COMP424 Project Report

Amelia Dong

April 5, 2018

Introduction

Tablut is a variant of the ancient Tafl games, which requires two players (the *Muscovites* and the *Swedes*) playing on a 9 by 9 board. The Muscovites consist of 16 black pieces, always moving first and seeking to capture the King of the Swedes. The Swedes, with 9 white pieces (including the King), aim to assist the King to one of the four corners without being captured. Any piece can move either horizontally or vertically without passing over other pieces, but only the King can stay at the center or the corners. The capturing is done by "sandwiching" an opponent piece, between two player pieces or between one player piece and either the center or the corner, at the player's move. However, 4 (or 3) black pieces are required to capture the King if it is at the center (or at the neighbors of the center).

Method & Motivation

The overall idea is to first set both players to choose moves randomly. Starting with the Muscovites, I implement an algorithm for them so that the improved Muscovites can always beat those random Swedes. Then, I focus on improving the random Swedes until it can beat the Muscovites that is previously implemented. Repeating this process for three rounds, both players get improved alternatively.

In the final version of my implementation, I manually set up the very first

move for both players. For example, for the Muscovites, the first move is always from (3,0) to (3,1). Then, no black piece can be captured in the Swedes' turn, but no matter how the Swedes move, the Muscovites can capture one white piece at the next move. Similarly, for the Swedes, the first move is from (4,5) to (7,5), if possible. This is due to the fact that the Swedes always lose a piece after their first move (unless the Muscovites choose not to play greedily). Such move makes it plausible to create an open row (i.e. row 7) by sacrificing one white piece. After the first two moves, AI starts choosing moves for both players based on the Minimax algorithm with alpha-beta pruning. Given each state, AI advances three more moves on the board, and picks the best current move outputted by the algorithm.

Note that the initial board is symmetric, so fixing the first several moves has no serious impact. Moreover, the benefits of the Minimax algorithm do not show at the beginning of the game, since it makes decisions purely based on the utility value, not a real "strategy". As the game closes to the end, each move is crucial and must be taken carefully, and it is when the Minimax algorithm plays its role. To make the Minimax indeed work, it is significant to assign an appropriate utility to each terminal state.

In my implementation, the max player represents the Swedes and the min player represents the Muscovites. Based on the previous experience, the utility is set to be

$$\text{utility} = \#\text{ Swedes} - \#\text{ Muscovites} - \text{kingDistanceToClosestCorner},$$

while the maximum or minimum value is assigned if the Swedes or the Muscovites is guaranteed to win, respectively.

Technical Approach

Minimax Algorithm is a popular AI algorithm which uses a simple recursive computation to maximize (or to minimize) the profit (or the loss) for the max player (or for the min player). The recursion proceeds to the cutoff level where all terminal states are evaluated using a pre-specified utility function. These values are then backed up to the current state, while the algorithm makes the best possible decision depending on the player on move. Note that if the opponent plays optimally and uses the same utility function (which are not true in general), then the Minimax algorithm gives the optimal solution.

Alpha-Beta pruning often builds on the Minimax Algorithm, aiming at simplifying the procedure without affecting the results. Instead of exploring every single state as the usual Minimax would do, alpha-beta search prunes away those branches that cannot influence the final decision, given the results from those already explored states. An important observation is that for the alpha-beta pruning, the ordering of nodes matter.

On average, the time complexity for the Minimax algorithm is $O(b^m)$ or $O(b^{3m/4})$, if the alpha-beta pruning is used. ($b = \#$ branching factor, $m = \#$ depth).

Advantages & Disadvantages

Pros:

• The final AI is simple and concise.

• It suits this two-player game perfectly by choosing moves of relatively good quality.

• Instead of elaborating different cases, it reduces the work to simply estimate the utility value.

Cons:

• Due to the large branching factor of this game, the Minimax with alpha-beta pruning cannot go deeper (i.e. depth 4 in our AI) within the limited time, which provides a move that is still optimized locally and therefore less accurate and perfect.

• The utility function is biased since it is set based on my own experience. It then leads to two potential problems:

⋆ the utility I set may be oversimplified and less thorough by overlooking some important cases (since I am not the expert of this game), and

⋆ when this AI is playing against the other AI with the Minimax algorithm, they may not share exactly the same utility functions, which leads to the unpredictability of the results.

Other Approaches

As mentioned previously, I implemented this AI in three rounds.

In the first round, all methods were implemented using solely intuitions and experience. For instance, experience gives that 1) the Muscovites should not always be greedy, in particular when the King is one move to the corner, and 2) the Swedes are prone to be captured (as they have only 9 pieces against 16), but they can be greedy as long as the King is in a safe position. Relevant methods were implemented. Although the resulting AI can beat the random players, it is still too simple and naive. By examining the chosen moves, I found some moves to be entirely meaningless and inefficient.

In the second round, the usual Minimax without alpha-beta pruning was implemented. Unlike the first naive AI which optimized only locally, i.e. focusing on the current move without worrying about the aftermath, the Minimax algorithm outputted a relatively efficient move. However, this version has a drawback: when the

depth was set to be 3, it was unstable as sometimes it timed out, and the algorithm totally crushed when the depth was larger than 3. Since I did not want to compromise with the depth (as in general, the deeper the better move outputted), I added the alpha-beta pruning in the final version of the method.

Future Improvement

The current method is still limited as it only looks three moves ahead. To obtain a move with higher chances of winning, after the cutoff level is reached, we can apply the Monte Carlo simulations which randomize the moves all the way until the end of the game and make decisions based on ample numerical results.

Moreover, as more and more data is collected, the utility function are expected to be improved tremendously (currently the utility function is rather crude in the sense that it does not take all cases into consideration). With some reinforcement learning techniques, the AI can automatically determines the ideal behaviour based on the feedbacks it receives. Gradually, we would expect the move it chooses converges to the global optimum.

References

Russell, S. and Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd edition). NJ: Pearson, pp. 165-169.