Artificial Intelligence

Adversarial Search Method for Line of Actions

Realizado por:

- Fabio Huang up201806829
- Ivo Ribeiro up201307718
- Maria Beirao up201806798

Specification

Board and Pieces:

- Simple checkers board (8x8)
- Each player has 12 pieces on the board (black and white)

Game Objective:

- Connect all your pieces together in a group (including diagonals)

Movements:

- Each turn, the player moves one of his pieces, in a straight line (diagonal included), exactly as many squares as there are pieces of either color anywhere along the line of movement.
- Players may jump over their own pieces
- Players may not jump over their opponent's pieces, but can capture by landing on them

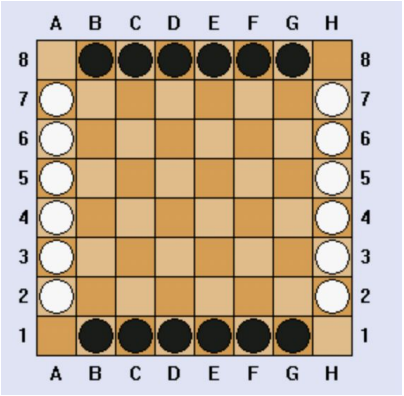

Figure 1 - LOA initial state

State representation

The state will be represented with an 8 x 8 matrix (board[8,8], or in general board[rows, columns]), filled with elements of the following type:

- Black Piece (B);
- White Piece (W);
- Empty Cell (0).

The player's turn to move (turn) will also be represented.

Initial State

The black checkers are placed in two rows along the top and bottom of the board, while the white stones are placed in two rows at the left and right of the board.

```
turn = BLACK                // Black begins
board =
[ [0, B, B, B, B, B, B, 0],
  [W, 0, 0, 0, 0, 0, 0, W],
  [W, 0, 0, 0, 0, 0, 0, W],     // B - Black Piece
  [W, 0, 0, 0, 0, 0, 0, W],     // W - White Piece
  [W, 0, 0, 0, 0, 0, 0, W],     // 0 - Empty Cell
  [W, 0, 0, 0, 0, 0, 0, W],
  [W, 0, 0, 0, 0, 0, 0, W],
  [0, B, B, B, B, B, B, 0] ]
```

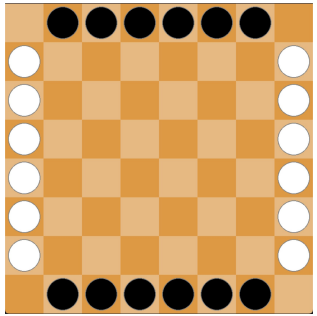

Figure 2 - LOA on Pygame

Objective Test

The game ends when all pieces of either player are connected together in a group, where diagonals are also considered to be connected.

- If a player is reduced to a single piece, that is a win for the captured player, because a single piece counts as a connected group.
- Another way to win is if the other player has no possible moves (all pieces blocked on all sides).
- If a move made by a player creates a win for both the victory is given to the player who made the move, and we control this, by checking the win condition after a move, first for that player.

```
// returns 1:Black Wins; 2:White Wins; -1:Game continues
def check_gameover():  // checks if the current state creates an ending condition
        …
```

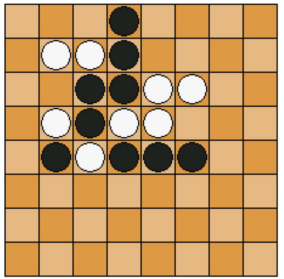

Figure 3 - Typical finished game, where black wins

The Approach

Programming Language:

- Python (pygame library used)

Development Environment:

- Visual Studio Code, Pycharm
- Python Interpreter 3.10.4
- Git

Data Structure:

- Board, Game, Pieces represented as Classes
- The Board Class contains a list that has the information of the pieces positions (board[8][8])

File Structure:

- The current file structure a main class (where the game is called) and inside the folder game contains a “classes” module with all the classes (board, game, piece) including a file constants.py, which contains constants used throughout the other files (pieces colors, window size, board size…)

The Approach

Heuristics and Evaluation Functions

- We created 3 different levels:
  - Easy (search depth = 1)
  - Medium (search depth = 2)
  - Hard (search depth = 3)
- All of them have the same heuristic with 3 evaluation functions

Evaluation Functions	**Points**
Concentration	-10*Average distance from centre of mass (max 70 points)
Centralisation	Sum of the points for every piece of the same color (max 200 points)
Largest Connected Branch	1000 per piece connected (max 12000 points)

Figure 4 - Evaluation Functions Table

-80	-25	-20	-20	-20	-20	-25	-80
-25	10	10	10	10	10	10	-25
-20	10	25	25	25	25	10	-20
-20	10	25	50	50	25	10	-20
-20	10	25	50	50	25	10	-20
-20	10	25	25	25	25	10	-20
-25	10	10	10	10	10	10	-25
-80	-25	-20	-20	-20	-20	-25	-80

Figure 5 - Centralisation Evaluation Score Table

Operators

- get_valid_moves(self, piece) - Finds out all the possible moves that a selected piece can make (passed by parameter)
- move(self, piece, row, col) - Moves a piece to a new square (identified by row and column)

Algorithm implemented

- We implemented the algorithms:
  - Minimax
  - Minimax with Alpha-Beta Pruning

- For both of the algorithms:
  - Code can be found inside the file ‘/minimax/algorithm.py’
  - Customizable depth size inside the file ‘/game/constants.py’

***Note:** inside ‘game/constants.py’ file, a flag called ALPHA_BETA is used to decide which algorithm to use

```
# Flag to decide if alpha-beta pruning should be used in minimax algorithm
ALPHA_BETA = True

# BOT LEVELS, the number represents the minimax algorithm DEPTH
EASY_LEVEL = 2
MEDIUM_LEVEL = 3
HARD_LEVEL = 4
```

Figure 6 - Screenshot of flags inside ‘constants.py’

Experimental Results - I

Time consumed per move	Minimax	Minimax With Alpha-Beta Cuts
Easy	0,26	0,22
Medium	7,58	2,05
Hard	397,96	38,71

Figure 7 - Execution time per move of each algorithm for different levels

Nodes visited per move	Minimax	Minimax With Alpha-Beta Cuts
Easy	1 280	403
Medium	36 776	5 767
Hard	1 967 967	74 403

Figure 8 - Nodes visited per move of each algorithm for different levels

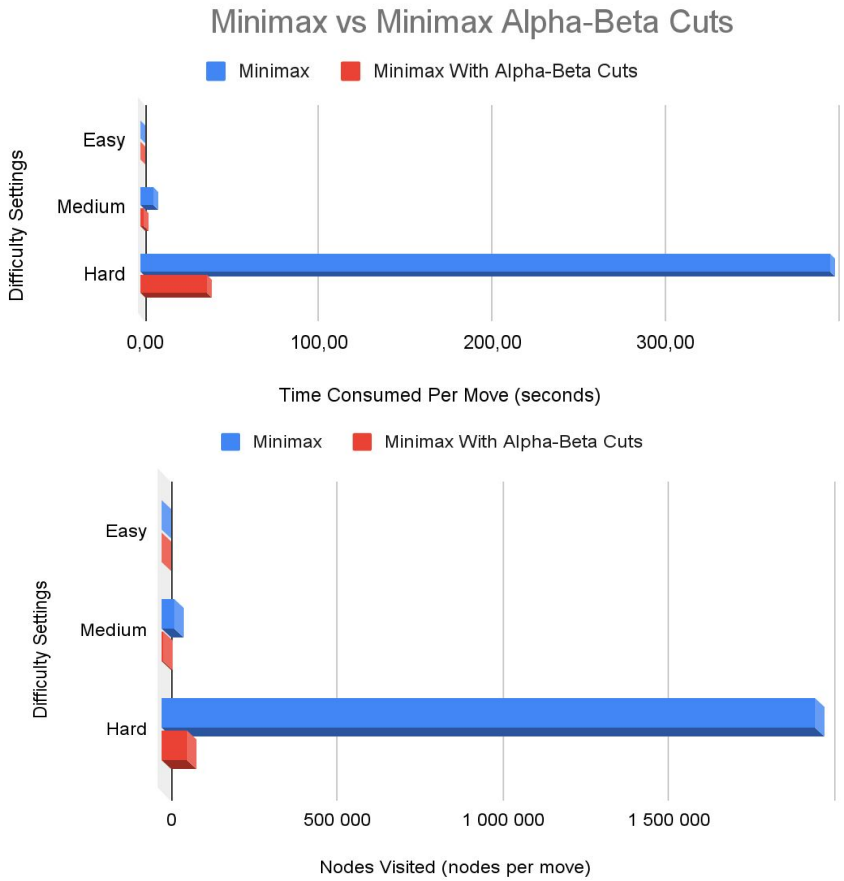

Figure 9 - Graphs comparing between Minimax and Minimax Alpha-beta cuts

Experimental Results - II

Games between AI of different level

AI Level	Search Depth Size
Easy	2
Medium	3
Hard	4

Figure 10 - AI level configuration table

Easy and Easy

Easy level AI (Black)	Easy level AI (White)
37 Wins	13 Wins

Figure 11 - Ex. Results Easy AI (B) vs Easy AI (W)

Easy and Medium

Easy level AI (Black)	Medium level AI (White)
4 Wins	16 Wins

Figure 12 - Ex. Results Easy AI (B) vs Medium AI (W)

Medium level AI (Black)	Easy level AI (White)
20 Wins	0 Wins

Figure 13 - Ex. Results Medium AI (B) vs Easy AI (W)

Medium and Medium

Easy level AI (Black)	Medium level AI (White)
12 Wins	8 Wins

Figure 14 - Ex. Results Medium AI (B) vs Medium AI (W)

***Note:** Hard level AI experiment were not carried out, due to being very time consuming

Conclusions

- During games between bots with same level of difficulty
  - Black pieces tend to have a higher win rate, due to the advantage gained by being the first to play;
  - As the search depth increases, we would expect this difference in win rate to be lower;

- The choice of a heuristic is important, in our project only 3 different evaluation functions were used for the heuristic.
  - Other evaluators that could be added to improve decision making, such as quad evaluator and block evaluator (some famous heuristics used to build a LOA AI) [5];
  - It does not affect the quality of the results as much as the search depth used in the Minimax algorithm;

- Using Alpha-beta pruning
  - Significant performance boost in terms of time consumption and number of nodes visited was achieved, specially in larger depth sizes;

- By modelling the Lines of Actions as a search problem
  - We were able to develop an AI which can make plays with a certain level of intelligence
  - Since the maximum depth used in the minimax algorithm was 4 (hard difficult level), the bot was still easily beaten, even by a newbie player, due to having a large branching factor;
  - Increasing the search depth to about 8, we should expect a much more challenging game (however this would not be practical with the setup that we have, would be very time consuming);

Bibliography and References:

[1] **Algorithms Explained – minimax and alpha-beta pruning**
https://www.youtube.com/watch?v=l-hh51ncgDI

[2] **Lines of Action**
https://www.boardspace.net/loa/english/index.html

[3] **THE QUAD HEURISTIC IN LINES OF ACTION**
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.3549&rep=rep1&type=pdf

[4] **An Evaluation Function for Lines of Action**
https://www.researchgate.net/publication/220717032_An_Evaluation_Function_for_Lines_of_Action/download

[5] **Analysis and Implementation of Lines of Action**
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.566.5504&rep=rep1&type=pdf

[6] **ryangmolina/lines-of-action-minimax**
https://github.com/ryangmolina/lines-of-action-minimax