Alpha-Beta AI approach to Andantino board game

Intelligent Search & Games project

Guillermo Quintana Pelayo*
AI Master Student
Maastricht University
Netherlands
g.quintanapelayo@student.maastrichtuniversity.nl

October 27, 2019

Abstract

The use of AI in video games is well extended and constantly growing, either for making enemies behaviours or with the aim to develop artificial intelligences capable of playing video games, this project focuses on the first approach. It consists in the creation of a Game Engine for a two-player board game called Andantino, and the implementation of several AI approaches to see which one performs better. The program is based on the alpha-beta framework, commonly used for machine playing in games like Chess, Go, etc. The work belongs to an individual project of the Intelligent Search & Games course for the AI Master program of the Maastricht University. The final implementation was presented in a competition among the students of the mentioned course.

***K*eywords** AI · Andantino · Unity 3D · Alpha Beta

1 Introduction

Andantino is a board game made by David Smith in 1995 [1]. The aim of this project is to implement several 1vs1 AI algorithms and see which one performs better in this game. As a secondary objective, the best AI implemented should be capable of playing a full game against a human or another AI in less than 10 minutes (taking only into account the "thinking" time).

2 Game Engine implementation

This section contains the process followed during the implementation of Andantino and the used tools. The chosen platform to implement the game is Unity 3D and C#. It's worth to mention that since Unity is a full Game Engine one could think that this would be reflected on the efficiency for the search time, and this is technically true, so in order to avoid this, all resources are forced to go to

*Personal web portfolio: https://guilleqp.github.io/.

the main AI thread; this means that during exploration time, everything that is not necessary for the AI search in the game is stopped, graphics, particles, sounds, etc. Unity was selected for this project due to previous experience with this tool and for the facility to implement a game user interface that turns to be very helpful for debugging this kind of game AI.

The hexagonal board is constructed in a 10x10 grid, the original game is meant to be infinite but this restriction is helpful for an easier implementation. The black player always starts the game with a stone in the center hexagon of the board (figure 1).

In the original approach the board has an axis represented by numbers and another one by letters. But in this implementation, the grid has x and y coordinates, the $(0, 0)$ is located on the bottom left corner of the grid. The y is incremented by 1 going right; and x when going upwards. This means that the center coordinates of the grid are $x = 9, y = 9$. In the code this board is represented as a 2-dimensional array of *Hexagon* objects called $grid$, because we're working with an hexagonal board, the corners in this array will be *null* objects.

Sometimes it's useful to know which Hexagons are next to one, for example when seeing if playing a move there is possible. In order to find this "neighbours" of a specific hexagon given it's coordinates this are some examples of the needed instructions (this can be seen in the $Hexagon$ class in the submitted code):

$$Left \Rightarrow grid[x, y - 1]$$
$$Right \Rightarrow grid[x, y + 1]$$

$$Up_left \Rightarrow \begin{cases} x < 9, grid[x + 1, y] \\ x \geq 9, grid[x + 1, y - 1] \end{cases}$$

...

There are three possible play options that have been implemented:

- **Human vs human**. In this case the game starts straight forward and the first black hexagon is already filled in the center tile. Then each player plays normally, after the second turn, blue hexagons indicate the possible moves that can be played. The game only allows the human player to click on this blue hexagons. The turns are switched automatically.

- **Human vs AI**. For this option the Human player needs to specify which color wants to play with and also the AI that wants to play against. As an option you can also indicate the search depth that you want for the algorithm, if you don't specify it then there are some default hard coded values for each one. It's not recommended to play with more than depth 7 (see section 4).

- **AI vs AI**. Here two *dropdowns* are presented to the user to select both AIs that you want to play against each other. When the game starts the AI will play every turn switching the player and the algorithm. There's also the option to simulate several games between two approaches and get all the matches information (see section 4).

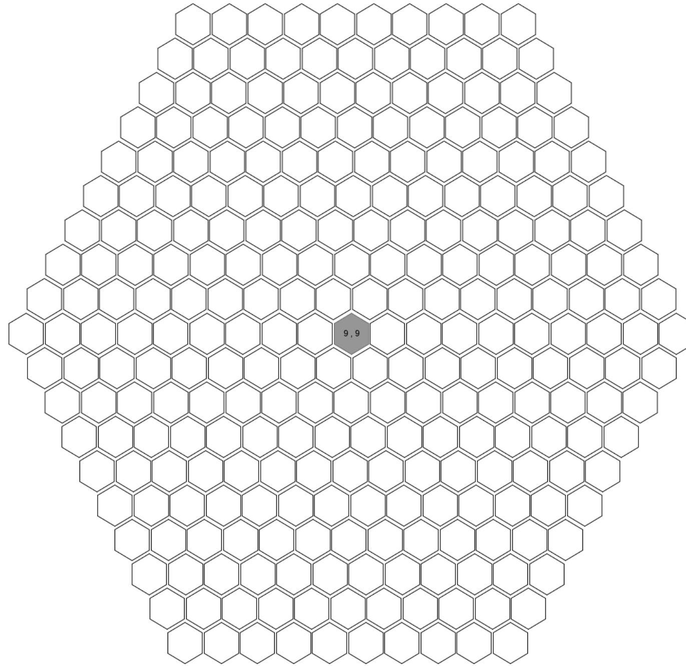

Figure 1: Illustrative hexagonal grid with initial black stone placed.

2.1 Winning possibilities

There are two options to win the game, each one requires a different checking function, both for checking if the game has ended and also for the evaluation function to select the best following move (see section 3.1). When a game ends a menu pops up showing some quick info about the game like the turns that have been played and also one of the two possible win conditions.

2.1.1 Five in a row

One player can win in this case by having 5 hexagons with the same color in a row in three possible directions: horizontal and the two diagonals. To check this, the first *CheckWin* function was created, it starts iterating from the last placed hexagon to see if that move completes a 5 in a row. The returned value is a boolean indicating if it's a win and also an integer with the largest encountered row. This last value is useful for the evaluation function (later explained in 3.1).

2.1.2 Locking up your enemy

For this winning condition one player has to fully enclose one or more enemy stones inside 6 or more of his own stones. This surrounding strategy was particularly hard to implement. The final *CheckWin2* function is a mixture of two solutions. First, we recursively try to find a path to get to the same last placed hexagon considering only the board tiles with the same color. If this path exists and the length is greater or equal to 6 (minimum number of stones to surround one enemy stone) then we can check is there's at least one enemy hexagon inside this "ring" path. This is made with a flood fill algorithm.

2.2 Final look and generating an executable

The figure 2 are examples of the final look of the program. As can be seen, no much effort has being placed into the game user interface (GUI), just the right amount for the game to run correctly and control all the settings for a game.

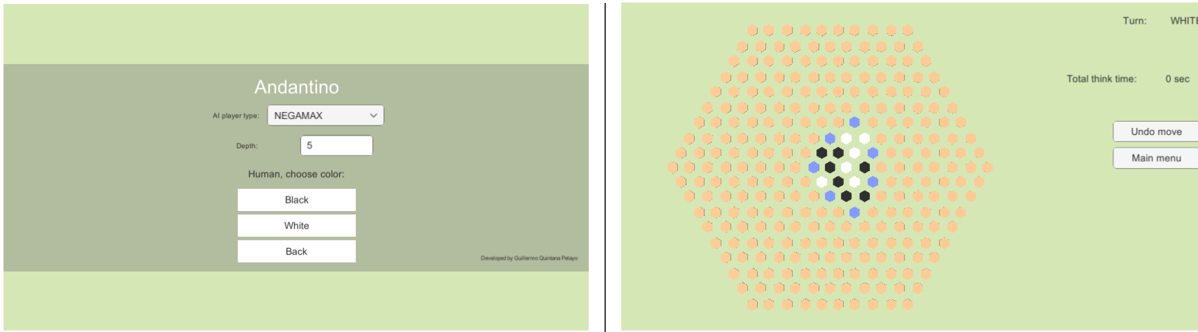

Figure 2: Left screenshot shows the AI vs Human Menu and the right one is an in-game screenshot.

In order to run the generated *.exe* some files need to be at the same directory, like the *Unity_Player.dll*. All of this required files come with this report. A class diagram with the final implementation is included in appendix 1 at the end of this report.

3 Developing the AI

Once the basic game structure is ready, it's time to implement the AI algorithms. In total, five actual alpha-beta approaches have been implemented without counting the random one (for testing). Since this is a project report, the actual algorithms will not be explained in detail because the actual point is to remark the principal aspects during the implementation of the different approaches and the results obtained.

The first decided step to follow was to implement a **random** algorithm that chooses randomly between the possible actions for the next turn. It's helpful for testing the game and new algorithms. It's very simple, as can be seen in this pseudocode:

```
RANDOM AI:
get possible moves for next turn
if there are not any possible move
    return
else
    select a random move between the possible ones
```

3.1 Evaluation function

Before implementing any new algorithm an evaluation function is needed. This function should provide an integer value with the score for a specific move and is used in the exploration to know the values of the leaf nodes in the search tree. To achieve this, the two check win methods explained in 2.1.1 and 2.1.2 were used.

If playing at the specified hexagon leads to a win then that's a score of 1000. If this is the case then the score is later multiplied by the actual searching depth so that encountering a win move in depth 5 (multiply by 5) is better that one in depth 0 (no multiplier). If the first check win function indicates that a row is being built and it's greater than 2 consecutive rows then the score is that length multiplied by 100. In any other case the returned score is a random value between 0 and 20.

```
int Evaluate(Hexagon h, int type){
    Tuple<bool,int> win = ctrTurns.CheckWin(h,t);
    bool win2 = ctrTurns.CheckWin2(h,t);
    if(win.Item1 || win2){
        return 1000;
    }else if(win.Item2 > 2){
        return win.Item2*100;
    }
    int r = new System.Random().Next(0,20);
    return r;
}
```

For all the searching algorithms that have been implemented there's a common set of instructions that need to be done before and after calling the recursive function. This instructions are basically used to recalculate the possible moves in the board for the next exploration and after the search to go back to the original board state. This avoids us to create copies of the entire board for every new exploration and use only the already existing one, it's useful for not using too much memory but it's dangerous if it's not handled correctly during the recursion.

The main idea is that one algorithm tries to search for a specific move, then that move is placed into the $grid$, the new possible moves are calculated and the next iteration of the algorithm is performed. When it gets back to follow a different path, the explored plays placed into the board are removed.

3.2 MiniMax

MiniMax uses a different version of the above mentioned evaluation function that also depends on the current player (Min or Max). The depth is normally reduced by one and the best score for each turn is returned so the best possible move is selected for the next play.

3.3 NegaMax

This method caused some interesting (and undesired) behaviours during it's testing and debugging. For example, during the tournament (see section 5) the method returned in some cases negative values that would correspond to a winning move and that's incorrect. New cases had been added into the stopping condition. First alpha-beta pruning was introduced with this approach. No modifications were done to the way $alpha$ and $beta$ refresh or the way the depth is decreased from the original algorithm.

3.4 Principal Variation Search (PVS)

The only remarkable thing to say about this approach is the new included recursive calls that produce more beta cut-offs.

3.5 Iterative Deepening with Variable Depth (IDVD)

The depth type was changed so now it is a *float* and the depth can be reduced in different ways. The next extract from the code shows one of the tested combinations for this reduction. If one specific move is interesting then the search should be more deep, so instead of -1, the next call to IDVD is done with depth -0.5. This is also the case with a less interesting play but reducing the depth with a value higher than 1.

```
newDepth = depth-1;
eval = Evaluate2(h,checkingType);

if(eval <100 && eval >50){
    newDepth = depth-1f;
}else if(eval <20){
    newDepth = depth-1.5f;
}else{
    newDepth = depth-0.5f;
}

value = -IDVD(child,newDepth,-beta,-alpha,newType);
```

3.6 Transposition Tables (TT)

This was the last method implemented into this Andantino Game Engine. For this approach one new class was implemented, the *TTValue*. The objects created from this class are used to store the moves information in a hash table, a $C\#$ dictionary to be more specific, using a hashed integer as the key and a *TTValue* object as value.

The hash value is computed with the Zobrist Hashing method. It uses the $grid$ information to create a unique key for the hash table. This method as will be demonstrated in the next section showed the best performance time and efficiency seen yet.

4 Testing

This section shows all the performed tests for the above explained AI implementations and the metrics used for that simulations.

First, a timer counter was introduced into the code to control the “thinking” time. This counter starts just before calling the specific AI algorithm to search and it’s stopped right before selecting the best move to play. Then, new counters were also included like: number of evaluation function calls, turns played...

In order to make the testing easier, a simulation tool was implemented into the game. When choosing “AI vs AI” you can also select an option called “Simulate” and introduce the number of simulations that you want to perform. This means that the two different selected AIs will play against each other this specified number of times and write all the relevant information into a text file.

As an example of this, the following table 1 shows 20 simulations performed for every pair of AIs played as the Black player and as the White player. In the table, the numbers are the victory count for the **black player**. All depths for the different searches were set to 7.

	Random B	MiniMax B	NegaMax B	PVS B	IDVD B	TT B
Random W	-	16	19	20	20	18
MiniMax W	13	-	19	14	19	19
NegaMax W	2	1	-	3	20	20
PVS W	7	5	2	-	20	20
IDVD W	14	9	12	18	-	18
TT W	0	5	3	4	9	-

Table 1: Correlation wins for black player in 20 simulations.

From this data we can see a clearly that transposition tables performs more smoothly, probably due to the fact that the explored board states are stored and don’t need to be re-evaluated. However, it’s also worth to mention that the average number of turns required (figure 3) to end the game with this method is a bit higher than PVS and NegaMax, and the worst case found was a game that run for 67 turns. It was one of the first one’s.

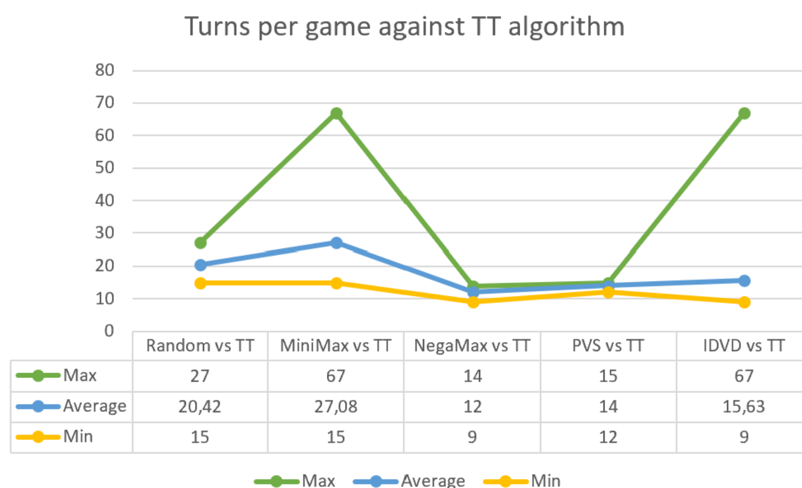

Figure 3: Illustrative hexagonal grid with initial black stone placed.

5 Tournament

For the tournament, I decided to do a mix between Transposition Tables and Iterative Deepening with Variable Depth, after some plays I also decided to try the NegaMax implementation for testing purposes.

During the tournament 14 matches were played, with 6 wins, 5 of them starting as the black player and only one as the white player, which clearly indicate something is wrong with the evaluation function, after some debugging the found error was that the player was not being updated correctly in some of the recursive iterations due to an if condition. The final presented work and the testing presented here were made after those fixes.

6 Further Work

During the tournament I noticed some interesting modifications made by the other participants like reducing the depth during game time according to the turn number. I'd like to have more time to include move ordering to improve PVS for example. And also to debug Transposition tables, because after the obtained results in the tournament, I'm sure something could be improved. Nevertheless, I think the implemented approaches and tests performed were successful and helped me to support the knowledge obtained with this course.

References

[1] D. Smith, "Andantino." http://www.di.fc.ul.pt/~jpn/gv/andantino.htm, 1995.

Appendix 1 - Class diagram

Figure 4 shows the class diagram for the final implementation.

Figure 4: Class diagram.