ANDANTINO
Project Report
Game Implementation and AI

Alexander Reisach (i6197692)

October 27, 2019

Abstract

The goal of this project is to implement an automated game playing algorithm for the two-player board game *Andantino*. For this purpose, a fully playable version of the game is implemented using a model-view-controler setup. Once this is achieved, a game playing agent based on an alpha-beta search framework is developed. The agent has to be able to take either of the two players' side and will have to make moves autonomously whilst adhering to a pre-defined time limit. Representations of game states are searched for optimal strategies using an enhanced version of the Negamax algorithm including iterative deepening and transposition tables. We find that those enhancements increase search depth substantially and propose further promising ideas for deeper searches and higher quality of game-play. Finally, this document contains an instruction manual describing the use of the game-playing algorithm.

Contents

1 Andantino

1.1 Game Rules

Andantino is a two player board game played on a 10x10 tile hexagonal board. The black player starts with a stone placed in the center of the board. The players take turns and stones can only be placed on tiles that share a border with at least two other occupied tiles (one other occupiedtile on the first move). A game is won once a player has reached either five connected tiles of his color in a straight line, or he has enclosed an arbitrary number greater than zero of enemy stones (and greater equal zero empty stone) with stones of his own.

1.2 Game Characteristics

With a total of 145 tiles, and a branching factor of only 2, Andantino lends itself to intelligent search algorithms. The game tends to give black an advantage by handing the black player the initiative from the start. However, with good play by white, draws and wins are well within reach for either player. Transpositions from onw position into another are possible when there are mutiple ways of reaching a certain outcome, however positions can not be reached repeatedly due to the add-on nature of the game.

2 Game Implementation

All components of the game were implemented in Python. While Python is not the fastest language due to its dynamically typed nature, it allows for rapid prototyping and short development cycles. Furthermore, compilation using e.g. *Cython* can improve run-time to get results closer to those of statically typed languages.

2.1 Software Architecture

The game was implemented following a Model-View-Controller achitecture. Additionally, the AI player has been implemented as a separate class of its own where each instance is handed a game instance. This architecture was chosen to achieve greater modularity of the components which allowed for more flexibility when trying out new approaches in the development process. As a result, the classes making up the game are:

- *Game_Logic*: Game state, legal moves and win conditions
- *View*: User interface
- *Controller*: Controller Game initialization and play out
- *AI*: Search based player instance

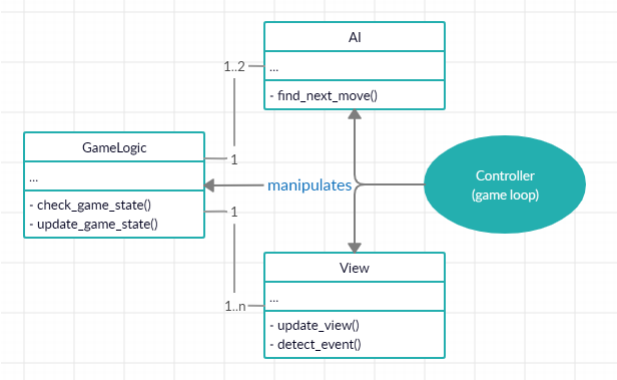

Figure 1: Classes (conceptualized)

The modules *Game_Logic* and *View* are implemented in an object-oriented fashion. Despite the possibility to access all class attributes directly, several getter and setter methods are used for a more consistent access pattern for similar queries of the game state. The *Controller* is implemented as a script following the requirements of *Pygame*'s game loop.

2.2 Data Structures

The game itself is represented as a Python dictionary containing the grid, turn and winner of a game. The grid in turn is represented as a nested list of tiles. tiles are again represented as dictionaries used to save the row, column and owner of each tile. In an earlier prototype, the game and tile were implemeted as classes, however this proved to be unnecessary overhead. The use of dictionaries allows for easy key-based access, iteration over elements and a simple way of manipulating the game state.

2.3 Game Actions

2.3.1 Legal Moves

The game rules are implelemented as part of the Controller module. Implementationally, the action of placing a stone is separated from its pre-and post-conditions with each being encoded in a function of their own. Any such function takes a game and information describing the action as a parameter. Whether a move is legal or not is checked based on the turn and neighbours of the tile about to be placed. The neighbours are identified according to their index in the nested lists making up the game grid.

2.3.2 Win Conditions

Five in a Row
The criterion for five in a row is implemented recursively. Each tile will recursively query neighbours in a straight line to count how many continuous tiles of the same color there are.

Surround
The surround win criterion is implemented in a depth-first floodfill fashion. Each tile recursively queries its neighbours to check whether there is a way to get to the border of the game board. If there is no such path without encountering a tile of enemy color, a surround is detected.

2.4 External Libraries

To save development time, a number of external public python libraries were used for the project. Most prominently, the graphical user interface is built entirely on top of the *Pygame* engine. Python's *logging* library was used extensively for debugging the game AI. To avoid undesired side-effects when copying dictionaries, *copy*'s *deepcopy* is used for copying nested mutable objects.Other external Libraries used in the project include *math* and *time*.

3 Search Techniques

3.1 Alpha-Beta-Search

The game AI used for this game is implemented using an alpha-beta format, more specifically, the Negamax algorithm. Alpha-Beta pruning is used to reduce the number of nodes explored while maintaining the quality of the evaluation. The AI's next move of choice is reset to the move in question, whenever there is an update of the evaluation at the very top of the search tree.

3.2 Debugging

To check the funcionality of the search techniques mentioned above, a debugging log as can be seen in 2 was used to keep track of the game tree. This proved to be a useful tool for ensuring correct evaluation values and prunings.

```
TURN 1

Iteration: 1
    |Depth: 0 -> Hash: 1057474577853092436
    |SUCCESSORS: [(8, 8), (8, 9), (9, 8), (9, 10), (10, 8), (10, 9)]
    |-> (8, 8)
        |Depth: 1 -> Hash: 3920428064494970004
        |Evaluation: 0
    |New candidate value at depth 0: 0
```

Figure 2: Debugging log excerpt

3.3 Evaluation Function

States with a definitive win are assigned a value greater than any other that can possibly be reached, plus their depth in their search tree to find the shortest win. Losing states are evaluated in the same way with an opposite sign. The evaluation of any other position relies on rewarding combinations of continuous lines. Each stone is assigned a value amounting to the sum of the lenghts of all continuous lines it is a part of in any of the three directions of the hexagonal board. In this method of evaluation, adding a stone to a continuous line of other connected stones will increase the evaluation not only by the value of the new stone, but also by the increase in value of the pre-existing ones. This results in an increasing marginal utility of continuous lines. The averages of these values per number of stones placed are taken for either player and subracted to get a comparative evaluation:

$$\frac{\sum_{own\_tiles}\sum_{directions} c}{\sum own\_tiles} - \frac{\sum_{opponent\_tiles}\sum_{directions} c}{\sum opponent\_tiles}$$

Where c denotes the lenth of the continuous line in a certain direction which the tile is a part of.

For example, in 3, the black player will evaluate the position as $\frac{3+3+3}{3} - \frac{2+2}{2} = 0$. Intuitively speaking, the black player does not consider this configuration advantageous since both players have been following the same strategy of laying straight lines.

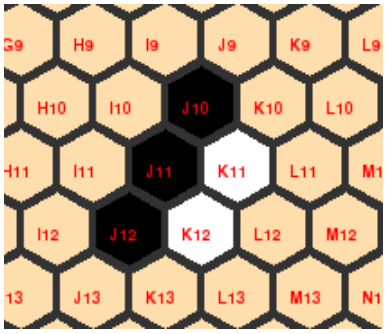

Figure 3: Evaluation of continuous lines

3.4 Performance

The evaluation function described in this paragraph favours placing long continuous lines whenever possible, with a tendency to maximize connected area if there is no obvious way of achieving lines of greater length. Since a large continuous area will usually result in mutliple possibilities to achieve n-in-a-row, this approach seems to serve the game objective well. Whenever a player using this evaluation function has the initiative, it will often go for simple wins, forcing the opponent to block. This results in disconnected chains and isolated stones for the opponent which allow for pressure play by threatening a surround win. When the AI player does not have the initiative, it will often try to achieve a large connected area with a high number of chains that are hard to block simultaneously, until it can gain the initiative. This method of evaluation was choosen after it proved able to beat a number of other hand-crafted evaluation functions.

4 Enhancements

4.1 Iterative Deepening

Iterative Deepening is used to keep track of the expansion of the game tree. The game tree is re-searched iteratively until a maximum depth is reached. This approach allows for usable intermediate results at shallow depth. These can be used for coming up with sufficiently good moves despite time pressure.

4.2 Transposition Tables

A Transposition table is used to store already processed variations of the game tree with their corresponding evaluation and search depth. Additionally, a flag indicating cut-offs at search time is stored. Whenever a new search is started, the transposition will already contain much of the work to be done, thus saving a lot of computation time, in particular much of the overhead introduced by the iterative depth search. The search-depth and flag on past cutoffs stored alongside the position in the transposition table give information about the quality of the value. Values found at a deeper level than the current level which have also not seen any cutoffs can be used as they are. Values seen at a deeper level which have seen cutoffs can be re-used to set the alpha or beta values for the current search iteration.

4.3 Move Ordering

If available, the values stored int he transposition table are also used for move ordering. The moves are ordered in a descending fashion where values are treated equally regardless of their flag. Positions which are not yet in the transposition table are put to the end of the list.

4.4 Time Management

Time management follows three separately defined strategies. For the first 80% if the time, a fixed time per move (based on the average length of games in self-play) is assigned. For the remaining 20%, each move can only take up a certain percentage of the time left. This leads to a gradual decrease in time spent per move as the game progresses. As a safeguard against loosing on time, a *panic mode* is implemented once only a certain amount of time is left. In this mode, a fixed search depth of 2 is applied.

4.5 Computational speedups

For runtime optimization, the static compiler *Cython* is used. A comparison of the search log has yielded that after compilation, higher search depths are maintained longer into the game, resulting in a higher quality of game-play.

4.6 Performance Evaluation

The following table compares nodes visited at each depth for a typical mid-game position:

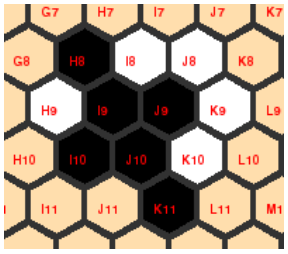

Figure 4: Game position used for performance evaluation

Iteration	Nodes visited	Nodes visited with transposition table
1	8	0
2	46	0
3	248	0
4	956	875
5	-	3843

Table 1: Performance evaluation

As can be seen in column two in 1, the introduction of a transposition table and move ordering are able to reduce the overhead of iterative deepening and improve overall search depth over regular alpha-beta pruning.

5 Ideas for Improvements

Upon ending the project there remain several domains for improvement. Apart from introducing new approaches such as opening books or move ordering heuristics, the following improvements based on existing components were determined to be the most obvious and promising.

5.1 Heuristics

The way in which the model of the game identifies legal moves and identifies structures such as surrounds and continuous lines on the board could be greatly improved by heuristics. For example, keeping track of legal moves would remove the need for checking repeatedly. The presence or absence of a surround win condition can be proven more efficiently by marking all tiles that are part of a single tile's floodfill search for a on unoccupied route to the border in one go. These and more opportunities have been postponed in favour of an expansion of alpha-beta search techniques but will likely speed up the search speed considerably.

5.2 Evaluation Funcion

The evaluation function currently in use works under several limiting assumptions. It does not yet reward a move towards a surround win and does not consider whether proximity of a tile to the border or to enemy tiles limits the tile's potential.

5.3 Thinking Ahead

The time taken by the opponent to make a move can be used to expand the transposition table. This would require a careful judgement of how much time an expansion will take, but might lead to greater search depths at almost no cost.

6 Conclusion

Implementing the game of *Andantino* in a modular MVC structure has been a successfull foundation for the development of a game-playing agent. Alpha-beta search proves to be an efficient search framework for the game. The enhancements lead to a jump in performance with notably more prunings due to the transposition table and move ordering. Iterative deepening and time management heuristics were successful in increasing the in-game performance for limited time controls and lead to a smooth degrading in play quality as positions get more complex. The enhancement approaches used for alpha-beta search are likely to benefit from further refinement and run-time optimization. Additional components such as opening books or heuristics will no doubt improve the performance of the game-playing algorithm even further.

A Instruction Manual

1. Open the folder *andantino_playable*, containing the executable and a *Cython*-optimized tournament program.
2. Doubleclick the file *controller.exe* to start the program.
3. Pick a color for the human player by clicking on either the white or the black square
4. Place a stone on a tile by clicking on it.
5. Wait until the AI's reply appears on the board (also indicated by an updated *AI Move*)
6. Keep playing - The blue *RETURN* button allows you to undo any unwanted actions.
7. The time (in seconds, counting down from ten minutes) left for the AI player is indicated next to the name of its color in the bottom right.
8. In case a player has won the game, the red text in the upper right corner will change to indicate this.

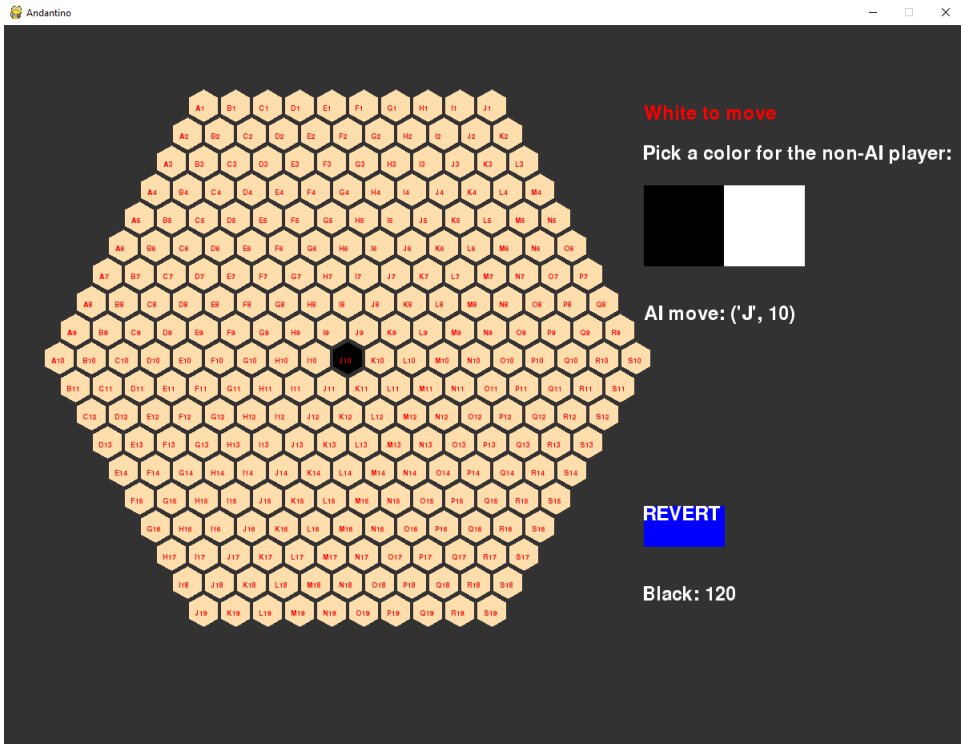

Figure 5: Andantino UI