COMP 424 - Project Report:
Tablut

Halil Murat Goksel (murat.goksel@mail.mcgill.ca)

April 8, 2018

1 Introduction

This report is written for the graded project of COMP 424 *Artificial Intelligence* course taught at McGill University in Winter 2018 semester. The goal of the project was to implement an A.I. player which plays the ancient Viking board game *Tablut*. Tablut is a zero-sum game of perfect information. What distinguishes it from many other board games such as Tic-Tac-Toe, Chess, etc. is that it is asymmetrical in the sense that the goal of the player is different than that of the opponent.

2 Explanation and motivation

This section explains how the artificial intelligence program works and motivation for the chosen approach.

The program is fundamentally a *minimax*-search implementation with $\alpha - \beta$ pruning. It consists of three major methods, namely `chooseMove`, `alphaBeta`, and `heuristicValue`. `chooseMove` is the method in which all successor states of current board state are evaluated and the one with the highest value is returned as the chosen move. Starting at state `s`, `alphaBeta(s)` method performs the minimax-search on the board-state tree until it reaches a node where the game is over or the maximum search depth is reached, and then returns the heuristic value of `s`. It uses $\alpha - \beta$ pruning to eliminate some of the nodes in the state tree to increase performance. `heuristicValue` calculates, as the name suggests, the heuristic value of a given board state.

Minimax algorithm is known for its use in the context of zero-sum games of perfect information. IBM's *Deep Blue* chess program is well known for defeating world champion Gary Kasparov in an exhibition match in the 1990's. Deep Blue ran on a parallel computer doing alpha-beta search[1], which is a variant of minimax. Tablut is quite similar to chess except for being asymmetric in terms of winning strategies. The asymmetry arises from the fact that, while Muscovites

[1] Russell & Norvig, Artificial Intelligence, 3rd ed., p. 185

are trying to capture the Swedish king, the Swedes are trying to move their king to a corner so that they can win the game, unlike chess where both sides are trying to capture the king of their opponent. This asymmetry makes it self apparent when designing the heuristic. For example, a good starting strategy for Muscovites is to form a blockade around Swedish king without sacrificing too many pieces, but Swedes should focus on moving their king to a corner as soon as possible without him getting captured. Discrepancies like this one gives way to interesting questions when designing the evaluation function.

During the development of the program, a good amount of effort and time was spent towards improving and fine-tuning the heuristics. The end result was the following linear evaluation function.

$$v(s) = w_1 f_1(s) + w_2 f_2(s) + f(3)$$

$f_1(s)$ is simply the difference of the number of player's pieces and the number of opponent's pieces in a give board state. $f_2(s)$ is the Manhattan distance of the Swedish king to the closest corner. Since Muscovites try to maximize and Swedes try to minimize this value, this function was designed as

$$f_2(s) = \begin{cases} \delta, & \text{if player is Muscovites} \\ -\delta, & \text{otherwise} \end{cases}$$

where δ represents the Manhattan distance of the king to the closest corner. $f_3(s)$ increases with the number of Muscovite pieces placed on the certain tiles on the board so that they form a blockade around the king to prevent it from escaping to one of the corners. Since this a good starting strategy for Muscovites only, the value of $f_3(s)$ is always 0 when the A.I. plays as Swedes.

3 Theoretical basis of the chosen approach

Minimax algorithm searches the state tree of a game. It does this by recursively calling itself on the children of a given state node which in turn applies the algorithm to its children, and so on. It goes down on a branch until it reaches a terminal state node, evaluates it, and returns the value of the node to its parent. The algorithm, then, proceeds to evaluating the next child the same way. This process is applied to every node in the state tree. When the algorithm comes back to the initial call, it has then evaluated all children of the root. Following that, tt returns the operator with the highest value, assuming that the initial call was made on a *max-node*. Max-nodes are the ones which return the maximum value of their children. They represent the maximizing player is to move, while the min-nodes return the minimum the value of their children and they represent the moves made by tge minimizing player or the opponent.

Given the time constraints and the high branching factor of the game, it was impossible to run a complete minimax search in its pure form. Thus, to meet the constraints, the search was cut off after the algorithm reached a certain depth, 3 in particular. This is where the evaluation function came into play. An

evaluation (or heuristic) function evaluates a node based on the game state that it represents. When the algorithm reaches the cut-off point, it evaluates the node without proceeding down to the leaves, according to the heuristic function provided.

Another tool used for efficiency was alpha-beta pruning. This algorithm utilizes two extra variables – usually called *alpha* and *beta* – to keep track of the best leaf value for the maximizing player (alpha) and the best leaf value for the opponent (beta). The simple idea is that if a path in the state tree looks worse than the best one already processed, that path gets discarded. This is the standard technique for deterministic games of perfect information, which suits the problem at hand exquisitely.

4 Advantages and disadvantages

This section provides a summary of the advantages and disadvantages of the chosen approach and weaknesses of the program.

The greatest advantage of minimax-search is that it enables the A.I. to *think* certain number of moves ahead of the current state of the game. If implemented well with sufficient resources, this number could be as high as 8 for a chess AI that runs on a typical personal computer. The search depth in our case was 3. Given the constraints and compared to some of the AI algorithms such as *greedy* this number definitely represents an advantage. However, it is quite far from 8. Thus, this can be seen as an advantages and a disadvantage at the same time.

Despite using alpha-beta pruning, no branch elimination is gained before the algorithm starts running on depth 1 of the state tree (no elimination between depth 0 and 1), due to the way it was implemented. Cloning the `TablutBoardState` objects everytime when generating new nodes in the game tree gives way to more inefficiencies in the program, because object creation is a relatively expensive task. Not eliminating symmetrical states, especially for first few turns of the game where there are many, and considering all legal moves for each state instead of considering a subset could also be seen as weaknesses.

During the testing phase, `chooseMove` method seldomly showed a strange behaviour of returning a random move instead of computing one. As this has become apparent only few minutes before the submission deadline, it could not be investigated.

The evaluation function `heuristicValue` could be seen as another strong advantage of the approach as a good amount of resources was spent on improving the evaluation function, instead of optimizing the search algorithm.

5 Other approaches considered

The initial approach chosen for this project was to implement a minimax algorithm with a cut-off depth but without alpha-beta pruning. This approach was

used as a stepping stone to see how far the simpler algorithm could go. The maximum search depth reached without pruning was 3. Later, this approach was abandoned in the hopes to reach a higher search depth and the entire program was re-written, this time with alpha-beta pruning, only to discover that it failed to reach this goal. On the other hand, node pruning enabled the program to include more robust and computationally heavy evaluation function.

Another approach considered but not chosen was the *Monte Carlo Tree Search* algorithm (MCTS). MCTS was tempting because it did not rely on heuristics as much as minimax. After some research, however, it was found out that an MCTS algorithm with a thousand playouts in the simulation phase would take more than 60 seconds to return the first of the moves.[2]. Considering the 30-second time limit for the initial move, this approach was dropped quite early in the development process.

Negamax search also some drew some attention. This minimax variant relies on the fact that the value of a state to player A is the negation of the value of that state to player B to simplify the implementation. Since it was discovered too late in the development phase, it is left out to potentially be utilized in the future.

6 Future improvements

As mentioned above, the first improvement to make would be the implementation of the negamax variant to see how much it simplifies the code. After that, the algorithm will be optimized by finding a way to represent child nodes without creating new `TablutBoardState` objects each time. Storing the parent node and writing a method that *unprocesses* moves could be useful for this purpose. Finding a way to start the pruning process at the root node is also in the realm of possible improvements. A database of good opening moves for both sides will be useful and could improve the efficiency drastically for first few turns. This could leave time to explore the higher depths of the state tree within the 30-second limit before executing the first move, which can be evaluated and stored in transposition tables for later use during the game. Another improvement planned is to find a way to eliminate symmetrical states, especially for the first few turns.

More of an experiment rather than an improvement, the MCTS algorithm could be implemented not only to test Murphy's observation on the subject but also to gain valuable insights into the implementation of a different approach and to get a better understanding of this technique.

[2]Murphy & Kehoe, Viking Chess Using MCTS, page 9