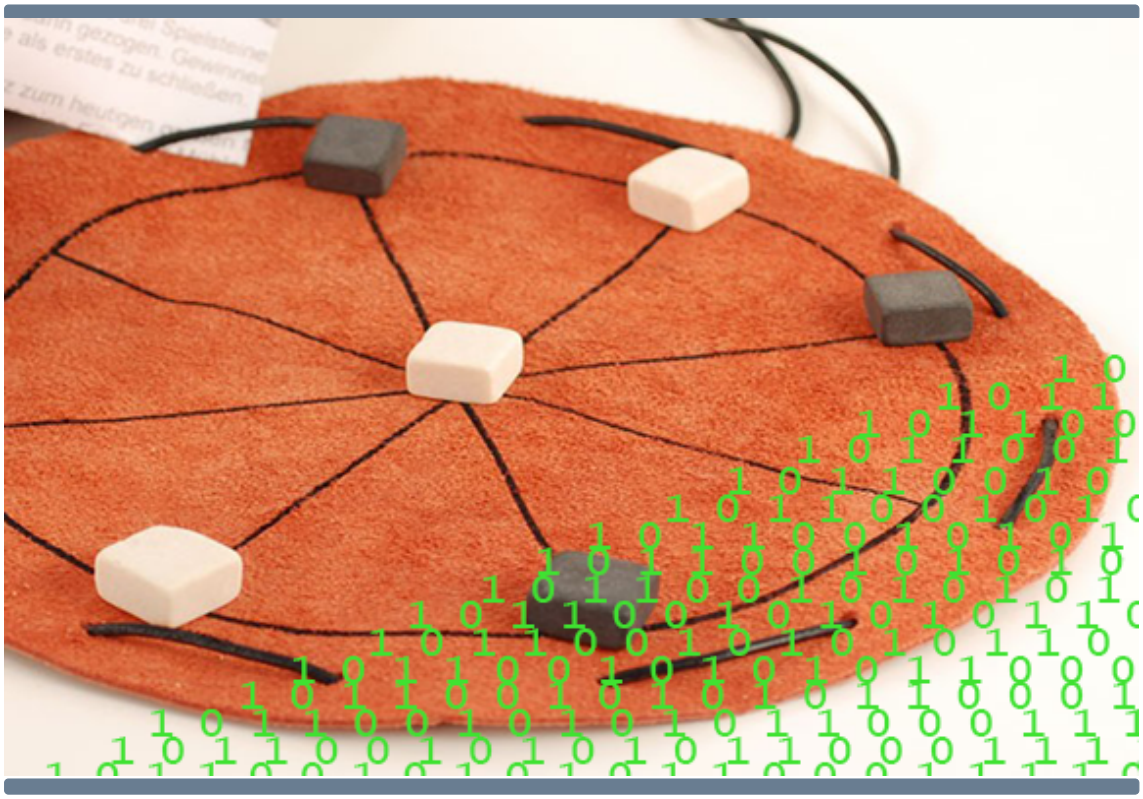
Berner
Fachhochschule



Three men’s morris bot

An exploration of search based bots to play the board game "Three men’s morris"

Semester project, Game Theory (BTI7501p) ’21

Topic of Study:	Computer Science BSc
Authors:	Alexandre Moeri [moera1@bfh.ch] Kevin Amalthas [kevin.amalathas@students.bfh.ch] Markus Joder [jodem1@bfh.ch]
Supervisor:	Prof. Dr. Jürgen Eckerle
Date:	June 25, 2021

Contents

1 Introduction

This report describes our sememster project conducted as part of the Game Theory lecture (BTI7501p) at BFH. The goal is to implement a simple game and several bots using different search based or other approaches. For our purposes we decided to go with the ancient strategy board game three men's morris.

1.1 Three men's morris

Three men's morris is an abstract strategy game played on a three by three board (counting lines) that is similar to tic-tac-toe [4]. The game is also known in the german speaking world as "Römische Mühle" (engl. roman mill) as it is reported to have been a popular game within the roman army [1] even though its origins date back to 1400 BCE [2]. It can further be described as a smaller and simplified version of nine men's morris, also known as "Mühle" in German respectively "Nünistei" in Switzerland, the still today very popular game - in the Region of Bern especially.

1.1.1 Rules

Each player has three pieces. The winner is the first player to align their three pieces on a line drawn on the board. There are 3 horizontal lines, 3 vertical lines and 2 diagonal lines. The board is empty to begin the game 1.1, and players take turns placing their pieces on empty intersections.

Once all pieces are placed (assuming there is no winner by then), play proceeds with each player moving one of their pieces per turn. A piece may not move to any vacant point, but only to any adjacent linked empty position, i.e. from a corner to the middle of an adjacent edge, from the middle of an edge to the center or an adjacent corner, or from the center to the middle of an edge [4].

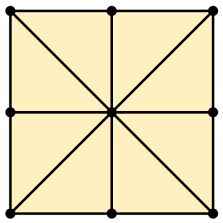


Figure 1.1: Empty three men's morris board [3]

2 Method

2.1 Technology

We took the opportunity to explore some of the newest technologies and thus the game engine and agents were implemented using an early develepor preview of Python 3.10 [1].

2.2 Game

2.2.1 Engine

The project architecture is very simple. ‘game.py‘ implements the game according to the rules mentioned above within two classes:

1. ‘Game‘ which takes two players and let’s them play against each other. The firt player passed will be going first.
2. ‘Board‘ which represents the current board state and has a nice string method.

2.2.2 Drivers

To simplify the running of games two driver scripts were implemented:

1. play.py to start a game or a series of games between two agents
2. tournament.py to start a tournament (several bots playing against eachother)

2.2.2.1 Play

play.py is the main script and helps set up games between players. It can be run with:

py play.py PLAYER0 PLAYER1 [NUMBEROFGAMES]

‘PLAYER0‘/‘PLAYER1‘ must be identifiers for agents that ‘play.py‘ knows of, such as: - ‘human‘ to play through the CLI - ‘random‘ for Rando the random Bot - ‘minimax‘ for minimax algorithm, max depth can be set in ‘play.py‘ as the second parameter of the ‘Minimax()‘ constructor. It has a default value of ‘7‘. - ‘reader+minimax‘ for a minimax player that uses the opening book ‘openings.book‘ and

[1] https://www.python.org/downloads/release/python-3100a7/

falls back to standard minimax when it reaches an unknown position. - ‘reader+random‘ same principle as ‘reader+minimax‘ but falls back to the random bot. - ‘monte‘ for the Monte Carlo search algorithm

‘NUMBEROFGAMES‘ defines how many games will be played. Players will alternate their starting turn. Default value is ‘1‘. ‘play.py‘ will also print whenever a game is started and ended to the terminal, aswell as a score total if more than one game is played.

2.2.2.2 Tournament

‘tournament.py‘ let all bots play against one another and saves the result in ‘tournament.results‘. The participating bots are declared in the constant PARTICIPATING-AGENT-IDENTIFIERS.

It can be run with:

py tournament.py [NUMBEROFGAMES]

‘NUMBEROFGAMES‘ defines how many games will be played per bot encounter.

2.3 Agents

A series of agents were implemented:

1. Human (Command line interface)
2. Random
3. Minimax (with Alpha-Beta pruning)
4. Monte Carlo Tree Search
5. Reader (Opening book)

Those are further described below.

To work with the game engine an agent must implement:

1. A constructor that takes one parameter (id) which will be 0 or 1
2. Fields ‘id‘ (passed to the constructor) and ‘name‘ (A string identifying what type of agent it is, ideally also include the id in the name)
3. A method ‘getNextMove(self, player, board)‘ where:
    - ‘player‘ is the identifier used on the board during this game for that player (either -1 or 1)
    - ‘board‘ is an Object of type Board as defined in ‘game.py‘
    - the method returns a valid move, which can either be:
        a) A tupel ‘(player, (x, y))‘ to set a stone (in the starting phase of the game), where ‘player‘ is the identifier passed to the method and ‘x‘ and ‘y‘ are the coordinates of the the position on the board (0, 1, 2), ‘x‘ beeing the row and ‘y‘ beeing the column

b) A tupel ‘((x1, y1), (x2, y2))‘ to move a stone from x1, y1 to x2, y2 during the second phase of the game.

- It is strongly encouranges to use the boards ‘legalMoves‘ method to get a list of the playable moves

There are no checks in place if the moves returned are acutally legal.

2.3.1 Human

The human agent implemented in human.py provids an interface for a human to play the game and thus an easy way to test the engine as well as other bots.

2.3.2 Random

The random bot was used as a baseline for further exploration and simply always plays a random legal move.

2.3.3 Minimax

minimax.py implements the Minimax algorithm. The maximum depth to be searched can be passed or set as default within the constructor method. It implements some basic Alpha-Beta-Pruning by only following one the first found winning line when one is available.

2.3.4 Monte Carlo Tree Search

For the Monte Carlo tree search, we followed the fundamental principle in implementing the heuristic search algorithm. Each round of evaluation of the best possible move consists of four steps:

- Selection
- Expansion
- Simulation
- Backpropagation

In our implementation, we try to follow the steps mentioned above.

The basic functionality of our algorithm is to play a given number of simulations until a winner is determined. During the simulation phase, a score is kept for every simulation, which defines the move’s efficiency for the AI to win. Afterwards, the best possible action, based on the highest score, is then chosen and returned.

2.3.5 Reader (Opening Book)

During testing of our Minimax bot at different depths we realized that at a depth of 9, the bot consitently played the move (1, (1,1)), indicating that this was a winning move. Analysis has shown that there is a winning strategy for the player playing the first stone.

Thus an opening book was written using (implementet in writeopeningbook.py, written in openings.book) and a wrapper bot reader.py. This bot will wrap around any of the other bots and play according to the opening book, which results in always winning when starting as the first player.

As this renders the evaluation of the other bots unuseful, it was disregarded in the further evaluation.

2.4 Logs

Full game logs, including the board state after each turn will be writting to ‘games.log‘.

Tournament results are written to ‘tournament.results‘

3 Results

3.1 Tournaments

To verify how well each bot performs, we have the different agents compete against each other in a tournament. To get a reference on whether a bot is reasonably configured, we also let a random bot compete in each case, which makes arbitrary moves. It is clear that all bots should win by a wide margin against this random agent. To get meaningful results, we let the bots face each other 1000 times in each encounter.

3.1.1 Participating bots

The participating bots in our torunaments are the bots described in the chapters before (Random, Monte Carlo and Minimax). We use different configurations for Monte Carlo and Minimax and verifiy if they perform better with it or not.

3.1.2 Best guess starting configuration

In a first pass, we tried to obtain a solid base configurations for our bots through empirical observation with a small number of games played. For Minimax we found a maximum depth of 5 (d5) and for Monte Carlo a maximum number of simulations of 1'000 (s1000) as a reasonable start configuration.

We let the bots with these basic configuration compete against each other:

Matchup	Result
Random vs Minimax (d5)	0 : 1000
Random vs Monte (s1000)	298 : 702
Minimax (d5) vs Monte (s1000)	813 : 187

Table 3.1: Results for basic bot configuration

Minimax with a depth 5 wins every game against the random bot, and around 80% of games against the Monte Carlo bot. Monte Carlo loses quite some games against the random bot (around 30%), against Minimax it only wins around 20% of all games.

3.1.3 Minimax with more depth

We increase the max depth for the Minimax algorithm from 5 to 7 and verify if this improves the performance against the other bots.

Matchup	Result
Random vs Minimax (d7)	0 : 1000
Random vs Monte (s1000)	304 : 696
Minimax (d7) vs Monte (s1000)	1000 : 0

Table 3.2: Results for increased Minimax depth

Minimax with depth 7 still wins all games against the random bot. Against Monte Carlo we see a significant performance increase, as it now wins every single game out of 1'000 games.

3.1.4 Higher maximum of simulations for Monte Carlo

We give the Monte Carlo algorithm a maximum number of simulations of 2'000 instead of 1'000. Let's verify if this improves the performance against the other bots.

Matchup	Result
Random vs Minimax (d7)	0 : 1000
Random vs Monte (s2000)	145 : 855
Minimax (d7) vs Monte (s2000)	1000 : 0

Table 3.3: Increased max simulations for Monte Carlo

The higher amount of maximum simulations increases the performance of our Monte Carlo algorithm against the random bot. Monte Carlo wins now around 85% of his games against the random but, compared to the 70% with a maximum of 1000 simulations). Against Minimax, Monte Carlo still cannot win a single game.

We further increase the max number of simulations for Monte Carlo to 5'000, which should improve its performance, and hopefully give Monte Carlo some wins against Minimax:

Now, Monte Carlo clearly defeats the random bot. Altough still losing significantly to Minimax, Monte Carlo is now capable of winning games against Minimax with a depth of 7. The winning rate is around 20%.

Matchup	Result
Random vs Minimax (d7)	0 : 1000
Random vs Monte (s5000)	0 : 1000
Minimax (d7) vs Monte (s5000)	866 : 134

Table 3.4: Further increased max simulations for Monte Carlo

Trying to get even more wins for Monte Carlo we double the maximum number of simulations from 5'000 to 10'0000.

Matchup	Result
Random vs Minimax (d7)	0 : 1000
Random vs Monte (s10000)	0 : 1000
Minimax (d7) vs Monte (s10000)	699 : 301

Table 3.5: Monte Carlo with 10'000 simulations

This further improved the performance of the Monte Carlo alogrithm against the Minimax. It now winns around 30% of all games against Minimax.

3.1.5 Minimax with opening book

Finally, we try to use the opening book for our Minimax algorithm. We use the base configuration (depth of 5 for Minimax and 1000 simulations for Monte) for our bots:

Matchup	Result
Minimax (reader d5) vs Random	1000 : 0
Minimax (reader d5) vs Monte (s1000)	946 : 54

Table 3.6: Minimax with openin books (reader)

Without the opening books strategy, Minimax had a winning rate of around 80%. With help of the opening strategies, Minimax is now winning more games (around 95%) against Monte. This corresponds to an improvement of 15%.

4 Discussion

4.1 Conclusion

Three men's morris was a simple enough game to implement quickly and get acquainted with the different algorithms that were to be studied in this course. Due to the nature of the game, especially it's low complexity, the minimax approach has shown to be the most useful, since at relatively low depth winning sequences can be found, and as seen previously even a winning strategy was found for player one. Minimax is, due to it's simple implementation, relatively slow. It was thus interesting to explore other options and as can be seen in our results the monte carlo approach also renders useful results, at a much faster speed. Lastly, our current assumption is that, when player 1 does not follow the winning strategy there is always potential for an endless game, that will at some point repeat, when both players refuse to play into a loosing line.

4.2 Outlook

There are several potential optimization that could be made to the project, especially the minimax implementation could be improved on by on one side caching the board state, which would be estimated to already dramatically improve the speed, as well a implement a paralleled version of the algorithm, which would be able to take advantage of multiple core machines and thus also render results faster. It is thus remains to be seen whether player 2 has winning strategies if player 1 does not start with the center stone.

4.3 Sources

The entire projects source code can be found at `https://github.com/amoeri/three-mens-morris`.

List of Figures

List of Tables

Bibliography

[1] Claudia-Maria Behling. “Der sog. Rundmühle auf der Spur Zug um Zug zur Neudeutung römischer Radmuster”. In: Wien: Phoibos Verlag, 2014, pp. 63–70.

[2] R. C Bell. “Board and Table Games from Many Civilizations, volume 1”. In: New York City: Dover Publications, 1979, pp. 91–92.

[3] Elembis. *Copied from Nine Men’s Morris.svg and modified., CC BY-SA 3.0*. URL: https://commons.wikimedia.org/w/index.php?curid=1505603 (visited on 05/20/2021).

[4] *Three men’s morris Wikipedia*. URL: https://en.wikipedia.org/wiki/Three_men%27s_morris (visited on 05/20/2021).