Alpha Beta Search on Andantino

Sai Koneru

October 26

Contents

Abstract

Andantino is a two-player game played on an infinite hexagonal board. It is a turn-taking game. Each player wins by making five in a row or surrounding the opponent. Alpha-Beta search is used in the project and this report compares a few of the different enhancements that are possible. All the enhancements with Alpha-Beta will work if there is a good evaluation function. Firstly, we need to develop an evaluation function that correlates with the true value of that game state. Then the Alpha-beta search algorithm, Mini-Max algorithm are compared to see the reduction in the total number of nodes visited. Then we use transposition tables with the current search algorithm and find the best way to handle collisions by comparing different approaches. Move-ordering is also discussed in the report and its impact on the current search algorithm. Iterative deepening for real time play is used in this implementation. Finally, Odd-Even effect is also addressed concerning the game and if it should be avoided.

1 Alpha - Beta

Alpha-Beta is an extension to the MiniMax search algorithm. Games like tic tac toe can be solved using MiniMax as the state spaces is relatively

small, but for games like chess and Andantino we spend too much time evaluating the nodes which min or max player will never choose. In this section MiniMax and Alpha-Beta search are compared. A key element of MiniMax or Alpha-Beta is an evaluation. Before we can compare the performance between the two search algorithms we need to define an evaluation function.

1.1 Evaluation Function

We cannot go deep enough in the tree until we find a win. It's possible for simpler games like tic tac toe but for andantino it is not practical. It would take far too long and a huge amount of memory. To solve this, we take help of an evaluation function that tells us how good it is to be in that state. A good evaluation function is a key element in the Alpha-Beta search algorithm. If the evaluation function doesn't correlate to the actual value of that particular state then we might prune the states that are the states present in the principal variation line when using Mini-Max. When move-ordering is applied and if the evaluation function is not proper then we might also have a case where there are very few prunings. Each game has its unique evaluation function which is generally created with the help of human knowledge. In Andantino five in a row or surrounding an opponent piece or pieces leads to a win. So our evaluation function should tell us which states are better by comparing how closer and probable they are to achieve their goal.

We can think that four pieces in a row are much closer to achieve the goal than three in a row. So, we start by making different features in our evaluation function that matter to our current game state. In andantino we can define the features as follows

- Two same pieces in a row
- Three same pieces in a row
- Four same pieces in a row
- Surroundability of a particular piece or pieces in the game state
- Winning Features

The features above make sense intuitively and we can say that if we have two "four in a row" features and one "three in a row" feature then we conclude that latter is worse. It might not always be the case.In figure 1 we can see that even though there are two rows of length four it is not a good state to be in for white. We also have to take account that if it is still possible to finish that row and then factor it in to our evaluation function. It is much better state for white to make five in a row shown in figure 2 than figure 1.

In figure 3 we can see a kind of winning feature. It is a win for black and loss for win if black plays optimally. We can assign this pattern as a winning feature and if we ever could achieve this state than black always should pick it. There are other features that are possible in the game such as a player having two "Four in a row" and both having the same start piece. If we have more patterns like this then we are more likely to win the game without it.

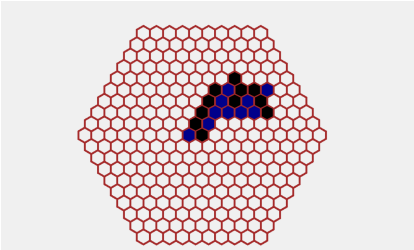

Figure 1: Blocked Rows

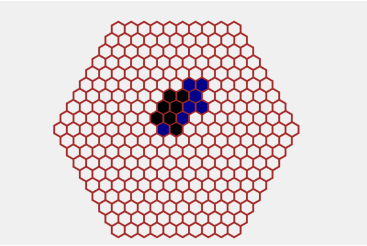

Figure 2: Unblocked Rows

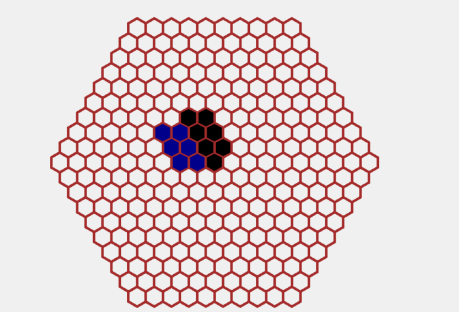

Figure 3: Winning Feature

We can also add additional domain knowledge in the evaluation function. If white has to move and black has two possibilities to win, then whatever white does black will win. We need to identify what game states lead to this kind of features. After detecting a winning feature we can give it a high value as we are certain that it is a win state in the future. This allows our search to be more tactfulness as our evaluation function is looking ahead. It also saves us time as it will look ahead and if its a win/lose it will lead to more prunings.

We now have defined various features that we would want in our function but we have to assign weights for these features. It can be done through trial and error and also with some intuition. It is obvious that "four in a row" feature must be given higher weight than "3 in a row", but we need to determine by how much we want it to value. Neural Networks are used for evaluation functions but it needs high quality training data and it is also slower. We can also determine the weights by using temporal difference learning which is much more effective than trial and error. We can also add whose turn it is to move in our evaluation function. In a game like Andantino it is not really that impactful. There are no pieces to capture and two boards cannot be having different players turn if they have the same pieces in the board. Now we can define our evaluation function as shown in

$$WhiteRowScore = 1 \times TwoInARow + 2 \times ThreeInARow + 4 \times FourInARow + 10 \times WinningFeatures \quad (1)$$

$$WhiteScore = WhiteRowScore + BlackSurroundability \quad (2)$$

$$BlackRowScore = 1 \times TwoInARow + 2 \times ThreeInARow + 4 \times FourInARow + 10 \times WinningFeatures \quad (3)$$

$$BlackScore = BlackRowScore + WhiteSurroundability \quad (4)$$

$$TotalScoreForWhite = WhiteScore - BlackScore \quad (5)$$

$$TotalScoreForBlack = BlackScore - WhiteScore \quad (6)$$

From equation (1) and equation (3) we can see that how much each of these features are weighted according to each other. When we calculate the score for white piece it is equal to the difference between them that is $WhiteScore - BlackScore$ and vice versa. This tells us how strong is our position compared to the other player. It is much better indication of our position than our score itself. After finalizing on our evaluation function we can now start implementing our search algorithm.

1.2 Mini-Max vs Alpha-Beta

Mini-Max search tree considers all the possible states from a given board position. It goes until a leaf node or terminal node is found and then backpropagate its values. There are several different states where the min or max player would not think it's possible to reach. They assume that the opponent is playing optimally and would never allow us to transition to those states. Alpha Beta algorithm eliminates these irrelevant nodes and decreases the time we spend in the search tree. This allows us to dig deeper than we could with our time constraints.

The table 1 gives a indication of the impact on the size of the tree between the two algorithms. This table is created on depth 4. As the number of moves increase the size of the tree also increases because we have many more moves we need to take into consideration. We compare the number of nodes visited by the algorithms. If the evaluation function is same for both the algorithms then both algorithms will result in the same move after building their tree. The third column indicates the reduction percentage of the nodes we visited. We can see that we almost save visiting 80 percent of the nodes. Evaluation functions are expensive in general. Using Alpha-Beta we can prune these irrelevant nodes. As the moves increase we can see the increase in the number of nodes is exponential. As the number of moves increases the reduction percentage also tends to increase but then drops down as the size of the tree gets much larger. If we increase the depth of the tree then the impact of alpha beta gets significant. In this setting alpha and beta are initially set to $-\infty and \infty$. We then gradually update our bounds as we go through the tree. Various windowing techniques have been proposed to reduce the tree further. We can try experiment with different values of this bounds if we have a good idea of what they can be. We can also use adaptive schemes to figure out the bonds by the agent itself. If we underestimate the bonds than we might play wrong moves as a result.

Table 1: Mini-Max vs Alpha-Beta

Depth of The Tree=4 Number of Moves	Total Number of Nodes		Reduction Percentage
	MiniMax	Alpha-Beta	
2	305	77	74%
4	1649	231	86%
6	3283	353	89%
8	7405	975	87%
10	11307	2343	79%

2 Transposition Table

If we can reach the same board state with the moves ordered in a different way than we can take the help of Transposition tables. Memory consumption increases as a trade-off in the decrease of time. We cannot remember all the states we visited during the whole game. If we calculate the value of a board state in a tree and if we end up visiting the same board state in the same or different tree, we don't need to calculate the score from that board. We can look up in our table and if it exists then we can pass our score.
We cannot remember all the possible states that we have encountered. Zorbist Hashing has been used in this implementation to encounter this problem. Hash Table also helps us in checking if the board exists in our lookup table without searching at every position. We need to save the most important information in our table. The following details are stored in the tranposition table implemented.

- Flag
- Depth
- BestMove
- Score
- HashKey(From Zorbist Hashing)

2.1 Alpha-Beta With Transposition Table

In Andantino we can use the Transposition table if we are looking at a depth of more than three. Consider the initial position where white has to play. Suppose white plays i9, black plays i10 and white follows by j11. This can be achieved in a different order where white plays j11, black plays i10 and white follows by i9. Both boards have the same configuration. If we were not using a transposition table then we would have evaluated the board again which is very expensive and also evaluate other boards that follow from this depth.

Now let us consider the opening position for white and compare the results with and without the use of a Transposition table. "Found in TT" column indicates the number of nodes that we found in the transposition table. In the Table 2 we can see that at a search depth of two there are no nodes found in the transposition table. As we increase our search depth

Table 2: Impact of Transposition Table on Andantino

Search Depth	Total Number of Nodes	
	Found in TT	Whole Tree
2	0	14
4	3	84
6	25	1171
8	765	17691

Table 3: New vs Old Replacement Strategy

Number of Bits	Replacement Strategy(No of Nodes visited in total for white)			
	New	Number of Moves	Old	Number of Moves
2^8	18868	25	41285	33
2^12	88476	36	227284	54
2^16	71939	35	71707	35
2^20	72489	35	72525	35

the number of nodes we save start to increase. It is even better in some cases when there is pruning at a node found in the transposition table. As the game progress in mid game and number of possible moves tend to increase the use of transposition table becomes even more influential.

2.2 Replacement Strategies

There will be collisions in our hash table as only store part of the possible boards that are possible. There can be two type of collisions one where both the key and the remaining bits are some, other where only the key is the same. First one is hard to detect and occurs rarely. Using 20 bits for the key can help making this collision very rare. Second one can be addressed by using different replacement strategies. In this report New and Old strategy are compared in the game Andantino. These results are also correlating as shown in this book.

When using New replacement strategy, we replace the previous no matter the other parameters of the entry are and in the Old replacement strategy we do not replace the entry. In the Table 3 the first column indicates the number of bits that we use as a key in the hash table. This table is filled with an agent of depth 4. We can see that as we increase the number of bits we allocate the difference in using this strategies start to shrink. This is because of the hashtable size, If it is big then the chance of collision start to decrease and also the impact of replacement strategies.

3 Move Ordering

Move ordering scheme is extremely important in the alpha beta search. If we can order our moves so that we can prune as early as possible it allows us to search much more deeper. There are different techniques or schemes that can be implemented. Child nodes can be evaluated and ordered so that we explore the best move first and prune early. Transposition table moves can also be tried first as they are more likely to be a good move. Moves which led to pruning(Killer Moves) can be saved and should be searched first before the other possible moves.

Killer move seems to yield better results than transposition table moves if searched first in this implementation. The following move ordering scheme is applied

1. All the boards are evaluated at every depth as we also will use iterative deepening. First the winning/losing boards are searched.
2. Moves which are present in the killer move list are searched.
3. Transposition table moves are applied after the killer moves
4. Remaining boards are ordered according to their heuristic values.

Further enhancements can be done to speed up the process such as to evaluate only when they are frontier nodes to save time. We don't need evaluate when we are near leaf nodes as it is redundant. We can also try the move in the transposition table as the first move.

4 Iterative Deepening

Real time game playing agents should adhere to time constraints. We also don't know how long we should allocate the time for our agent. Initially we can start by giving less time and as the game progresses we can allocate more time. The end game may require less time because the number of possible moves start to decrease but it is crucial to search deep in end games. Overhead of calculating again at every depth can be eliminated with the help of transposition tables. Every node in the tree is evaluated at every depth but this can help to cause more prunings as it has more accurate information.

Lets suppose that each player has a time of ten minutes. In the start game we can limit our search. We don't want to waste time using the full allocated time. After we finish achieving a certain depth that we can guess to yield good results we can stop the search and return the move. This will save time in two cases. Firstly, In the start position where there are only two possible moves. Secondly, If there is any obvious move that the agent should play we do not want to waste time searching again to get the same move. Each move is allocated for a period of 10 seconds and search depth of maximum 8 is used in this implementation.

5 Odd-Even Effect

When an agent is search at an odd depth its tree is evaluated on the basis of the last move played by him. This will result in back propagating the leaf values that are generally optimistic. Similarly, when searching at an even depth the agent behaves pessimistically. This can be solved by adding whose move as a parameter in the evaluation function or by iterating two at a time.

In Andantino black player is more than 90 percent likely to win against white. This means that if the agent is black he should look optimistically as he has a significant advantage than the other player. As for the white player he should look at even depth as he is in disadvantage. To conclude, Odd even effect should not be addressed in andantino and agents should look at their tree on odd or even depth depending upon their color.