10 151
Tabuleiro 4	804 050	**4 336**	804 050	**4 336**
Tabuleiro 5	266 645	**9 023**	681 891	10 817
Tabuleiro 6	1 106 821	**5 185**	1 106 821	**5 185**
Tabuleiro 7	1 106 821	**5 185**	1 106 821	**5 185**
Tabuleiro 8	17 783 701	**25 201**	17 783 701	**25 201**
Tabuleiro 9	1 635 352	**6 987**	967 533	7 257
Tabuleiro 10	overflow	**1 212 485**	overflow	1 820 004

Tempo de execução (profundidade da pesquisa = 5).

Estados	Minimax c/ função de utilidade simples	Minimax alpha-beta c/ função de de utilidade simples	Minimax c/ função de avaliação	Minimax alpha-beta c/ função de avaliação
Tabuleiro 1	789 ms	**19 ms**	433 ms	31 ms
Tabuleiro 2	34571 ms	96 ms	38844 ms	**83 ms**
Tabuleiro 3	286 ms	**2 ms**	194 ms	7 ms
Tabuleiro 4	601 ms	**3 ms**	408 ms	4 ms
Tabuleiro 5	289 ms	**6 ms**	352 ms	8 ms
Tabuleiro 6	267 ms	**1 ms**	583 ms	4 ms
Tabuleiro 7	270 ms	**2 ms**	559 ms	4 ms
Tabuleiro 8	11764 ms	**14 ms**	9298 ms	20 ms
Tabuleiro 9	1649 ms	6 ms	508 ms	**4 ms**
Tabuleiro 10	?	2586 ms	?	**977 ms**

Tabuleiros

```
char[][][] boardConfigurations = {


        { // Configuration 1
                { 'W', 'B', 'W', 'W', 'B', 'B', 'W' },
                { 'B', 'W', 'B', 'B', '.', 'W', 'B' },
                { 'W', '.', '.', 'W', 'B', 'W', 'W' },
                { '.', 'B', 'W', '.', 'B', 'W', '.' },
                { '.', 'B', '.', 'W', '.', 'B', '.' },
                { 'W', 'W', '.', 'B', '.', '.', '.' }
         },
```

```
{ // Configuration 2
      { '.', '.', 'W', 'B', '.', '.', '.' },
      { '.', 'B', 'W', 'W', '.', '.', '.' },
      { 'B', 'W', 'W', '.', 'B', '.', '.' },
      { 'W', 'W', 'B', '.', '.', '.', '.' },
      { 'B', '.', '.', '.', '.', '.', '.' },
      { '.', '.', '.', '.', '.', '.', '.' }
},

{ // Configuration 3
      { 'W', '.', 'B', 'W', 'B', '.', 'W' },
      { 'B', 'W', '.', '.', 'B', 'W', 'B' },
      { 'W', 'B', '.', 'W', '.', 'W', '.' },
      { 'B', '.', 'W', '.', 'B', '.', 'W' },
      { '.', 'W', 'B', 'W', '.', 'B', '.' },
      { 'W', '.', 'W', 'B', 'W', '.', 'B' }
 },

 { // Configuration 4
      { '.', 'B', 'W', '.', 'B', 'W', '.' },
      { 'W', '.', 'B', 'W', '.', 'B', 'W' },
      { '.', 'W', '.', 'B', 'W', '.', 'B' },
      { 'B', '.', 'W', '.', 'B', 'W', '.' },
      { '.', 'B', '.', 'W', '.', 'B', 'W' },
      { 'W', '.', 'B', 'W', '.', 'B', '.' }
  },

  { // Configuration 5
      { 'B', 'W', 'B', 'W', 'B', 'W', 'B' },
      { 'W', 'B', 'W', 'B', 'W', 'B', 'W' },
      { '.', '.', '.', '.', '.', '.', '.' },
      { 'W', 'B', 'W', 'B', 'W', 'B', 'W' },
      { '.', '.', '.', '.', '.', '.', '.' },
```

```
        { 'W', 'B', 'W', 'B', 'W', 'B', 'W' }
    },


    { // Configuration 6
        { 'W', 'W', 'W', 'W', '.', '.', '.' },
        { 'B', 'B', 'B', 'B', '.', '.', '.' },
        { 'W', 'W', 'W', 'W', '.', '.', '.' },
        { 'B', 'B', 'B', 'B', '.', '.', '.' },
        { 'W', 'W', 'W', 'W', '.', '.', '.' },
        { 'B', 'B', 'B', 'B', '.', '.', '.' }
    },

    { // Configuration 7
        { '.', '.', '.', 'W', 'W', 'W', 'W' },
        { '.', '.', '.', 'B', 'B', 'B', 'B' },
        { '.', '.', '.', 'W', 'W', 'W', 'W' },
        { '.', '.', '.', 'B', 'B', 'B', 'B' },
        { '.', '.', '.', 'W', 'W', 'W', 'W' },
        { '.', '.', '.', 'B', 'B', 'B', 'B' }
     },

     { // Configuration 8
        { 'B', 'W', '.', '.', '.', '.', '.' },
        { '.', 'B', 'W', '.', '.', '.', '.' },
        { '.', '.', 'B', 'W', '.', '.', '.' },
        { '.', '.', '.', 'B', 'W', '.', '.' },
        { '.', '.', '.', '.', 'B', 'W', '.' },
        { '.', '.', '.', '.', '.', 'B', 'W' }
     },

     { // Configuration 9
        { '.', '.', '.', '.', '.', '.', '.' },
        { 'B', 'W', 'B', 'W', 'B', 'W', 'B' },
        { 'W', '.', '.', 'B', 'W', 'B', 'W' },
        { 'B', 'W', 'B', 'W', 'B', 'W', 'B' },
        { '.', '.', '.', '.', '.', '.', '.' },
```

```
            { 'B', 'W', 'B', 'W', 'B', 'W', 'B' }
        },



        { // Configuration 10
            { '.', '.', '.', '.', '.', '.', '.' },
            { '.', '.', '.', '.', '.', '.', '.' },
            { '.', '.', 'W', 'W', '.', '.', '.' },
            { '.', '.', '.', 'W', 'B', '.', '.' },
            { '.', '.', '.', '.', 'B', '.', '.' },
            { '.', 'B', '.', '.', '.', '.', '.' }
        }
};
```

Conclusão

Em alguns casos, a pesquisa minimax com poda alpha-beta e uma função sofisticada de avaliação teve um desempenho melhor do que a pesquisa minimax com poda alpha-beta e uma função de utilidade simples. A função de avaliação simples, embora mais rápida em alguns casos, não oferece a mesma profundidade de análise e, portanto, não é tão eficaz em prever jogadas favoráveis em estados não terminais. Com uma avaliação mais precisa das posições intermediárias, o algoritmo pode tomar decisões mais informadas sobre quais ramos do jogo explorar.

Devido às estimativas mais precisas e melhor ordenação de movimentos, a função de avaliação sofisticada ajuda a reduzir o número de nós que precisam ser expandidos. Isso é especialmente benéfico em profundidades de pesquisa maiores, onde o número de nós potencialmente exploráveis é muito grande.