Maastricht University

Department of Data Science and Knowledge Engineering

Intelligent Search and Games
(2019-2020-100-KEN4123)

Final project – Andantino Assignment

October 2019

Tomasz Stańczyk
(i6209867)

1. Introduction

This report summarizes the work done over the realization of the Final Project assignment for the Intelligent Search and Games course. The main objective was to develop the game of Andantino [1] and implement searching algorithms based on alpha-beta framework which would be playing intelligently against their opponent.

This report is divided into the following sections. Section 2 describes how to launch (run) the game and how to play it. Section 3 presents the general overview of the key elements of the game implementation. Section 4 discusses the designed and deployed evaluation functions, whereas Section 5 describes all the searching algorithms implemented as well as further enhancements. Finally, Section 6 discusses the tests performed over the implemented searching techniques and presents the relevant results.

2. How to use the program

2.1. Running the game

The program has been developed in Python, version 3.7.1. Its executable version was generated using PyInstaller [2].

The .exe file occupies size of approximately 231 MB, which, which makes it troublesome to be uploaded via the Student Portal (EleUM). Therefore, the following instructions are provided below in case the file could not be uploaded.

In order to run the program, Python environment needs to be installed. Furthermore, two external libraries have been used during the development: Pygame (version 1.9.6) [3] and func-timeout (version 4.3.5) [4]. They can be installed e.g. using the Python's pip installer [5], by executing the following instructions in the console window:

pip install pygame

pip install func-timeout

Subsequently, the program can be run by executing the following instruction in the console window:

python game.py

Note that the current location of the console must set to the location of both files (game.py and boardclasses.py), which in turn can be accomplished by running the following statement:

cd "location/of/the/files"

Having started the program, the game window as presented in Figure 1 appears, whereas the console expects first input from the user, as visible in Figure 2.

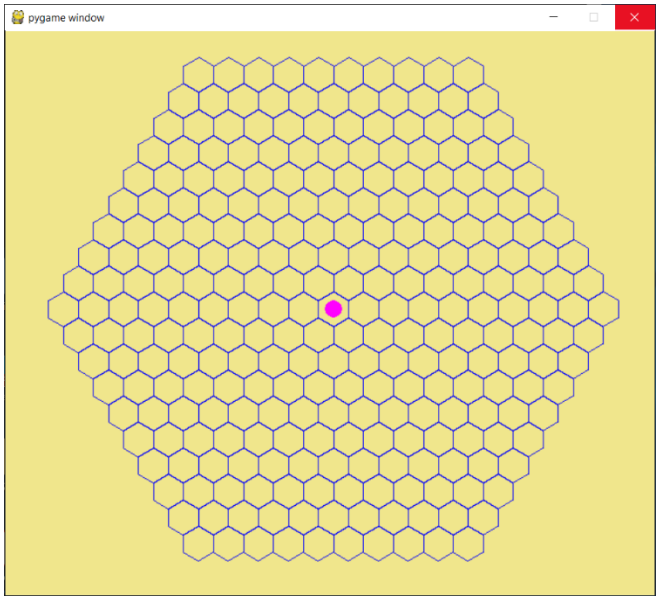

Figure 1 - Initial game window after the program has been started

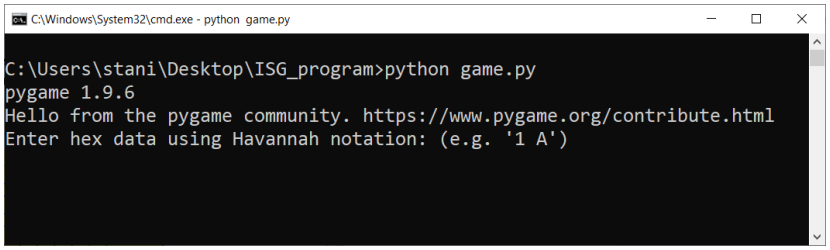

Figure 2 - The state of the command window after the program has been started

2.2. Playing the game

The game can be played via the console window. The user is expected to enter his next move as a string input, following the Havannah notation [6] in the following order: a number from 1 to 19, exactly one space, a letter from 'A' to 'S' (upper or lower case), e.g.:

10 j

Furthermore, only certain moves are allowed per round, which are indicated as pink pawns on the board. All the move attempts from the user which do not meet requirements of the expected format input or which would result in an invalid move, will be rejected. In such a case, user will be prompted to enter another move, until a valid move can be made.

It is important to mention that the game window presenting the current state the board has only illustrative purposes of the program. It is thus not possible to interact with this window. If the user does click on the window though, it will seem to be blocked and/or frozen. Nevertheless, it does not mean that the program as a whole is blocked or crashed. Namely, after entering any input via the command window, the game window will return to its regular appearance and the game can be normally continued. Moreover, all the instructions and messages are also communicated via the console window.

After the game is over, the screen background color changes into green (the victory of the user) or into red (the victory of computer). Appropriate message is then displayed in the command window as well. The program needs to be closed manually by user, which can be accomplished by closing the game window or by pressing the [CTRL + C] combination over the command window.

3. General Overview of the Game Implementation

3.1. Hexagon Class

The fundamental building block of the whole game structure is the Hexagon class. Each Hexagon object stores information about its coordinates (x and y), which enable to draw it appropriately on the boards, as well as its row and column numbers, which enable its identification on the board. The last two properties are principally stored and processed as integer numbers, but when the game interacts with the user, they are appropriately transformed from and to Havannah notation. The whole playable board consists of 271 considered and drawn hexagons (since, the board size of 10 by 10 is assumed for this game), stored in a relevant, singleton list.

3.2. BoardState Class

A single state of the game is represented using the BoardState class. This class contains multiple properties which contribute to the overall flow of the game. Each BoardState object stores information about current hexagons occupied by player 1 (black) and player 2 (white) separately. On the visual board, theses hexagons are marked with black and white pawns (circles) respectively. Furthermore, valid moves are also stored. I.e., the hexagons which could be selected at the current state of the game to make a move leading to the next state. These are determined by the adjacency rules of the game, as described in [0], depending on the current round of the game. It is worth mentioning that valid moves should not be mistaken with valid hexagons (aforementioned as the considered and drawn hexagons). Whereas the former reference corresponds to the possible moves depending on the current game state, the latter refers to all the (playable) hexagons considered throughout the game.

Another important part of every BoardState object is the reference to last move hexagon, so the hexagon, executing last move of which resulted in bringing the game to this state. Moreover, current player (player 1 or player 2) information is also stored, which not only points to the player of whom the next turn (move) is, but helps determine the player of the last move as well. Finally, the terminal node property indicates whether the state storing it is a game over state. Its Boolean value is determined the relevant parts of the evaluation function which will be described in the next section.

Additionally, amongst the others, the BoardState contains two crucial functions. First is the neighbor getter function get_hexagon_neighbours(…) which returns a list of neighboring hexagons, given the reference hexagon. Second one is the move making function make_move(…), which given the row and the column of the next move hexagon, creates the new (board) state resulting from executing such a move (provided, that it is a valid move with respect the current game state).

4. Evaluation function

The evaluation function helps determine the usefulness of a tree node (i.e. a game state), when it is being considered by a search function. What is more, it is applied to determine whether the node which has just been created is a terminal node (game over state). The evaluation function designed and developed for the purposes of this game has two main parts: checking the line score and checking if an enclosing has occurred. The former part checks the number of the pawns of the same color forming a straight line on the board. The higher the number of such in a row, the higher the evaluation value, with five pawn straight line indicating a win of the relevant player. The latter part checks if at least one pawn of a player has been fully enclosed by the pawns of the other, opposite player. These two parts are described in more detail in the following subsections.

4.1. Checking for a Straight Line

After a move has been made, the corresponding last move hexagon is checked if it has just contributed to making a (longer) straight line. Three lines are considered: horizontal line, ascending line (which could be seen as a line going

from bottom left toward the upper right direction) and descending line (from upper left to bottom right). For each of these lines separately the following strategy is applied.

A counter with initial value of 1 is created (since there is already 1 pawn) and the check firstly goes through the left part of a considered line and increments the counter if the inspected hexagon is occupied by the player of the last move. If a value of 5 is reached, then the victory of that player is communicated and the counter score is returned. However, if the next hexagon of the check is not occupied by the player of the last move or if there is no next hexagon, because the border of the board has been reached, then the counter score gathered so far is kept and the check starts exploring the right part of the same line, starting from the last move hexagon. Analogously, if a value of 5 is reached here, the victory is communicated and the counter score is returned. Otherwise, just the aggregated value of the counter is returned. Although the whole process is performed for each of the three potential mentioned lines separately, once a line of length 5 has been detected, the consecutive line options are not checked further for a potential win.

When it is checked if the last move has just contributed to the end of the game, only values of 5 are further considered and communicated. When evaluating the game state within a search function though, all the three line options (horizontal, ascending and descending are considered) and the maximum score with respect to this state is returned. Such evaluation is combined together with evaluation coming from the enclosing condition algorithm. It will be described more in detail in the next subsection.

4.2. Checking for an Enclosing

For this winning condition the algorithm represented by the following pseudocode has been implemented:

1. Set currently_considered hexagon as the last move hexagon

2. Initialize empty open_list and closed_list

3. While True:

4. Initialize empty can_see_border_list and considered_neighbours lists

5. Get neigbours of the currently_considered hexagon

6. For each hexagon in neighbours list:

7. If hexagon is of the same player as the last move hexagon, then:

8. Continue (the whole iteration, so go to the next hexagon from the list)

9. If hexagon can see the border from at least one side, then:

10. Add hexagon to can_see_border_list

11. Continue

12. If hexagon is already in closed_list:

13. Continue

14. Add hexagon to considered_neighbours

15. From considered_neighbours, remove each hexagon, such that at least one of its neighbours can see the border from at least one side (using can_see_border_list)

16. Add all remaining considered_neighbours to open_list

17. Add currently_considered hexagon to closed_list

18. If open_list is empty, then:

19. Break

20. Set currently_considered hexagon as first element from open_list, remove that element from open_list

21. Remove first element of closed_list (it is a hexagon occupied by the player of the last move hexagon)

22. If at least one pawn of the player opposite to the one of the last move hexagon has been enclosed (meaning that at least one hexagon occupied by the player to be enclosed in present in closed_list), then:

23. Return True

24. Else:

25. Return False

The algorithm above checks whether at least one hexagon occupied by the player different than the one from the last move hexagon has been fully enclosed by the hexagons occupied by the player of the last move hexagon. E.g., if the last move hexagon belongs to player 1 (black), then the algorithm checks whether at least one white pawn has been fully enclosed by the black pawns.

An important aspect of this algorithm is the part referenced in line 9 (underlined), checking if the considered hexagon can see the border from at least one side of all directions: left, upper left, upper right, right, bottom right and bottom left. For this algorithm, it was assumed that a hexagon can see the border from the given direction if there are no pawns of the player from the last move standing on the way of the relevant straight line from the considered hexagon to the border. Since all hexagons occupied by pawns of the last move player are rejected beforehand (line 7 and 8), only the hexagons occupied by the different player or not occupied at all are considered there, in line 9.

The algorithm checking for an enclosing is used not only to possibly communicate the victory of the last move player, but also to contribute to the evaluation value of a state inspected inside the evaluation function during the run of a search function. Namely, it is put together with the algorithm checking for a straight line (as described in the last paragraph of the previous subsection). After the evaluation value has been determined by the checking for a straight line algorithm, the checking for an enclosing algorithm is used to refine that value. If the enclosing for the considered (evaluated) state is detected, additional 5 points are added to the value. Furthermore, since enclosing (as well as a straight line) can only be made by putting the pawns of the same player next to each other, the evaluation value of a state which results from placing a pawn not adjacently to any other pawns from the same player (color) is decreased (meaning penalized) by one point.

5. Implemented Search Techniques

For the sake of this project, several search alpha-beta framework algorithms have been implemented. Their pseudocodes and/or descriptions are presented in the following subsections. For these searching algorithms, a considered node corresponds to a specific board state of the game

5.1. Move Ordering

Before coming to the actual searching algorithms, it is worth mentioning, that for most of them, the move ordering technique has been implemented. Namely, the children of the currently considered state are sorted based on their value received from the evaluation function, from the best to the worst value respectively. Therefore, the chance of finding the node with the best value first is increased. This concept is also present in pseudocodes displayed in the next subsections. It has been applied to AlphaBeta, AlphaBeta NegaMax and PVS.

5.2. MiniMax

The pseudocode based on which the Minimax has been implemented is as follows:

```
1. minimax(board_state, depth, player_type):
2.     If board_state.terminal_node or depth == 0, then:
3.         Return board_state.evaluate_state()
4.     Generate children based on the current board_state
5.     If player == MAX_TYPE, then:
6.         score = -∞
7.         For each child in children:
8.             value = minimax(child, depth-1, MIN_TYPE)
9.             score = max(score, value)
10.    Else:
11.        score = ∞
12.        For each child in children:
13.            value = minimax(child, depth-1, MAX_TYPE)
14.            score = min(score, value)
15.    Return Score
```

5.3. AlphaBeta

The following psudocode corresponds to the implemented AlphaBeta algorithm (move ordering included):

```
1. alpha_beta(board_state, depth, alpha, beta, player_type):
2.     If board_state.terminal_node or depth == 0, then:
3.         Return board_state.evaluate_state()
4.     Generate children based on the current board_state
5.     If player_type == MAX_TYPE, then:
6.         Sort children based on their evaluation value, from the highest, to the lowest
7.         score = -∞
8.         For each child in children:
9.             value = alpha_beta(child, depth - 1, alpha, beta, MIN_TYPE)
10.            score = max(score, value)
11.            alpha = max(alpha, score)
12.            If alpha >= beta, then:
```

```
13.                         Break
14.             Return score
15.     Else:
16.             Sort children based on their evaluation value, from the lowest, to the highest
17.             score = ∞
18.             For each child in children:
19.                     value = alpha_beta(child, depth - 1, alpha, beta, MAX_TYPE)
20.                     score = min(score, value)
21.                     beta = min(beta, score)
22.                     If alpha >= beta, then:
23.                             Break
24.             Return score
```

5.4. AlphaBeta NegaMax

The pseudocode below represents the implemented algorithm of AlphaBeta with the NegaMax approach (move ordering included):

```
1. alpha_beta_negamax(board_state, depth, alpha, beta):
2.     If board_state.terminal_node or depth == 0, then:
3.             Return -board_state.evaluate_state()
4.     Generate children based on the current board_state
5.     Sort children based on their evaluation value, from the highest, to the lowest
6.     score = -∞
7.     For each child in children:
8.             value = -alpha_beta_negamax(child, depth - 1, -beta, -alpha)
9.             If value > score, then:
10.                    score = value
11.            If score > alpha, then:
12.                    alpha = score
13.            If score >= beta, then:
14.                    Break
15.    Return score
```

5.5. Personal Variation Search (PVS)/NegaScout

The pseudocode based on which the PVS algorithm has been implemented is presented below. Since this algorithms takes a great advantage from a move ordering, this technique has also been incorporated here.

```
1. pvs(board_state, depth, alpha, beta):
2.     If board_state.terminal_node or depth == 0, then:
3.         Return board_state.evaluate_state()
4.     Generate children based on the current board_state
5.     Sort children based on their evaluation value, from the highest, to the lowest
6.     For each child in children:
7.         If first child, then:
8.             score = -pvs(child, depth-1, -beta, -alpha)
9.             return score
10.        Else:
11.            score = -pvs(child, depth-1, -alpha-1, -alpha)
12.            If  alpha < score and score < beta, then:
13.                score = -pvs(child, depth-1, -beta, -score
14.        alpha = max(alpha, score)
15.        If alpha >= beta, then:
16.            break
17.    Return alpha
```

5.6. Iterative Deepening

As a further enhancement, Iterative Deepening has been implemented. It was deployed with AlphaBeta and PVS. The pseudocode is presented below.

```
perform_iterative_deepening(boardState):
1.    Set time_threshold      //in seconds, 3 for PVS
2.    Set max_depth           //20 for PVS
3.    Set time_out to False
4.    Set depth to 1
5.    Generate children based on the current board_state
6.    Sort children based on their evaluation value, from the highest, to the lowest
```

7. *While True:*

8. *best_score = -∞*

9. *For each child in children:*

10. *Call pvs(child, depth, -∞, ∞) on a separate thread with a timeout of time_threshold*

11. *If execution from 10. aborted by timeout, then:*

12. *time_out = False*

13. *break //exit the for loop*

14. *Else:*

15. *Set score to the return value of the call from 10.*

16. *Set current_time_passed to the time of the execution of the call from 10.*

17. *threshold = threshold – current_time_passed*

18. *If score > best_score, then:*

19. *best_score = score*

20. *best_next_state = potential_statae*

21. *If time_out, then:*

22. *break //exit the while loop*

23. *Make a new move based on the best_next_state and save the new state to boardStateHolder*

24. *depth = depth + 2 // odd depths only*

25. *If depth > max_depth, then:*

26. *Break*

27. *Return boardStateHolder*

Itertive Deepening deployed with AlphaBeta would use exactly the same code with the only difference at line 10:

Call AlphaBeta(child, depth, -∞, ∞, MAX_TYPE) on a separate thread with a timeout of time_threshold.

6. Tests and Results

Several tests have been performed, results of which are presented in the subsections below. All the tests presented in this section have been performed for the same sequence of user input: "10 J", "9 J", "9 I", "8 J".

6.1. Time Elapsing Measurements

Firstly, each of the implemented searching algorithms has been called with depth set to 3. Table 1 shows the time elapsed (in seconds) for making the move by each of the algorithm after 1,3,5 and 7 moves makes in total (first move always belonged to the user).

DEPTH 3	MiniMax	AlphaBeta	AlphaBeta NegaMax	PVS
after 1st move	0.330051	0.331139	0.455735	0.115920
after 3rd move	0.785621	0.585668	0.753866	0.098925
after 5th move	2.781000	1.971544	1.701300	0.194925
after 7th move	5.587028	2.931860	2.856786	0.272829

Table 1 – Time elapsing for depth 3

Analogous tests have been performed for depth 5. The results are presented in Table 2.

DEPTH 5	MiniMax	AlphaBeta	AlphaBeta NegaMax	PVS
after 1st move	10.323606	2.668387	11.847068	0.245247
after 3rd move	39.368713	6.393835	6.817475	0.194909
after 5th move	183.019375	29.576717	22.753526	0.334821
after 7th move	347.197967	44.522248	43.537603	0.312819

Table 2 – Time elapsing for depth 5

Figures 3 and 4 present comparison of time elapsed for searching after the seventh move has been made for depth 3 and 5 respectively.

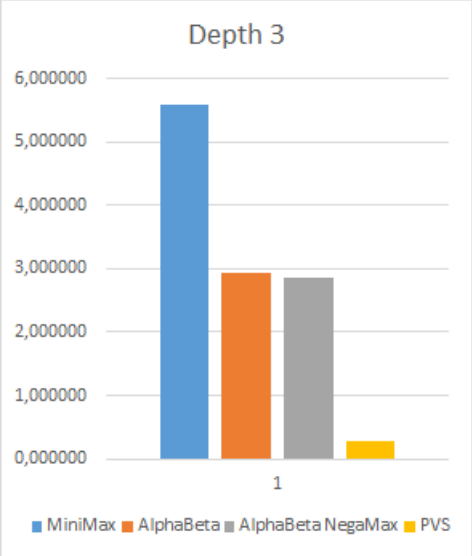

Figure 3 – Time elapsed for searching after seventh move (in seconds), depth 3

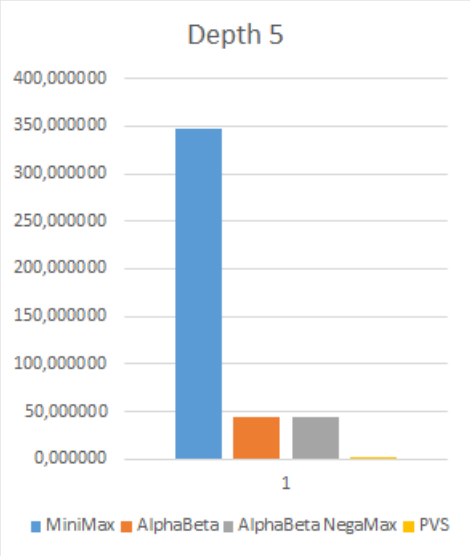

Figure 4 – Time elapsed for searching after seventh move (in seconds), depth 5

As it can be observed in Figures 3-4 and analyzed from Tables 1-2, the time elapsed for searching is comparable for AlphaBeta and AlphaBeta Negamax. However, for Principal Variation Search (PVS) this time is immensely smaller. On the other hand, Minimax needed enormously more time to finish. Based on the Tables 1-2, in the discussed tests for depth 3, PVS was over 20 times faster than Minimax and about 11 times faster than AlphaBeta and AlphaBeta

NegaMax. For depth 5, PVS was about 1110 times faster than Minimax and about 141 times faster than AlphaBeta and AlphaBeta NegaMax.

6.2. Iterative Deepening Depth Measurements

Next series of tests involved the Iterative Deepening approach. It was measured how deep the algorithms can go before the time out. Timeout threshold has been set to 3 seconds, whereas max depth was set to 20. Table 3 and 4 present time elapsed per depth and indicate the depth reached before the timeout by AlphaBeta and PVS respectively.

AlphaBeta (**depth**)	after 1 move (**3**)	after 3 moves (**3**)	after 5 moves (**3**)	after 7 moves (**1**)
max time depth 1	0.008982	0.016005	0.035968	0.033958
max time depth 3	0.089935	0.255208	0.677339	TIMEOUT
	TIMEOUT	TIMEOUT	TIMEOUT	

Table 3 – Iterative Deepening depth measurements for AlphaBeta

PVS (**depth**)	after 1 move (**15**)	after 3 moves (**19+**)	after 5 moves (**15**)	after 7 moves (**19+**)
max time depth 1	0.006983	0.0129923	0.021972	0.027985
max time depth 3	0.034971	0.0409920	0.060935	0.063962
max time depth 5	0.046986	0.0679755	0.076956	0.078937
max time depth 7	0.118963	0.1049706	0.124956	0.070945
max time depth 9	0.115956	0.0909829	0.142950	0.070976
max time depth 11	0.079035	0.0890080	0.141972	0.070959
max time depth 13	0.118034	0.0900294	0.141934	0.080936
max time depth 15	0.118920	0.0899305	0.142935	0.070964
max time depth 17	TIMEOUT	0.0892171	TIMEOUT	0.071975
max time depth 19		0.0899300		0.070958
		(no timeout)		(no timeout)

Table 4 - Iterative Deepening depth measurements for PVC

As it can be observed from Tables 3-4, PVS could go much deeper than AlphaBeta. During the tests performed, the former was reaching depths of at least 15, sometimes being stopped due to the max depth threshold. The latter on the other hand could not reach more that depth of 3, also being stopped after depth 1 in the testing environment described above.

References:

[1] http://www.di.fc.ul.pt/~jpn/gv/andantino.htm

[2] https://pyinstaller.readthedocs.io/en/stable/

[3] https://www.pygame.org/news

[4] https://pypi.org/project/func-timeout/

[5] https://pypi.org/project/pip/

[6] http://www.iggamecenter.com/info/en/havannah.html