COMP 424 Project - Tablut AI

Ivelin Bratanov
ivelin.bratanov@mail.mcgill.ca
260535395

***Abstract*— This paper explores the implementation of an AI agent to play the Tablut boardgame, as the final project of the COMP 424 AI course at McGill. The strategy implemented uses the minimax algorithm with alpha-beta pruning to traverse the game state tree. The evaluation function used takes into account several variables such as both players' current piece count, and king distance from the goal. The agent achieves a 100% win rate against the baseline agents (random and greedy) as well as simpler agents with other heuristics.**

I. INTRODUCTION

The field of artificial intelligence continues to progress in parallel with increased computing power and resource availability, and we've reached a point where powerful game-playing agents can be run even on personal computers. The purpose of this project is to develop such an agent, limited to certain resource and time constraints, while exploring algorithms and game-playing approaches discussed in the COMP 424 course and beyond.

Tablut is a two-player board game - a Finnish variant of the classic Viking game *Hnefatafl*, where the each side has a different goal. The *Swedes* (white pieces) must move their king piece to a corner without it being captured. The *Muscovites* (black pieces) must capture the king by surrounding him and prevent him from reaching any of the corners. The full game rules can be found here.

II. THEORETICAL BACKGROUND

A. Related Work

Much research has been done on AI agents that play turn-based games such as chess, shogi (Japanese chess), Go, and checkers. These agents often rely on variants and even hybrids of α-β search (Chess, Checkers) or Monte Carlo tree search (Go) with huge opening and endgame databases. More recently, multi-layer neural networks with reinforcement learning have become popular and allow AI agents to learn from a huge dataset of games as well as improve with each new game they play. In the case of DeepMind's *AlphaGo Zero*, the agent was able to learn entirely by playing against itself given no domain knowledge except the game rules, and defeat world-champion agents for chess, shogi, and Go [1].

In 2007, Philip Hingston [2] From Edith Cowan University developed an AI agent to play Hnefatafl (and some of its variations). He uses a neural network and evolutionary algorithm to develop an evaluation function and evolve a suitable agent (over 300 generations). On the other hand, in 2016 Declan Murphy [3] from the Institute of Technology, Carlow, used Monte Carlo Tree Search to develop an AI agent to play Tablut, Hnefatafl, and other variations. Albeit limiting MCTS to 1000 playouts, Murphy reports that for Tablut, the average first-move turn time is 1 minute and 8 seconds, which greatly exceeds our 2s general restriction (and even our 30s first-move restriction).

Additional discussions and blog posts can be found suggesting strategies for Hnefatafl and Tablut AI agents, most notably by Jay Slater [4] who developed an AI to play Hnefatafl called *OpenTafl*. *OpenTafl* uses minimax with alpha-beta pruning, along with what he refers to as "extension search" - more commonly known as "quiescence search" and explored, among many others, by Hermann Kaindl in 1983 [5] and T.A. Marsland in 1986 [6] for chess. Interestingly enough, Slater seems to be organizing a Hnefatafl tournament in 2018 (more info here) and I have contacted him to know more about how to participate.

Tablut is a game for which the rules vary greatly across different sources, and even the variation of the game used for this course seems to differ from most descriptions found online. Additionally, Tablut and its variations are relatively obscure when compared to games like chess, checkers, and Go, and thus resources were limited in terms of works related specifically to this game.

B. Algorithms Used

This section contains a brief technical overview of the algorithms used in this approach. For a more detailed explanation, please see the Articial Intelligence book by Russel & Norvig [7].

1) Minimax: This algorithm, commonly used for turn-based games, takes into account that we only have control over every other layer in the game search tree, and assumes an optimal opponent. It consists of expanding the entire search tree until terminal states have been reached or until a maximal "cutoff depth" has been reached. Utilities for these terminal nodes are then computed via an evaluation function with which to determine the "value" of a particular game state (or node in the tree). An effective evaluation function is critical to the success of the minimax algorithm. At each min node, the lowest value among the children is saved, and at each max node, the highest value is saved. This attempts to mimic the game dynamic where each player tries to maximize their own score, while their opponent tries to minimize the player's score. The operating assumption is that both players play as optimally as they can.

2) α-β Pruning: This is a standard pruning technique for games where perfect information is available (such as in Tablut), that allows the size of the game state tree to be reduced by removing branches where the path looks worse than what we already have (and thus saving computation resources which can be allocated to searching deeper in the tree).
α-β pruning behaves like minimax in the sense that it returns the same "best moves" that minimax would.

III. MOTIVATION

A. Algorithms

Despite the large branching factor of Tablut, motivation for using minimax with α-β pruning came from its popularity and successful results for other games such as chess, where the branching factor is also quite large. α-β search runs into trouble when the branching factor is large, Monte Carlo tree search was also considered, though α-β search was ultimately prioritized due to the fact that MCTS converges slowly and our time constraints for move selection were quite restrictive. Murphy's results seem to support the view that MCTS would only be effective with more flexible move selection time. Due to time (and computation resource constraints), using neural networks to develop an agent was not attempted.

In terms of implementation, minimax (with cutoff depth 3) was initially implemented in the minimaxDecision() and minimaxValue() methods. minimaxDecision() returns a specific move, given a set of move options and a specific board state. It sets the maximal depth, then for calls minimaxValue() on each available move, recording the value of each possible moving and returning the move with the highest possible value.

minimaxValue() would initially search through the entire tree until the cutoff depth was reached, which would allow the agent to search to a depth of 3 (where move timeouts would occasionally occur). Afterwards, α-β pruning was added to the minimaxValue() method, which, despite not allowing the search depth to increase (many move timeouts would occur at depth 4), prevented the majority of the timeouts which occurred with pure minimax at depth 3. Thus, the minimaxValue() method is a sort of hybrid between minimax and α-β, all in one method, although the standard algorithm for α-β is used. minimaxValue() looks at successor states for a given state, recursively calls itself on successor states and returns if the conditions for tree-pruning are met. Otherwise, (in the worst-case scenario) it will go through the entire tree, up to the specified maximum depth. When a true terminal state is reached, a loss is given a very large negative value, a win is given a very high positive value, and a draw is considered neutral, giving it a value of 0.

Additionally, a specific check in minimaxDecision() prevents turn timeouts (which would cause a random move to be selected and in certain cases could cost the player the game). If we approach 2s in terms of time since move selection started, the agent simply returns the move with the highest value found by α-β search thus far.

getSuccessors() is a helper method which retrieves all successor states for a given board state by applying all legal moves to clones of the current board state.

B. Evaluation Function

In order to be effective, an evaluation function must return a value which accurately represents the current game state and allows it to be compared to other game states from the perspective of the current player and determine which is more (or less) favorable for that player.

When it comes to assigning a specific value to a given game state, there are many variables to consider, and one could potentially develop an incredibly complex heuristic. However, there is always a trade-off to be made in terms of evaluation function complexity, and the general consensus seems to be that a simpler evaluation function (which allows deeper search) gives better results than a more "accurate" but computationally-heavy evaluation function which restricts search depth.

Therefore, a relatively simple heuristic was chosen, using only some of the variables which could be considered when evaluating the board state. Other variables considered are discussed in the Alternate Approaches section. Additionally, it was important to assign weights to each of the variables considered in the evaluation function, as it is difficult to determine exactly which variable best represents the "value" of a game state. This is also especially important when considering that there are two sides of the game with different goals, and also that agents would likely benefit from playing more aggressively as the game progresses in order to attempt to reach a win and avoid a draw. Thus, the heuristic varies from side to side, and weights are adjusted as the game progresses.

In the end, the heuristics (and variable weights) chosen were the ones that, all else held equal, defeated the other considered heuristics.
It is interesting to note that there are multiple "tactics," outlined below, which could be adopted by the agent, and the choice of heuristic would determine which tactic is adopted. Tactics are outlined as follows:

- Long-term: attempting to create a barricade around the Swedes (from the Muscovite side)
- Short-term/offensive: choosing the moves which capture as many opponent pieces as possible
- Defensive: Avoiding moves that create opportunities for your opponent to capture pieces, blocking corners (Muscovites), or avoiding moving the king (Swedes)
- Well-rounded: attempting to consider as many variables as possible in terms of piece positions and piece count difference

Variables used are the following:

- PD: piece difference count, calculated as player count - opponent count
- KM: a function of the minimum number of moves for the king to reach any of the 4 corners

- PC: the number of Muscovite pieces in the closest position diagonal from each corner (determined to be key in protecting the corner)

The final functions used are implemented in MyTools.java and illustrated below:

Heuristics for the Swedes:

First 40 moves:

$$boardStateValue = PD + KM$$

Moves 41-70:

$$boardStateValue = PD + 2 \times KM$$

Final 30 moves:

$$boardStateValue = PD + 3 \times KM$$

Heuristics for the Muscovites:

First 40 moves:

$$boardStateValue = PD + PC - KM$$

Moves 41-70:

$$boardStateValue = 2 \times PD + PC - 1.5 \times KM$$

Final 30 moves:

$$boardStateValue = 3 \times PD + PC - 1.5 \times KM$$

IV. PROS & CONS OF CURRENT APPROACH

A. Advantages

α-β search allows the agent to look 3 moves ahead, allowing the agent to often outwit human Tablut players, and always defeat greedy and random agents.

Additionally, the heuristic is computed very quickly, and encompasses other, more specific and computationally-heavy heuristics (described below). Preventing timeouts prevents any random moves from being returned, which could cost the player the game. Additionally, the heuristic used takes into account many different variables and more effectively evaluates the game state than simpler heuristics which only take into account one variable. This heuristic was shown to beat simpler heuristics.

B. Disadvantages

The AI agent may still occasionally not have time to evaluate the full search tree and would thus not return the "best" state in a specific situation. It would detect the impending timeout and return the best move found thus far. Additionally, a search depth of 3 is not very effective when considering how many moves a game like Tablut could have, and that the agent may be encounter the "horizon effect", being blinded to potentially low value states further down the game tree due to its inability to foresee those outcomes using the evaluation function alone.

In general, α-β search is risky because it assumes the opponent will play "optimally", with the same given heuristic. However, in our competition it is likely that everyone's implementation and heuristics will be very different, with some which are "more optimal" than this agent. In these cases, the agent will likely lose to the opponent who has a better sense of true board state value.

Additionally, more efficient agents who are able to reach a deeper search tree depth (and thus "see further in the game" might perform better than the current agent.

Another source of failure that has been noticed is that sometimes, when the player is pitted against an inferior agent who occasionally times out (and thus includes an element of randomness in its move selection), the randomly-behaving agent occasionally wins. Thus, including an element of randomness can throw off the "optimal" expectation made by α-β search.

V. ALTERNATE APPROACHES

Obviously, α-β search agent performs better than the purely minimax agent that was initially implemented, because it's able to avoid most timeouts at depth 3.

A form of iterative deepening was attempted in miniMaxDecision(), where, depending on how far we are in the game, (in terms of remaining piece count) we increase the maximal depth for α-β search, since the assumption is that with less pieces on the board, there are less moves to be evaluated, decreasing the branching factor and allowing us to go deeper in the search tree. After testing, however, it seemed that the decrease in piece count was not enough to allow for search even at a depth of 4 (timeouts would still occur). Thus, this approach was abandoned and search depth was fixed at 3.

The more basic heuristics using one variable alone were considered but all lost to the more complex heuristics. For example, the agent considering only piece count difference was the baseline for a long time during testing.

Other heuristic variables considered:

- Muscovites try to encircle the Swedes by forming a chain of their pieces
- Moves adjacent to enemies are discouraged if enemies can immediately move to the opposite side
- Otherwise moves adjacent to enemies are favored
- Moves that increase board coverage are favored, where board coverage is the total number of possible moves available from the new position.
- Moves that decrease enemy board coverage are favored.

These variables were eventually abandoned as they were deemed either too computationally heavy or less effective at representing the value of the board state.

VI. DISCUSSION

A. Future Improvements

Many future improvements could increase the effectiveness of the agent. Benefit would be drawn from studying the game in more depth. This would make it possible to determine a more accurate heuristic function that takes into account real-world strategies of Tablut experts.

Setting up a database of opening moves could be useful to quickly hone-in on moves that are beneficial to the player at the beginning of the game when the search tree is particularly wide and deep.

In our approach, the Weights were tweaked manually via trial and error. However, the agent would have benefited greatly from using a genetic algorithm to tune the weights of the evaluation function by creating generations of say 50 players, each with their own randomly generated weights, battling them against each other, as well as the random and greedy agents, and adding the top 5 players from each generation to the next one.

Bart Selman, in his Cornell University Artifical Intelligence lecture [8] proposes another way to tune weights in an evaluation function. A neural network could be used to find the best least-squares fit with respect to common winning moves found in a sort of "winning moves database." This could be developed by playing a large number of games (say 5000+) and storing the commonly occurring moves which led to a win.

A multi-layer neural network could also be used to learn from a huge more "general" dataset of game states and their eventual outcome (win, lose, draw). This would allow us to develop a weighted heuristic which takes into account a variety of different game features that might not be evident just by looking at each game state as a human player.

Additionally, a heuristic function which factors in current search depth and "average value" of a branch (as opposed to simply looking at the "best move" in a branch) would allow the agent to determine which branch has more possible positive outcomes for it, allowing it to take the branch that has the most potential wins down the road. This would be especially useful against unpredictable agents.

REFERENCES

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[2] P. Hingston, "Evolving players for an ancient game: Hnefatafl," in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pp. 168–174, IEEE, 2007.

[3] D. Murphy, "Viking chess using mcts," 2016.

[4] J. Slater, "Opentaflvecken: Ai progress," Aug 2016.

[5] H. Kaindl, "Searching to variable depth in computer chess.," in *IJCAI*, pp. 760–762, 1983.

[6] T. A. Marsland, "A review of game-tree pruning," *ICCA journal*, vol. 9, no. 1, pp. 3–19, 1986.

[7] S. J. Russell and P. Norvig, "Artificial intelligence: A modern approach (third edition)," 2009.

[8] B. Selman, "Intelligent machines: From turing to deep blue to watson and beyond," Apr 2018.