Andantino Engine
Intelligent Search & Games

Chinmay Rao

October 2019

1 Introduction

Andantino is a two-player abstract strategy game played on a hexagonal board made of hexagonal tiles. The standard version of the game starts with a black piece in the centre of the board followed by the second player's turn to place a white piece adjacent to it. The game then proceeds with each player taking turns placing their piece adjacent to at least two other pieces. The game is won by the player who either ends up with five consecutive pieces in a row or encircles at least one of the other player's pieces. This report describes the construction and working of an Andantino game engine including the implementation of a game-playing bot based on the alpha-beta framework.

2 Code Organisation

The Andantino game engine is implemented completely in Python 3 and can be sub-divided into a number of crucial components responsible for handling different aspects of the game. Figure 1 illustrates the code organisation.

2.1 Game Board

The *game_board* file contains the classes and functions needed to render the hexagonal Andantino board the size of which is set to 10. For the given size the board consists of a total 271 hexagonal cells/tiles. The coordinate system used internally is $[x, y]$ representing the "columns" and rows of the board respectively in order to utilize the memory-efficient 2-D *Numpy* array used for board state representation.

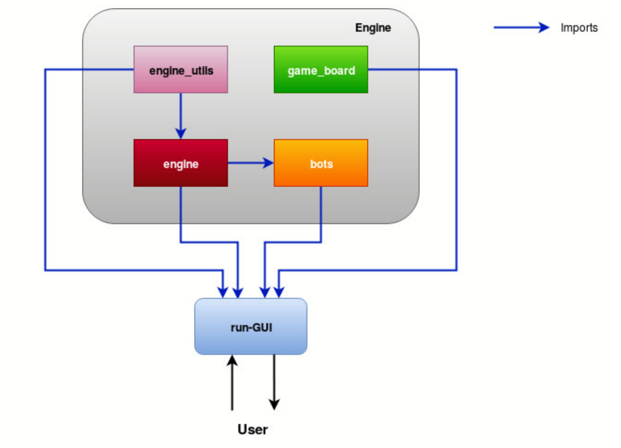


Figure 1: Structure of the code

2.2 Engine

The *engine* file defines the core aspects of the game which can be summarised as follows:

1. Global variables- Includes important game variables for representing the board state, information about cell neighbourhood, game steps, player colours, move history, etc.
2. Helper functions- Some crucial and/or frequently used utility functions.
3. Win conditions: Used for detecting the 5-in-a-row and encircling cases that define a win.
4. Miscellaneous gameplay functions- Used for non-GUI gameplay and testing.

2.3 Engine Utilities

The *engine_utils* file contains some bulky and less frequently used functions.

2.4 Bots

The *bots* file contains the *AndantinoBot* class that defines the game-playing bot as well as the implementation of multiple alpha-beta variants and the heuristic evaluation functions.

2.5 GUI Front-end

The *run-GUI* file renders an interactive game console for a Human v/s Bot match. The miscellaneous file *run-BvB* is used only for testing and isn't needed for the GUI-based gameplay.

3 Game Components

This section elaborates on the main aspects of the game engine.

3.1 Players and Colours

By default, the human player/user is *Player-1* and is represented by the player ID 1, while the computer/bot is *Player-2* having the player ID 2. Although the player IDs are fixed, the user can choose among the two colours (black and white) at the beginning of the game. The player colours are stored as a tuple in the global variable *player_ colors* where the first and second entries correspond to colours for the user and the bot respectively.

3.2 Board state representation

The board state (i.e. the position) is represented as a 20x20 Numpy array of 8-bit integers. This required using the x, y coordinate system to address the tiles of the board and is illustrated in figure 2. Figure 3 shows the board representation. The empty cells/tiles of the board are marked as 0, user's $(Player-1)$ pieces as 1, bot's $(Player-2)$ pieces as 2 and out-of-the-board (illegal) cells as -9. In this example, the user has chosen colour white making the board have the bot's piece in the center at the beginning of the game which is expressed by a value of 2 (bot's player ID) in the cell $[10, 10]$. The board representation is stored in the global variable $BOARD\_STATE$ of the *engine* script.

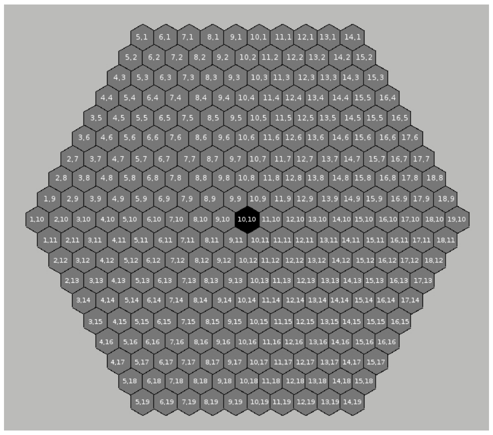
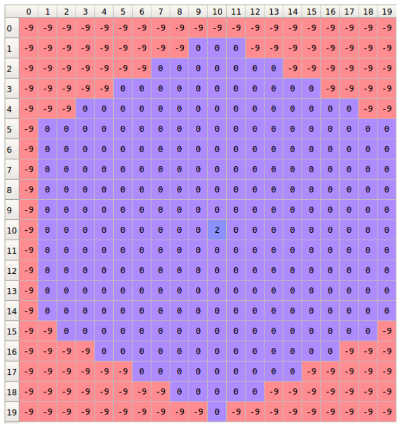


Figure 2: Game board with x,y coordinates



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9
1	-9	-9	-9	-9	-9	-9	-9	-9	-9	0	0	0	-9	-9	-9	-9	-9	-9	-9	-9
2	-9	-9	-9	-9	-9	-9	-9	0	0	0	0	0	0	0	-9	-9	-9	-9	-9	-9
3	-9	-9	-9	-9	-9	0	0	0	0	0	0	0	0	0	0	0	-9	-9	-9	-9
4	-9	-9	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-9	-9
5	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	-9	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0
11	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	-9	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-9
16	-9	-9	-9	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	-9	-9	-9
17	-9	-9	-9	-9	-9	-9	0	0	0	0	0	0	0	0	0	-9	-9	-9	-9	-9
18	-9	-9	-9	-9	-9	-9	-9	-9	0	0	0	0	0	-9	-9	-9	-9	-9	-9	-9
19	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	0	-9	-9	-9	-9	-9	-9	-9	-9	-9

Figure 3: Board representation

3.3 Rules

3.3.1 Legal moves

In the standard version of the game, a player must place their piece adjacent to at least two other pieces except while playing the first white piece which can be placed adjacent to the central black piece. The engine implements this by generating a list of allowed moves for each turn using the function *getAvailableMoves()*. The user's move is processed only when it's one of the moves in the list.

3.3.2 Winning conditions

In the engine, the encircling situation is considered the first winning condition and is implemented using the *floodfill* algorithm. To check for this, a copy of the board state is passed to the *_floodfill()* function which "floods" it with a high value leaving out only the pieces of the player who made the last move. If the opponent's pieces are detected, they would certainly be inside an encirclement of the former player's pieces which prevented them from being washed away indicating a win for the former player.
The 5-in-a-row situation is considered as the second winning condition. The engine checks for this by trying to detect a straight line of 5 consecutive similar pieces along each of the three straight axes $(0^o, +60^o, -60^o)$ of the hexagonal board.

3.4 Search algorithm

A number of search algorithms, ranging from the basic Minimax to enhanced Alpha-Beta have been implemented in the engine, although only one can be used by the bot per game. The best performing of them all is the Negamax based Alpha-Beta with Transposition Table and Iterative Deepening (with move ordering) as enhancements, identified within the engine as *'ab-negamax-TT-ID'*. Its implementation is discussed in the following lines.

3.4.1 Transposition Table

The transposition table (TT) is represented as the nested Python dictionary *TRANSPOSITION_ TABLE* of the following form:

$$\begin{aligned} TT = \{&'2495' : \{'value' : 3,' flag' :' Exact',' depth' : 4\},\\ &'14374' : \{'value' : -2,' flag' :' UpperBound',' depth' : 4\},\\ &'90' : \{'value' : -4,' flag' :' UpperBound',' depth' : 3\}\} \end{aligned}$$

For a given board state, the state hash is calculated using Zobrist Hashing with a 3-D array of random numbers *RANDOM_ MATRIX* of dimensions 20x20x3 each of whose elements is a 16-bit integer. The following three functions facilitate the implementation of the TT-

1. *buildInitialTT()*: Initialises the *RANDOM_ MATRIX* array and the *TRANSPOSITION_ TABLE* dictionary at the beginning of the game.

2. *pushIntoTT()*: This function is used to write the board state information into the TT as an entry and is called within the search function *ABNegamaxTTPlayer()*.

3. *retrieveTTEntry()*: This function is used to access the board state information in the TT in case this position had been encountered earlier in the search. Like the previous function, it also invoked in *ABNegamaxTTPlayer()*.

3.4.2 Iterative Deepening

The search function *ABNegamaxTTPlayer()* implements the Alpha-Beta search with TT. It can either be used as it is using the configuration keyword *'ab-negamax-TT'* or within an iterative framework of progressive deepening using the keyword *'ab-negamax-TT-ID'* in the run script. In the latter case that is iterating deepening, the Alpha-Beta search function is invoked within a *for* loop iterating over depth up to the specified maximum depth (*ID_ max_ depth*). A simple Move Ordering scheme is used which sorts the moves based on evaluation scores before the next ID iteration. Each time the search returns a move, it also returns a list of scores of the root's children (scores that were back-propagated from the leaf nodes). In the next iteration before the search is initiated again, the child nodes of the root are sorted from best to worse evaluation scores. This makes the Alpha-Beta search to first investigate the children with high scores further deeper and then those with lower scores.

3.5 Evaluation function

The evaluation function implemented in this engine calculates the utility score of a board state from the perspective of the root player (i.e. the bot). So a win for the bot has a value 10^5 (ideally infinity) while a win for the human player has -10^5 (ideally -infinity) Two different evaluation functions have been implemented and tested in the engine, although only one can be used by the bot per game. Within the *AndantinoBot* class, an evaluation function selector is used to set the function to be used for the game. Each of them is elaborated in the following sub-sections.

3.5.1 Function 1

The evaluation function *Fn1*, implemented as *_ evalFn1()*, uses a very simple criterion to evaluate the board state. If the player to move at the leaf node of the tree can win by making a single move, that board state is favourable for them. Features based on win conditions are defined for each player and depending on the player, weights are assigned to each feature. The final score is then calculated as a linear weighted sum of the those features.

$$\begin{aligned} score =& w_1 \times bot\_line\_score + w_2 \times bot\_surr\_score \\ &+ w_3 \times en\_line\_score + w_4 \times en\_surr\_score \end{aligned}$$

There are four features defined - the human player's and the bot's "line scores" and "surround scores". Line scores signify the number of potential ways the player-to-move can land a 5-in-a-row arrangement with their next move, while the surround scores represent the ways of encircling the opponent with a single move. The weights w_1 and w_2 that correspond to the bot's scores are set to a high value (10) and weights w_3 and w_4 corresponding to the user's scores are set to a low negative value (-1) when the player to move (at the leaf) is the bot. When it's the human player's turn at the leaf, the weight magnitudes are interchanged $(w_1, w_2 = 1; w_3, w_4 = -10)$.

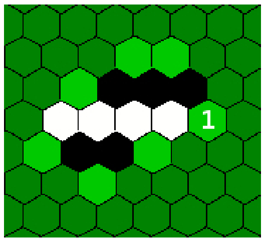


Figure 4: White (the bot) can win in one move

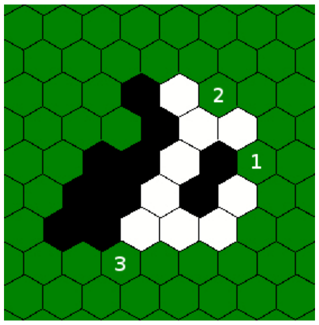


Figure 5: A more complex arrangement

Consider the scenario illustrated in figure 4. Here, the white pieces belong to the computer (bot) and black belong to the human player. The player to make the next move is the bot. The weights w_1, w_2, w_3 and w_4 will bear the values 10, 10, -1 and -1 respectively. The bot can get a 5-in-a-row and win if it places its next piece in the cell marked by 1. Therefore the line score of the bot will be 10, while the line score of the human player is 0 (default value). The total score will hence be 100.

In a more complex arrangement as shown in figure 5 where the bot has three potential win moves (marked by 1, 2 and 3), its surround score and line score are incremented to 10 and 20 respectively resulting in a total score of 300.

3.5.2 Function 2

A drawback of *Fn1* is that it only quantifies how favourable the board state is to the bot (player at the leaf). The second evaluation function *Fn2* tries to quantify the favourability of the board for the opponent (i.e. the human player) in addition to that for the bot, and is implemented as the Python function *_evalFn2()*. It not only checks whether the player at the leaf node can win making their next move but also accounts for the winning chances of the opponent (i.e. the risk) if the leaf player doesn't go for the winning move. The score is calculated similar to *Fn1* as:

$$\begin{aligned} score =& w_1 \times bot\_line\_score + w_2 \times bot\_surr\_score \\ & + w_3 \times en\_line\_score + w_4 \times en\_surr\_score + random\_term \end{aligned}$$

The weights w_1, w_2, w_3 and w_4 bear the values 20, 20, -1 and -1 when the player-to-play (at leaf) is the bot and 1, 1, -20 and -20 when it is the human player. The line and surrounding scores, like earlier, are calculated for the player at the leaf following a simulated move. In addition, the same move is simulated for the opponent player and the new position is then used to calculate the opponent's scores. Also, a small random term, which is a uniformly distributed random variable over the integer range [-5,5], is included in the score to guide the search towards the sub-tree with more options on good states.

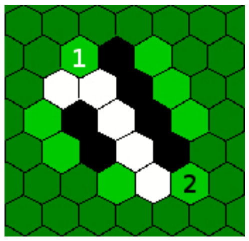


Figure 6: White can win in one move, but is also at a risk of losing to Black if it doesn't play that move

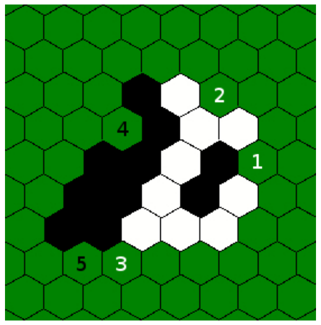


Figure 7: A more complex arrangement

Consider the same example from earlier where the bot (player White in this case) is at a leaf node. In figure 4 the bot could win by placing a piece in cell 1. Similarly in figure 6, it can win the same way, however additionally this position is also potentially favourable for the human player because of the cell 2. Assuming the value of the random term is zero, *Fn2* calculates the evaluation score as:

$$\begin{aligned} & bot\_line\_score = 10; bot\_surr\_score = 0 \\ & en\_line\_score = 10; en\_surr\_score = 0 \\ & w_1, w_2, w_3, w_4 = 20, 20, -1, -1 \\ & score = 20 \times 10 + 20 \times 0 + (-1) \times 10 + (-1) \times 0 = 190 \end{aligned}$$

Fn1 would score the board states in both figure 4 and figure 6 the same value of 100 when clearly the latter is more beneficial for the opponent than the former. *Fn2*, taking this into account, scores them as 200 and 190 respectively.

Figure 7 illustrates the same complex arrangement as earlier. The bot can win by placing a piece either in cell 1 encircling the enemy or by achieving a

5-in-a-row in the cells 2 or 3. At the same time, this board state is also favourable for the human player as they could potentially win (if they were to play) by completing 5-in-a-row in cell 3 or cell 4. The score is hence calculated as:

$$bot\_line\_score = 20; bot\_surr\_score = 10$$
$$en\_line\_score = 20; en\_surr\_score = 0$$
$$w_1, w_2, w_3, w_4 = 20, 20, -1, -1$$
$$score = 20 \times 20 + 20 \times 10 + (-1) \times 20 + (-1) \times 0 = 580$$

Fn2 has been found to perform better than *Fn1* since it captures the risk factor of a board state for the player at the leaf node and so is used in the engine by default. In addition to the board state evaluation, if a leaf node is a terminal node, it's evaluation score is augmented by a value of 10^5 (ideally infinity) if it's the bot's win or by -10^5 (ideally -infinity) if it's the human player's win.

4 Performance

To evaluate the performance of various search algorithms, two AndantinoBot objects were created and made to play with each other both using the same algorithm per game. Three specific algorithms - Alpha-Beta (*ab-minimax*), Alpha-Beta with Transposition Table (*ab-negamax-TT*) and Alpha-Beta with Transposition Table and Iterative Deepening(+ Move Ordering) (*ab-negamax-TT-ID*) - were tested and the performance data is visualised in figure 8.

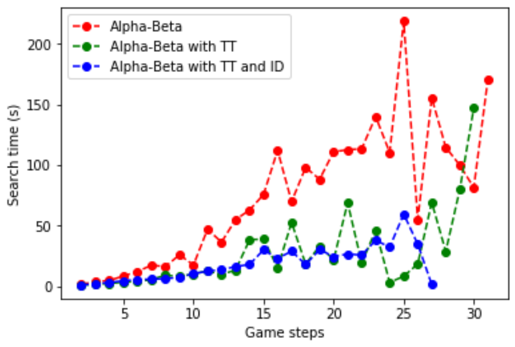
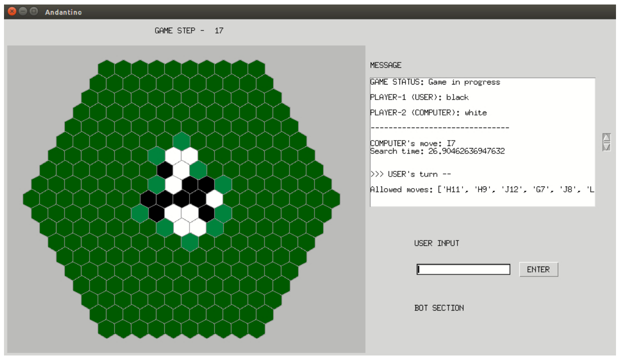


Figure 8: Performance of different Alpha-Beta variants

The basic Alpha-Beta (depth=3) performs the lowest with a consistently high search time. When enhanced with transposition tables, the performance is significantly improved which is highlighted by the green plot. On using iterative deepening with simple move ordering (max_depth=3), the performance is yet improved, though slightly.

5 Game-play

The *run-GUI* file, when executed, loads the start screen of the game with which the user can interact to begin the the game play. Figure 9 illustrates the game GUI.



```
GAME STATUS: Game in progress
PLAYER-1 (USER): black
PLAYER-2 (COMPUTER): white
------------------------------
COMPUTER's move: I7
Search time: 26.90462636947632

>>> USER's turn --
Allowed moves: ['H11', 'H9', 'J12', 'G7', 'J8', 'L
```

Figure 9: Game interface

The game window contains the board, game step information, a message box and the user and the bot sections. Move coordinates have to be manually entered into the entry bar by the user. Figure 10 shows the board coordinates and this map is included in the game file.

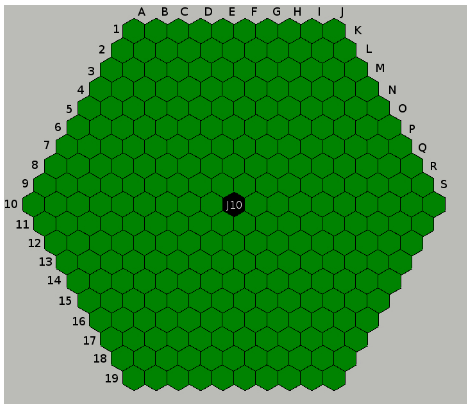


Figure 10: Board coordinates

When its the bot turn to play, the *Initiate search* button which appears in the *Bot Section* needs to be clicked to begin the tree search.

6 Conclusion

This report described the crucial components of an Andantino Engine that implements the Alpha-Beta search framework and programmed in Python 3. Performances of three Alpha-Beta variants were compared and it was found that the use of transposition table and iterative deepening (with move ordering) significantly improved the speed of the search. The overall search time is high due to some code inefficiency as well as due to the nature of Python itself. Improvement in speed can be achieved by further optimising the code and using parallel computation.