Project Report

Simon Zheng
260744353

April 8th, 2018

Introduction

> Tafl games are a family of ancient Germanic and Celtic strategy board games played on a checkered or latticed gameboard with two armies of uneven numbers, representing variants of an early Scandinavian board game called tafl or hnefatafl in contemporary literature.
>
> —*Wikipedia on Tafl games*

Tablut is one variation with Muscovite attackers (black) and Swedish defenders (white) having 16 and 9 pieces (including a king), respectively. The attackers must surround the king and the defenders must move it to a corner. All pieces move horizontally and vertically, as would a rook in chess.

A relatively simple classical style turn-based board game. A minimax algorithm was used as the student player agent.

1 Explanation and motivation

The student player agent uses a minimax algorithm. It is simple, when it is the agent's turn it calls the algorithm on the current board state.

The *pickMove* method will return the best move of all the legal moves by calling $findMove$ on each of them (all the children board states). It gets the value of each of them and selects the move with the best score.

$findMove$ is the recursive version (minimax algorithm). The original caller will always be the max player and we can easily get the min player in the current state with *getOpponent*. The algorithm iterates through all the moves by performing a depth search on them going all the way down to leaf nodes to evaluate them, and "bubble" back up the best value (maximum for max node, minimum for min node). Since it is turn based, the max and min player alternate and with that, the node levels will follow. To "try" a move, the board state is cloned and then the move is processed on it. Then the method is called recursively on it. When it reaches a terminal node or the maximum depth (limited as it is not too efficient), it calls an evaluation function which determines the value or "score" of the node (board state) by simply adding and subtracting points on certain criteria, e.g. number of pieces left, Manhattan distance of the king from corners, proximity of Muscovite pieces to the king. Winning gets the best score and losing the worst. Draw is the same but could be made a worse score.

The algorithm was picked for its straightforwardness, more complex algorithms are limited by the restrictions and coding better ones might be difficult to optimize (though they would certainly end up more efficient and smarter, and many other players will have used them). The evaluation method is not too complex as to save computation time since it is already pretty slow.

2 Theoretical basis

It is a standard minimax algorithm (as seen in class). The algorithm will generate the state tree. Here the root is the current board state and its children are all the possible states resulting from all the possible legal moves.

The algorithm performs a depth-first search for leaf nodes (win or draw states, or when max depth is reached) and runs the evaluation function on these nodes. It evaluates the node based on criteria as mentioned previously.

The values are then returned to their parent. The parent will take the best or worst value depending on whether they are a min or a max node. A max node is a node it is the turn of the player while a min node is one where it is the turn of the opponent, from the perspective of the root (which is always a max node normally since it is the perspective of the original caller).

A max node takes the best move (highest value) and the opposite for the min node. The algorithm assumes optimal play from both players. "Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent."[1]

The values are "pushed up" using this and the root then selects the best move from these values.

The advantages and the disadvantages of the algorithm itself will be discussed next.

[1]Russell, Stuart J. (Stuart Jonathan). Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice Hall, 2010.

3 Pros and cons

With the simplicity of the algorithm comes the lack of sophistication, which means it is not too "smart" compared to other ones. From what was seen in class (slides):

- it is complete on a finite game tree, which is an advantage but the tree will become very large which is a disadvantage with the lack of optimization or smarter search;
- it is only optimal against an optimal opponent, as explained previously, and many players or agents will probably not be optimal (random player is obviously not, and greedy probably neither), so it is not an advantage most of the time. It would only be if the opponent also has the same minimax algorithm, but would also need to have the same evaluation function;
- it also has $O(b^m)$ time complexity which is pretty slow in practice (the depth was limited to 2 after some testing);
- space complexity is acceptable as DFS is used.

So in summary, it is pretty slow and limited, and not optimal against different opponents and the branching factor and depth is too much to be able to get the optimal solution anyway in practice in an actual game with limited time.

4 Other approaches

While not having managed to make it work, $\alpha - \beta$ pruning would have been much, much better in every way as it would have allowed greater depth, even if only slightly more, which could make a relatively large difference. There is also pruning mirror or symmetric moves and states that would allow to avoid unnecessary or inefficient searches and also greatly improve the search time and depth.

Different evaluation functions and criteria were also tried with varying results. This was the most complicated part, finding the right evaluation function, and it could have been done much better. Some would perform better against certain agents in certain conditions, but worse in others, so it was not all that obvious to find one absolutely better. In fact, finding a working evaluation function at all that is efficient enough is the biggest challenge.

Different search depths were also attempted, ranging from 0 to 100 (draw). The algorithm being quite slow limited the maximum search depth to 2 to be "comfortable" and avoid timeouts.

5 Improvements

As mentioned previously, $\alpha-\beta$ pruning, ignoring mirror or symmetric states, better evaluation function and utility, heuristics, more efficient code.

The Monte Carlo method could have also been useful, though with the restrictions and limit I think the best that could be done is randomized Monte Carlo and with only a very few simulations. It is not sure whether such a limited Monte Carlo method would improve enough, but it certainly could.

Other methods could have been hill-climbing and simulated annealing, though I chose not to explore those paths.

Optimizations to the algorithm and code themselves could be made too. Getting all the legal moves is extremely expensive as noted and finding a subset of pieces to move using a function would be smarter. The Java code itself was fairly high level with the API we had but understanding it better could give many benefits, mainly as there are many ways to do certain things and knowing which way is *actually* more efficient "under the hood" would help. Different implementation of DFS itself might also help (the ordering for example). Which variables and values to pass, which to keep, etc. Maybe even some micro-optimizations (though limited in Java). Creating our own structures and functions that are optimized for our algorithm too, etc.