A.I. in Nine Men's Morris

[*]Solving Nine Men's Morris game with adversarial search methods implemented in Java (Theme 5 / Group 46)

Bernardo Barbosa
Artificial Intelligence
FEUP
Porto, Portugal
up201503477@fe.up.pt

Duarte Carvalho
Artificial Intelligence
FEUP
Porto, Portugal
up201503661@fe.up.pt

Joao Sa
Artificial Intelligence
FEUP
Porto, Portugal
up201506252@fe.up.pt

***Abstract*—In this paper we describe the problem and our approach to Nine Men's Morris game as a adversarial search problem. We will start by modeling the problem and talk about it's rules and constrains.**

Finally we share and discuss some of the work that has been already been done related to this subject, drawing some conclusions about our future work and development expectations.

***Index Terms*—artificial inteligence, java, adversarial search algorithms, alpha beta pruning**

I. Introduction

The purpose of this paper is to describe how we are going to implement min-max with alphabeta pruning in the Nine Men's Morris game.

The codebase is totally done in Java, initially we thought about doing this project in NodeJS so that we could make a browser game, however soon after we realized doing the project in Java would save us time in the development phase and it has enough performance for this type of workload.

Following this section, there are four sections with different topics.

On the first section, we describe the game itself and the rules.

Next, we formulate the problem to the best of our abilities, the game representation, it's operators and rules, and the utility function(s).

On the third section, we describe source codes that we found on the internet that are related to this game, like game implementations and artificially intelligent agents that have already been implemented which might be useful to our project implementation.

Lastly, we summarise the work that we have done and draw some conclusions about the subject.

II. Problem description

Nine men's morris aka the mill game, is a strategy board game for two players.
The board consists of a grid with twenty-four points. Each player has nine pieces, or "men". The goal is to form 'mills'. A mill is a line of three men alined (horizontally or vertically). A mill allows you to remove an opponent's man from the game. A player wins by reducing the opponent to two pieces, or by leaving him without a legal move.

The game proceeds in three phases:

- Placing - Placing men on vacant points.
- Moving - Moving men to adjacent points.
- (optional phase) Flying - Moving men to any vacant point when a player has been reduced to three men.

Fig. 1. Nine mens morris traditional board.

III. Problem formulation

A. State representation

For a state representation we need:

- board - An integer array containing 24 positions, each corresponding to a certain board position. *
- nmoves - Number of executed moves.
- npieces - An integer array of size 2, each position of the array represents the number of player pieces.
- boardHistory - An array of previous boards in order.
- millCountdown - Counting down the moves without any mill.
- flyingMillCountdown - When both players are left down to three pieces counting down the moves without any mill.

* - We also have a neighbour position matrix, and a possible mills matrix, these are needed due to our simplification of the board.

B. Initial State

The initial state, S0, would be represented as follow:

- Board

```
000000000000000000000000
```

- nmoves - 0
- npieces - [9,9]
- boardhistory - []
- millCountdown - 50
- flyingMillCountdown - 10

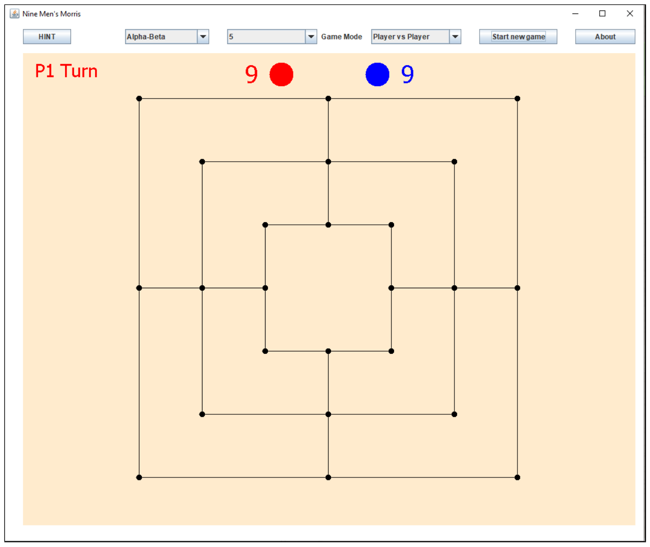

Fig. 2. Initial state representation.

C. Terminal Test

There are multiple ways to end a game:

1) Tiying
   - If there are 50 moves without any mills.
   - 10 moves where both players only have 3 pieces.
   - if the board repeats three times.
2) Victory
   - A player loses if he has no legal moves.
   - A player loses if he is left with only two pieces.

D. Operators

We consider two operators, but the actual implementation is a bit more blurry in it's separation:

- Placing(startPos, takenPos)
- Moving(startPos, targetPos, takenPos)

A operator can have up to three items:

1) A startPos is always required, and in the Placing operator it works as a point where to place the currentPlayer piece, in the Moving Operator it works as a current player piece selection.
2) A targetPos is required if the operator is of the Moving type, it specifies the new position for the selected piece.
3) A takenPos is optionally required in both Operators if the would-be move forms a mill, and is corresponding to the opponent piece to remove.

Pre-Requisites:

1) A startPos for the Placing operator must indicate a open spot.
2) A startPos for the Moving operator must indicate a piece corresponding to the current player.
3) A targetpos validity for the Moving Operator depends on the state of the game, if the current player has more than three pieces, the selected position must be open, and in a neighbouring position to the startPos, if a player has exactly three pieces then he can ”fly” and the selected position must only be open.
4) A takenPos for both the Placing and Moving operator must indicate a opponent piece.

Effects:

1) The Placing operator would place a current player piece in the startPos and if a mill is formed it would remove the opponent piece corresponding to the takenPos.
2) The Moving operator would take the selected piece of the current player, and move it to the target position, and as always if a mill is formed it would remove the opponent piece corresponding to tha takenPos.

E. Utility function

For our utility functions we have the following methods:

- **evaluatePossibleMovingMoves** - evaluates the number of possible moves during the moving phase, this is usefull even during the placing phase otherwise the computer might be inclined to place his pieces in a ”greedy” way (looking for mills) and end up without possible moves later on the game.
  It gives 50 points per possible move during the moving phase.
- **evaluatePiecePlacement** - evaluates the pieces placement, we decided that the more neighbours a cell has the bigger value it has since it could be used to form more mills.
  It gives 50 points per each cell neighbour.
- **evaluateNumberOfPieces** - evaluates the number of pieces for each player that is if a player has had more pieces taken than the opposing player then he will receive a smaller pontuation, this ends up being the same as giving value for forming mills.
  It gives 1000 points per piece.
- ***evaluateGameOver*** - evaluates the end game situation, awarding points for both a draw and a game win.
  It gives 50000 points per game win and half that for a draw.
- **randomComponent** - this method returns a very small (in comparison) value so that the games are always a bit different.
  It gives a random value between -10 and 10.

Then we have two utility functions:

- **fav1** - Should be the more complete utility function, it uses all the previous explained methods.

```
public static int fav1 (GameState gameState) {
    int value=0;

    value += evaluatePossibleMovingMoves(gameState);
    value += evaluatePiecePlacement(gameState);
    value += evaluateNumberOfPieces(gameState);
    value += evaluateGameOver(gameState);
    value += randomComponent();

    return value;
}
```

- **fav2** - For this utility function we decided to remove the value awarded for piece placement, and double the coefficient for the number of pieces, effectivelly awarding 2000 points per piece.

```
public static int fav2 (GameState gameState) {
  int value=0;

  value += evaluatePossibleMovingMoves(gameState);
  value += evaluateNumberOfPieces(gameState) * 2;
  value += evaluateGameOver(gameState);
  value += randomComponent();


  return value;
}
```

IV. GUI - Graphical User Interface

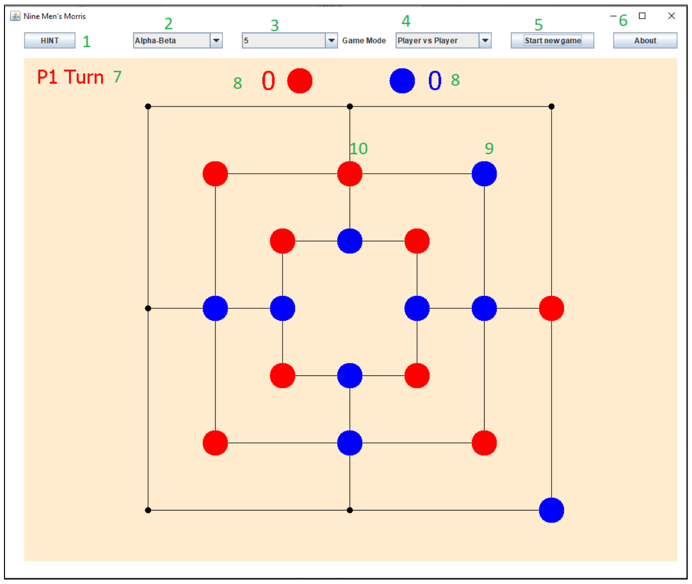

Fig. 3. Our graphical user interface for the game with guide numbers

To provide a better user experience we implemented a GUI with useful buttons and labels, making the gameplay faster, easier and more pleasant to the user.

As you can see every GUI component in the picture is labeled with a green identifier. This numbers are not present in the game, they are just there to facilitate the components description and functionality.

- **1. Hint button** - if the game has already started, if a user presses this button, it will fill a label next to it with a computer generated hint, that supposedly is the best move a player can make.
- **2. Algorithm picker** - Before the game starts, the user must choose of two available algorithms. The user option is the algorithm that A.I. will use to give hints or to play.
- **3. Algorithm depth** - With this combo box, the user can choose the graph depth of the I.A. algorithms. It's basically difficulty level of the Intelligent agent. Depth ranges from 3 to 7, meaning that 3 is the easiest setting and 7 the hardest.
- **4. Game mode selector** - With this component the user can choose the game mode, it has 4 modes available.
- **5. Start game button** - this button will check if the user made the correct game configuration (selected at algorithm, depth and game mode) and start the game.
- **6. About button** - this button will display in a new window relevant information about the project.
- **7. Turn indicator** - this label is constantly changing text and color, in order to facilitate the current player turn identification.
- **8. Available rocks indicator** - this label indicates how many rocks a player still has to place in the board.
- **9. P2 rock** - example blue rock, blue rocks belong to player 2.
- **10. P1 rock** - example red rock, red rocks belong to player 1.

V. Results

- 0 - Draw
- 1 - Player 1 Won
- 2 - Player 2 Won

- aB - alphaBeta
- mM - MiniMax

A. *alphaBeta Algorithm*

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
fav1	fav1	3	3	1	107	8,2s
fav1	fav1	3	3	2	138	7,8s
fav1	fav1	3	3	0	47	7,4s
fav1	fav1	3	3	2	52	7,0s
fav1	fav1	3	3	1	71	7,6s

TABLE I
5 Games - Depth = 3

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
fav1	fav1	5	5	1	107	6:01s
fav1	fav1	5	5	1	61	5:12s
fav1	fav1	5	5	1	47	5:48s
fav1	fav1	5	5	1	61	5:04s
fav1	fav1	5	5	1	49	6:00s

TABLE II
5 Games - Depth = 5

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
fav1	fav2	3	3	1	47	7,8s
fav1	fav2	3	3	1	103	7,5s
fav1	fav2	3	3	2	200	10,1s
fav1	fav2	3	3	1	211	8,3s
fav1	fav2	3	3	2	72	6,2s

TABLE III
5 GAMES - DIFFERENT EVALUATION - DEPTH = 3

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
fav1	fav2	2	4	1	33	23,6s
fav1	fav2	2	3	0	49	6,7s
fav1	fav2	3	2	1	71	6,2s
fav1	fav2	4	2	1	33	37,7s
fav1	fav2	2	2	1	31	1,7s

TABLE IV
5 GAMES - DIFFERENT EVALUATION - VARIABLE DEPTH

B. *MiniMax Algorithm*

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
fav1	fav1	3	3	1	31	20,7s
fav1	fav1	3	3	1	133	26,1s
fav1	fav1	3	3	1	73	20,4s
fav1	fav1	3	3	1	43	21,7s
fav1	fav1	3	3	2	42	20,9s

TABLE V
5 GAMES - DEPTH = 3

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
fav1	fav1	5	5	1	605	02:26:22s

TABLE VI
1 GAME - DEPTH = 5

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
fav1	fav2	3	3	1	117	25,2s
fav1	fav2	3	3	2	64	23,2s
fav1	fav2	3	3	1	39	19,51s
fav1	fav2	3	3	1	113	21,8s
fav1	fav2	3	3	1	205	24,7s

TABLE VII
5 GAMES - DIFFERENT EVALUATION - DEPTH = 3

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
fav1	fav2	2	4	1	43	3:45s
fav1	fav2	2	3	2	40	12,5s
fav1	fav2	3	2	1	31	17,6s
fav1	fav2	4	2	1	29	4:46s
fav1	fav2	2	2	1	35	3,4s

TABLE VIII
5 GAMES - DIFFERENT EVALUATION - VARIABLE DEPTH

C. *alphaBeta Algorithm vs MiniMax Algorithm*

PC 1	PC 2	Depth 1	Depth 2	Result	Moves	Time
aB (fav1)	mM (fav1)	3	3	2	44	14,7s
aB (fav1)	mM (fav2)	3	3	1	71	14,4s
aB (fav2)	mM (fav1)	3	3	2	50	12,5s
aB (fav2)	mM (fav2)	3	3	2	270	20,0s
aB (fav1)	mM (fav1)	4	4	1	51	4:37s
mM (fav1)	aB (fav1)	3	3	1	113	17,5s
mM (fav1)	aB (fav2)	3	3	1	107	16,2s
mM (fav2)	aB (fav1)	3	3	1	807	52,6s
mM (fav2)	aB (fav2)	3	3	1	83	15,5s
mM (fav1)	aB (fav1)	4	4	1	33	6:37s

TABLE IX
5 GAMES - DIFFERENT ALGORITHMS - DIFFERENT EVALUATION - VARIABLE DEPTH

VI. RELATED WORK

As far as previous studies are concerned, we took inspiration from the work done for the 2nd Moodle Activity, and from another work where we implemented the alphaBeta algorithm in LAIG. We also checked GitHub for other implementations of the game.

VII. CONCLUSIONS

We implemented both MiniMax as it's alphabeta variant, the game is ”shipped” with alphabeta and depth 4.
The GUI is located in the gui package under *gui.NineMensMorrisGUI*.
The CLI is located in the cli package under *cli.NineMensMorrisCLI*.
To use the GUI or the CLI simply run the main method corresponding to each class.
Most computer components such as depth or the decision function (normal MiniMax or alphabeta) can be altered in the *utilites.Global* class.

REFERENCES

[1] https://www.wikiwand.com/en/Nine_men%27s_morris
[2] https://github.com/zstoychev/nine-mens-morris/
[3] https://github.com/yadav-rahul/Nine-Mens-Morris
[4] https://github.com/cyrilf/windmill-Morris