Developing an Adversarial Agent

Declan Simkins, Rebecca Walton, Jonathan Ismail
Macewan University

Abstract – A common approach in dealing with decision based game theory when creating a rational agent is implementing an effective Minimax algorithm. Our paper focuses on strategies of developing an agent for the game Kōnane otherwise known as Hawaiian checkers which is a two player, zero-sum strategic board game. The Minimax algorithm is a recursive algorithm best fitted for maximizing a players' move used in zero-sum games to denote minimizing the opponents maximum payoff. During the course of our work, we investigate different methods of implementation that increase the algorithms' effectiveness given a restricted time limit. The scope of our paper describes a detailed analysis of our Minimax algorithm which is rep- resented by an evaluation function and illustrates the problems encountered along development and possible solutions that improve the effectiveness of the algorithm.

I. Introduction

Kōnane is a two player strategy game with an 8x8 board filled with alternating patterns of black and white pieces. The goal of the game is to reach a state in which one player is unable to capture an enemy piece. Players have the ability to maneuver across the board by jumping over adjacent pieces thus removing opponents pieces. If desired, players also have the ability to perform multiple jumps which may or may not benefit the jumping player. Therefore, to create an intelligent adversarial agent, one must develop a logical path in which the agent can employ a comprehensive technique in achieving a desired goal. The Minimax algorithm is a decision rule based algorithm formulated for two-player zero-sum game theory which minimizes the possible loss for a worst case scenarios.. The algorithm covers both cases in which players take alternate moves and extends to more complex decision making games where uncertainty in present [1]. Considering the nature of Kōnanes' zero-sum environment, our team implemented the Minimax algorithm when developing our adversarial agent.

1. 1 Minimax

This algorithm uses an informed adversarial search that utilizes a cause-and-effect search that considers each possible action available at a given moment, then considers subsequent moves for each of those state in attempt to satisfy a goal condition. Our team chose to use this algorithm after dissecting the Kōnanes' deterministic environment. Our implementation of Minimax takes a the current state, a depth limit and a pointer to a move (represented in the chess notation). The current state is stored as a node which contains the current board, the heuristic value and the parents and children of the node. The depth limit is used for one of our enhancements, implementing iterative deepening minimax, which will be discussed later. Our function returns the index of the best move in the list of generated successors as well as placing the move in chess notation in the location of move.

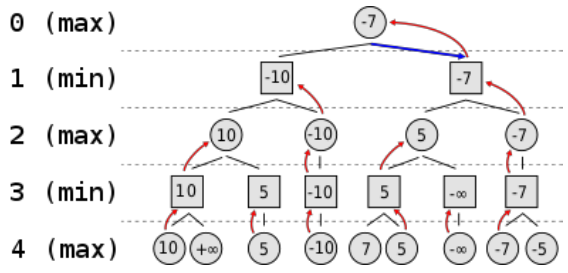

Fig. 1. The image above provides a depicted description of the Minimax algorithm

For each node visited either by max or by min our algorithm generates all the possible successors of that node. The successors are then explored by the opposing side (ex. max if we are currently in a min call) starting with the best

```
void evaluate(node_s *node)
{
    static const char black = 'B';
    static const char white = 'W';
    int moves_b = available_moves(black, node);
    int moves_w = available_moves(white, node);

    switch (model->colour) {
        case 'B':
            node->heuristic = moves_b - moves_w;
            break;
        case 'W':
            node->heuristic = moves_w - moves_b;
            break;
    }

}
```

Fig. 2. The above provides the code for the evaluation function used in determining a heuristic for possible moves. The heuristic is the amount of available moves for the agent minus the amount of available moves of the opponent.

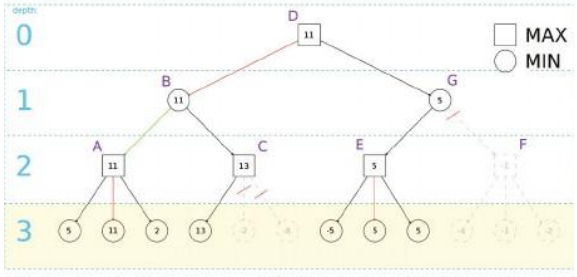

Fig. 3. The image above presents a visual description of the Minimax algorithm with Alpha-Beta Pruning and Iterative Deepening.

heuristic value and moving towards the worst. Once all successors of a node have been appropriately considered, our function then updates the heuristic value of its parents to be the alpha or beta value discovered. Since our agent has limited time, in the event that it runs out of time the previous best move is returned.

Our evaluation function is very simple as we prioritized speed over accuracy. We calculate the number of moves left for our program and the number of moves left for the other agent. Our heuristic value is the subtraction of the number of moves the other program has left from the number of moves our program has left. Since the goal is to make the last move, knowing the number of moves each player has left is a good indicator as to whether a state would be beneficial or not.

II. Implementation and Enhancements

2.1 Alpha-beta Pruning & Iterative Deepening

The first enhancement we made to our Minimax algorithm was to implement Alpha-beta pruning which seeks to decrease the number of nodes that are evaluated by the traditional Minimax algorithm in its search tree. This enhancement would increase the probability of finding the "best" move by terminating the evaluation of a path that has been proven to be worse than a previously examined move [2]. When this enhancement was applied to out Minimax algorithm, it allowed us to set a deeper cut off value for our search which could still be completed in the time limit provided.

Secondly, we enhanced our search by applying iterative deepening minimax. Simply using the alpha-beta method, we noticed that weren't able to use partial results with confidence and had to set a cutoff value that did not necessarily take advantage of the full time limit. By using iterative deepening minimax we were able to be conservative and set a depth-limit which guaranteed finding a move given a time limit. We used a separate thread to keep track of time, and when the clock reached 10 seconds, our algorithm would use the solution found in the previous depth limit. This greatly improved our searching ability as we were able to reach greater depths and use more of the time we were given for searching.

Finally we implemented move ordering.. The move ordering allowed us to pick the node that we thought would be most likely to be the best first which would allow alpha-beta pruning to prune more branches if we were indeed correct about the "best" move.

III. Result

	Minimax Agent	+ Alpha-Beta Pruning	+ Iterative-Deepening	+ Move ordering
Win Ratio against Random Agent	40%	80%	100%	100%
Time	< 10s	< 10s	< 10s	< 10s

Fig. 4. The table above describes the Minimax algorithm playing against a Random Agent who's moves are chosen at random and the Win ratio of the enhancements of the algorithm throughout development. The ratio is based on 5 games played and the percentage of wins our created Agent has.

IV. Problems

Initially our agent had some problems generating moves and detecting win states. It would miss available moves and would determine that win states had been achieved when there were still moves available for players. By careful examination of problematic board states we were able to determine that an off-by-one error prevented us from making a jump from row 6 to row 8 or column C to column A. Fixing this mistake not only allowed us to ensure that win states were detected accurately but also improved the effectiveness of our agent as it was now considering all possible moves instead of just a subset of them.

We also experienced problems having our agent select the "best" move. In the competition against our classmates agents we lost nearly every round and we even lost to the random agent once. It seemed that our agent sometimes made suboptimal moves. In an attempt to fix this we came up with some other evaluation functions. However, these did not improve our chances of winning and in some cases caused us to lose every game. We then decided that it must be our implementation of the algorithm that caused the problem. While tracing our code we realized that our algorithm would determine the best move but then could often not find that value in the successors and would instead just return the best of the immediate successors, effectively just searching a depth of one. To solve this problem we updated the value of the parent when we found a better value for max or min (depending on whose turn we were evaluating). This gave us significant improvement in our performance, allowing us to even beat KoNoName if we were playing as white.

Finally, we attempted to implement a hash table to help with detecting duplicate states. However, this caused multiple problems with memory leaks and then segmentation faults. Even when it ran with memory leaks it did not provide a noticeable benefit to the performance of our agent. As a result, we decided to remove the hash table and move it to the future improvements section.

V. Future works

To improve our agent we would like to add a hash table to determine if we've seen a node previously and also to aid our move ordering. If we had a hash table we would not have to re-evaluate nodes which could give a small advantage. The big improvement would be in the aiding of move ordering. By using a hash table we would be able to use the values of nodes from previous iterations of our iterative deepening minimax to more accurately pick the best move to try first.

We would also like to combine our generation of chess notation moves and our generation of nodes for the search and store the chess move that caused the state in the node. This would greatly reduce the number of malloc calls and would simplify the return of the correct move in chess notation.

VI. Conclusion

During development, we've observed the effectiveness of using the Minimax algorithm as a solution for developing an intelligent agent. We noticed that the algorithm, if implemented correctly can result in an optimal move being performed in zero-sum games, which was demonstrated in our game agent for Kōnane. However, the specific nature of Kōnane adds elements that require a more complex algorithm then the traditional Minimax. Seeing as Kōnane introduces an element of time, we would require an agent to play optimally within a 10 second time limit, therefore we've implemented additional modifications to the Minimax algorithm . First was the creation implementation a alpha-beta pruning techniques which also saved time by terminating paths of less optimal moves, however, did not guarantee the best move within 10 seconds. We then used iterative deepening which set a limit to our search and returned the best move from the previous iteration which allowed us to take advantage of all the time we had. The enhancements given to our algorithm saves time during our search and thus additionally increases its depth which leads to a greater chance of finding a true "best" move. Our algorithm does not always guarantee a win against its opponents, however provides sufficient strategies of playing intelligently for the game of Kōnane.

VII. Reference

https://wikivisually.com/wiki/Minimax
https://en.wikipedia.org/wiki/Alpha – beta_pruning

https://www.ics.uci.edu/~rickl/courses/cs-171/2012-wq