Alpha-Beta Search for Andantino Game

Assignment Report

A. Aboukhadra

October 27, 2019

Contents

1 Introduction

Andantino is a connection game where 2 players take turns to place their pieces based on their colors. It is played normally on an infinite hexagonal grid but in our implementation, we constraint the size to 10x10 hexagonal grid. The game starts with a black piece in the center of the board and then the white player chooses a place adjacent to the center to place his white piece. After that, each played piece should be adjacent to at least 2 other pieces. The game ends when 1 of the 2 players have a 5-in-a-row of his pieces or if the player was able to trap or fully enclose 1 or more of enemy's pieces inside 6 or more of his pieces.

2 Program's description

The program was created using Java programming language. The Java project contains 13 files, 1 of them which is called `Hexagon` is only used to draw hexagons. The submitted file is a zip file that contains all 13 classes and an executable jar file. It can be easily run through any Java IDE like Eclipse or IntelliJ by importing the zip file inside the IDE. Another way to run the project is to execute the jar file using the command `java -jar andantino.jar`. The program starts by running the `Main` class which is a class that extends a JPanel and considered the GUI of the program.

2.1 Class: Main

The first view is where the player chooses the color that he wants to play with. After clicking the color, the view is switched to the game view. The mouse is used to play and the player needs to give an initial click to initialize the game. In case if the player plays with black then a white piece is randomly placed around the middle black piece to start the game.

To play your move, you can click the cell that you want to drop your piece on. There are 271 cells uniquely identified by a character that represents the diagonal axis and a number that represents the row and you can see the labels of the axis in the game view. For example, the middle piece is identified by J10. The program will not apply your move unless it is a valid one, which should be an empty cell adjacent to 2 non-empty cells. You're also allowed to undo your move to remove the last piece that was played by the AI and your last move. You cannot undo your move while the AI is searching. Once the game is finished, a notification appears announcing the winner and players cannot play anymore. These functionalities are all implemented in the `Main` class.

2.2 Class: Game

The controller is a class called `Game`, it carries the state of the game and other important game attributes. The state of the game is represented with a 1D array of size 271. Each entry carries the value of 0, 1 or 2 based on whether the

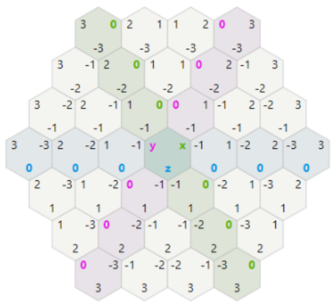

Figure 1: The 3 main axes of a cell with 2 directions for each axis.

corresponding cell is empty, black or white respectively. A move is represented with the index of the cell in the array. The same representation is used for the states in the search algorithms but we use a different class for that which is the `State` class and it will be discussed later.

The `Game` class also handles moves and compute the neighbors of each cell at the start of the game. Also, the main search function is implemented there and the search technique can be changed easily by commenting the current one and un-commenting the preferred one in the search function. To test the program, there is a random agent that plays any valid move. The game class initializes the 'r' array which is an array of random values used to do Zobrist hashing in transposition tables and it will be discussed further later.

2.3 Class: StateDecider

To decide the winner and to validate the moves, the `Game` class extends another class called `StateDecider` which contains 2 main methods:

1. `isValid(idx)` which is used to make sure that the move represented with idx is a valid one given the static state of the game. It grabs the cell's neighbors that are pre-calculated using shortest distance and stored in a static array and it makes sure that at least 2 of them are occupied.

2. `isWinning(idx)` which splits into 2 main winning conditions: five-in-a-row and trapped. five-in-a-row or `countConsecutive(idx)` method is calculated for a cell represented by the idx by inspecting the 3 main axes of that cell counting the number of similar cells on each axis on both directions from the cell on that axis. If the count is greater than 4 then it is a win. The 3 main axes and how there are 2 sides for each axis are shown in Figure 1. [1]

   This method is also used as a heuristic for the evaluation function. it has a helper recursive method responsible for checking one side of an axis which

[1]Image source: `https://www.redblobgames.com/grids/hexagons/`

is called `checkLine(idx, m, x, y, depth)` where m is the slope of the axis and x, y are the parent node indices and the depth is the number of cells reached so far in the same direction which is the value that we are looking for.

The second condition is the trapped condition. This condition is checked using the Flood fill algorithm. For each cell that is of the opposing color around the cell that was lastly played -represented by the idx-, we do a test to make sure that it isn't trapped. If the cell can reach an edge cell using Flood fill(DFS) through similar cells and empty cells, then it is not trapped. If one of the cells cannot reach an edge cell then it is trapped and the player who played the idx move won. A cell is an edge cell if it has less than 6 neighbors. The helper method is called `freeDFS(idx, color)` where color is the opposing color to the player who played the idx move.

We also implement a method called `isLosing(idx)` which handles the very rare case of playing a losing move. The only losing move that was found is to play the piece inside the enemy's empty traps. Every move is tested if it can reach the edge using Flood fill as well to see if it is a losing move or not.

2.4 Class: State

The `State` class extends `StateDecider` too to use the validity and winning conditions implemented there. The main attributes of a state is: 1) The grid representation that was described in section 2.2. 2) The index of the move that led to that state. 3) The hashCode of the state which will be discussed later. 4) The list of successors which are all the valid moves that can be played. 5) A boolean flag that indicates the player that should move.

The class contains the `getSuccessors()` method. It is responsible for scanning the whole grid for valid moves. It creates new grid representations and new states that correspond to those valid moves. Those states are appended to the successors' array. Furthermore, the class has a method to sort those states on their evaluation values to use them for Move Ordering techniques in search.

3 Search techniques

The search techniques are implemented in different classes but all of them have their original call from the `Game` class which passes the call to the `Search` class. All search strategies are implemented using the ideas discussed in the lectures of ISG course. The `Search` class has also the full implementation of Minimax, Alpha-Beta and Negamax algorithms. The search algorithms don't return the best score but they return a `Play` which consists of the best move and the score of the move. So instead of looking for the principal variation after the search, the best moves are propagated through the nodes of the tree. Every search method contains a variable called `color` that carries the color of the root player and it is used for the evaluation function.

3.1 Minimax

The naïve Minimax algorithm was implemented using the Pseudocode discussed in class. Minimax is a tree search algorithm where the Max player is the root node and it is trying to maximize the score while the Min player is trying to minimize it. The algorithm starts with checking the anchor case which is reaching depth 0 or a winning node. In that case, it evaluates the state and returns the move that led to that state alongside the value of the state.

Both Max and Min are checking all successors and they call the Minimax recursively on those successors after switching the turn flag and decrementing the depth. The only difference between the 2 players is that Max is trying to maximize their score starting from $-\infty$ and comparing all successors. On the contrary, Min tries to minimize their score from $+\infty$. Because we are propagating the best move that leads to the best move in the next depth, It is important to note that every node returns the move that led to the best node, not the move that is coming from the best node itself.

3.2 Alpha-Beta

Alpha-Beta on top of Minimax is a great improvement because it leads to a large number of tree prunings. We use this algorithm as the base code for all the rest of the techniques. Two parameters are added to the Minimax algorithm which are Alpha and Beta. Alpha holds a lower bound to the best score that can be achieved so the maximizing player now maximizes Alpha as well. On the other hand, beta holds the best score that the opponent can achieve and the minimizing player is minimizing it. Once alpha is greater than Beta, it means that further search in the current node is irrelevant so we stop the search at that node.

Negamax is a different implementation for the Alpha-Beta technique. In Negamax, every value is negated and alpha and beta values are negated and switched at every recursive call. Therefore, every player is maximizing their own score so there is no need for conditionals on the type of the player. However, Negamax needed more consideration for the parity of the depth and required a different evaluation function that needs more careful analysis at the leaf nodes, so the implementation was done but was not working perfectly. For that reason, the standard Alpha-Beta implementation was used for the rest of the techniques.

3.3 Iterative Deepening with Move Ordering

Iterative Deepening (ID) is defined as running the search repeatedly with increasing depth limit until the resources time out or the maximum depth is reached. The advantage of ID is that it makes good use for the resources allocated for the task. A constant depth at every state could cause instability problems and crashes or could be too small for some states. For example, if the branching factor at the start of the game is low then the search can go to high depth, however, at a different state where the branching factor is very high then the depth should be decreased to fit the branching factor. ID is adaptive to the state of the game and that's why it is very crucial to implement.

Another important aspect of ID is that we can use the results of depth d to order the moves at depth d+1. The assumption is that the best move at depth d could be a very good move at depth d+1. At our implementation, we sort the moves only at the root node every new depth based based on their evaluation score obtained from earlier depth. A time limit of 3 seconds is given to the ID with Move Ordering search (IDMO). This method is implemented in the `IDMO` class.

3.4 Transposition Tables

Some states could occur more than once during the search due to different order of moves leading to those states. To make a good use of the search at some state, the result of that search is stored using a representation for the state in a Transposition Table. Since the number of distinct states is in the order of 3^{271}, Zobrist hashing is used to limit the size of the state representation to 64-bit values.

For each cell in the grid there are 3 possible states which are empty, black, or white. Therefore, in the `Game` class, a 2d array of size 271x3 is initialized with random 64-bit numbers to represent the state of each cell. The Zobrist hashing representation for a state is defined as the XORing of the random values for all 271 cells of the state. To incrementally build the hash value of a state, the hash value of the parent state is XORed with the long value of the last played move and XORed with the long value that represent the cell when its empty. This is done to remove the effect of the empty cell from the hash value and adding the new move. In other words, `newHashCode = this.hashCode ^ Game.r[i][0]^ Game.r[i][newGrid[i]]`. The hash value is stored with each state as an additional parameter.

Java's `HashMap` is used to map the hash value of a state to a `TTEntry` object. The class `TTEntry` matches the structure presented in class and it contains the following attributes: 1) Value 2) Type of value (exact value, lower bound, upper bound). 3) Best move (the move that led to the value. 4) Search depth. 5) Hash Key. The hash value of a state is split into 2 parts (A primary hash code and a secondary hash core or a hash key). The hash code is the right most 30 bits and it is used for mapping the `TTEntry` in the `HashMap`. The hash key is the remaining part of the hash representation of the state. We do this split to avoid having large sized tables that exceed memory limits. The hash key is used to make sure that the retrieved entry matches the intended one by comparing it to the left 32 bits of the hash value stored in the state.

There are 2 main parts added to Alpha-Beta to include Transposition Tables. The first one is at the top of the method is the retrieval and comparison part. The `TTEntry` is retrieved from the map using the hash code of the state if the state exists in the map. If the state exists the depth is compared to check whether the value stored in the map was obtained from a larger depth or not. Because, otherwise, we are retrieving a inaccurate value. The hash key verification is done at the same condition as well. Based on the type of value we decide whether to return it or to use it as a new Alpha or a new Beta.

The second part is responsible for deciding the type of the new value and storing new entries in the table. This part is added after the Alpha-Beta search. The collision error is observed and was very low, therefore, it was ignored by using the replacement scheme of always storing the new value as it was the simplest to implement. An iterative deepening version of the Transposition Tables is implemented just to utilize time resources without the use of move ordering. The time limit was 3 seconds.

3.5 Killer Moves

Killer moves are defined as the moves that lead to an $\alpha - \beta$ pruning. Those moves are expected to produce more prunings when investigated at the same depth in a different node. In the implementation, 2 killer moves are maintained for each depth. They are stored whenever a pruning happens by moving the old first killer move to be the second one and setting the first killer move to the new one. If 1 or 2 of the successors of a node correspond to killer moves at that depth, they are swapped with the first 1 or 2 successors in the successors' list to be considered first in the search. This step is done after Transposition table consideration. Transposition Table variations with killer moves were implemented in `TranspositionTable` [2]

The iterative deepening version of this method was also implemented with a time limit of 3 seconds and a maximum depth of 11 ply deep. It was noticed that Alpha-Beta with Transposition Tables and killer moves in iterative deepening fashion was the most successful approach in terms of depth reached and time needed to play a move. We used this approach for participating in the Andantino tournament and ranked 4th place among 32 participants.

4 Evaluation Function

The static evaluation function is implemented in the `Evaluator` class. The evaluation function starts with checking for the winning condition. If the root player is winning then the maximum score value of 500 is returned, however, it is subtracted by the depth to encourage the algorithm to play the winning move that has the shortest depth. If the position is a losing one then the score will be -500 and the current depth is added to the score to stall losing as much as possible in case the opponent doesn't realize it's winning yet. If the state is not a winning or a losing one, then the evaluation function consults 5 different heuristic functions. Only 1 of the heuristic functions isn't implemented in the `Evaluator` class. It is important to mention that all heuristic functions consider only the last move to be played not the state of the whole board.

[2]The implementation was inspired from the answer at the following link: `https://stackoverflow.com/questions/17692867/implementing-killer-heuristic-in-alpha-beta-search-in-chess`

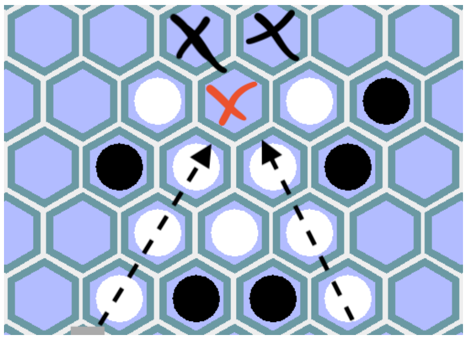

Figure 2: If the white player plays at the cell marked with a red X then the white player won because the black player can stop 1 diagonal by playing at a cell with a black X but cannot stop the other diagonal so the black player has to stop the white player by playing at the red X cell. That's why the fiveInARowComponents method is used.

4.1 Five-in-a-row heuristics

There are 2 types of evaluating whether the last move contribute to a five-in-a-row winning condition or not. The first one is the `countConsecutive(idx)` which was discussed earlier in the Section 2.3. It is responsible for counting the number of pieces in the 3 axis with respect to the idx move and returns the maximum of them. This method tells how many pieces left to complete a five-in-a-row component which gives an indication for winning or losing. 2 weights are associated with the value to indicate performance of the heuristic. Those value are 6 and -6 to represent which player is winning -Max or Min-. This method is the only heuristic that isn't implemented in `Evaluator` class.

The second heuristic function related to five-in-a-row winning condition is `fiveInARowComponents(s, idx)`. It is very similar to the first one but the only difference is instead of returning the maximum value, it returns the sum of all rows in different directions. A noticeable strategy for winning is to build 2 rows in different directions simultaneously to force the opponent to stop one of them while completing the other one. An example for this strategy is shown in Figure 2.

4.2 Size of the Connected Component

Playing your piece next to your other pieces is a defensive mechanism the prevents the other player from trapping your pieces. The method `countComponent(s, idx)` is used to count the number of pieces in the connected component of the cell indicated by idx. The value produced is associated with the weights 4 and -1 based on the player as the root player doesn't care about enemy's components as much as he is concerned with his component and defending his pieces.

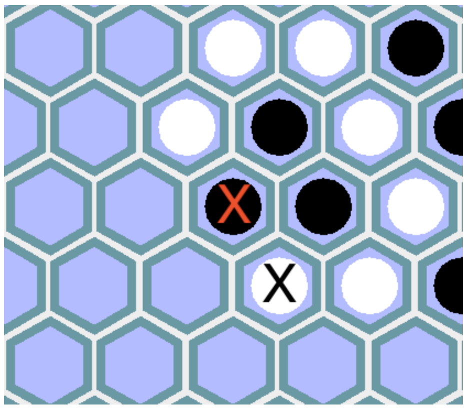

Figure 3: The enclosing value for the cell marked with a white X is the maximum trapped value for its neighbors. The trapped value for the cell marked with a red X is the number of white pieces in the 2 layers surrounding it which is 7 white pieces.

4.3 Enclosing Value

Counting the number of the player's pieces around the opponent's piece gives an indication whether the player is surrounding the opponent or not. For the last played cell, the enclosing value is the maximum between the trapped value for all its neighbors. The trapped value for a cell is counting the number of opponent pieces in the 2 layers surrounding it. This is done using Breadth First Search (BFS). An example for this value can be found in Figure 3. The method that calculates the enclosing value is called `enclosementValue(s, idx)` and has a helper function which is `trappedBFS(idx)` that calculates the trapped value for the neighbor represented with idx.

4.4 Chain Value

Getting closer to trapping enemy's pieces can be represented with the distance between the 2 ends of the chain surrounding the enemy. If the distance is short then the trap is about to be complete so there is an inverse relation between the distance and the trapping value. So the chain value is calculated using $\frac{1}{dist}$ where dist is the euclidean distance between the chain's ends. This value is scaled by the depth to which the 2 ends are far apart and this depth is calculated using BFS. The value is also multiplied by 100 to avoid floating points. The method `chainValueBFS(s, idx)` calculates the described value for all nodes in the chain and returns the maximum among them. The weights of this value are 5 and -5 depending on the player.

Search Strategy	Initial depth	Mid-Game
Minimax	6	4
Alpha-Beta	7	5
ID with Move Ordering	8	5
Transposition Tables (TT)	8	6
TT with ID	8 (11)	6
TT with killer moves	8	7
TT with killer moves and ID	8 (11)	7

Table 1: Initial depth and mid-game depth for different search strategies. Value between brackets show the maximum depth that was ever reached when the number of valid moves was less than 4.

Search Strategy/Valid Moves	2	4	6	8	10
Minimax	825	6194	22407	148655	417435
Alpha-Beta	450	1708	3411	14082	46534
ID with Move Ordering	351	1624	3546	21400	30132
Transposition Tables (TT)	306	1168	5235	10982	26123
TT with ID	312	974	2786	7550	24438
TT with killer moves	237	952	1839	11337	19631
TT with killer moves and ID	240	786	2329	3968	5812

Table 2: The number of explored states for each search strategy based on the number of valid moves.

5 Results and Compraison

Seven different search techniques were investigated. In Table 1 we compare between the depth that each technique can reach. We distinguish between 2 types of depth which are the initial depth and the mid-game depth. Initial depth stands for the maximum stable depth that the search can reach without crashing and in relatively short time when the number of valid move is around 10. Mid-Game depth is the maximum depth that the search can reach without crashing and in short time when the number of valid move is around 20.

In Table 2, we show the number of states explored by each algorithm at different states of the game while maintaining a maximum depth of 5. The table shows how the number of explored states drops significantly by adding different pruning techniques. It is important to note that for iterative deepening versions, the reported number of states is only at the maximum depth reached, that is 5.

6 Conclusion

Alpha-Beta pruning is a intelligent optimization for Minimax algorithm and improves its performance. Adding Transposition Tables helps in remembering explored states such that no additional investigation is needed for them. Iterative deepening can utilize time resources in a smart way. Move ordering at all nodes needs to be investigated further as it showed good performance when applied to the root node only. Therefore, Using Transposition Tables and history heuristics to order moves could be part of future work to optimize the search. Killer moves are considered a type of move ordering that was easy to implement and showed good performance when compared to other techniques. Another part of possible future improvements should be done to the evaluation function as extra tuning for the weights of each heuristic in the evaluation function is needed to make it more realistic. Combining all former techniques resulted in an efficient algorithm that was able to rank 4th in the Andantino tournament among 32 participants.