Project report
Tablut
COMP 424

Ahmed Jamoussi

April 8, 2018

Date Performed: April 7, 2018
Instructor: Professor Jackie Chi Kit Cheung

Figure 1: Tablut board.

1 Introduction

Tablut is a (9x9) board game from Laponia, described in the 18th century by the Swede botanist Lineo. The game represents a battle between two unequal forces. The goal of the game and the number and properties of the pieces reflect the two powers (Moscowians vs Swede):

- Moscowians (the largest force) must capture the Swede King, initially positioned in the center of the board, avoiding him from escaping by reaching any of the four board corners.
- Swedes (the smaller force) win the game if their King reaches any of the corner squares.

The main task of the project is the design and implementation of an effective and efficient AI algorithm for Tablut. Therefore, the algorithm's goal is to find and infer Playing strategies to defeat the opponent player and then win the game. Of course, the program offer bother scenarios: either playing Moscowians or playing Swedes.

2 Motivation

When implementing AI algorithms for computer games, it is essential to estimate the quality of the game state using an evaluation function. Moreover, building an effective and efficient evaluation function that estimates a none terminal node is a complex task, especially when considering large board games, such as Tablut (9x9). Nevertheless, the Monte-Carlo based technique that emerged in the past decade does NOT require any evaluation function for none-terminal moves. The success of numerous applications of Monte Carlo tree search in complex games, including Poker and Scrabble, demonstrates its potential to significantly improve AI algorithms. Furthermore, the first computer Go program to beat a human professional go player on a (19x19) board developed by Google uses Monte Carlo tree search algorithm to select its action based on previously learnt data and knowledge by machine learning.

3 Program explanation

3.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a search technique that relies on multiple stochastic simulations. The power of MCTS resides in inferring effective and smart strategies from simulating a multitude of random games. The algorithm uses and builds a tree that represents potential future game states. The development steps of the algorithm are the following:

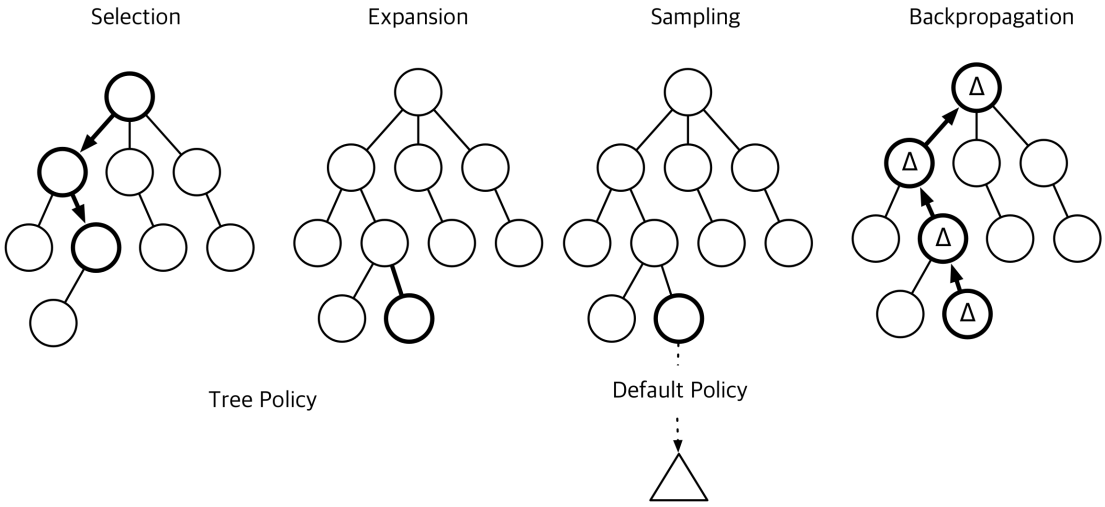

Figure 2: Phases of Monty-Carlo search tree.

3.2 Selection

The next action (move) is chosen based to the statistic stored in the tree nodes. On the one hand, often the selection should be the promising move that leads to a win, this task is called exploitation. On the other hand, due to the uncertainty of the statistic, less promising move should also be selected, which is denoted by exploration. In order to make an adequate balance between exploration and exploitation, Upper Confidence tree (UCT) was the attended statistic for the algorithm:

$$Q(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}} \tag{1}$$

- Q(s,a) Value of taking an action a from states s
- n(s,a) Number of times we have taken action a from state s
- n(s) Number of times we have visited state s in simulations
- c Scaling constant

3.3 Expansion

When a game state is encountered for the first time and cannot be found in the built tree, the game state is added to the MCTS. As a result, for each game simulated the tree is expanded by a new node representing the new state.

3.4 Simulation

During the simulation phase, moves are selected randomly and processed on the board state, selected in the previous step, until the end of the game. If all legal moves are selected with equal probability, then the deployed Playing strategy is often weak and suboptimal. In order to develop a stronger strategy, heuristic functions have been implemented to select the best moves subset from all the legal moves set for a given player turn and a specific board state. It important to notice that use of the heuristic knowledge reduces randomness and increases chances of selecting a promising move. Additionally, such heuristic functions reduce the branching factor of the tree, and therefore allowing the algorithm to simulate a greater number of games on every node during the same time slot. As a result, the algorithm will perform accurately, effectively and efficiently by selecting promising moves. Finally, the effectiveness, efficiency and optimality of the Playing strategy will depends on the "goodness" of the considered heuristic.

3.5 Back-propagation

After reaching the end of the simulated game, there is an update of each tree node, traversed during that game. The visit counts are increased and the win/loss ratio is modified according to the outcome.

Finally, the game action executed by the program in the current game is the one corresponding to the most explored child.

4 Advantages and Drawbacks

Like any other approach, the algorithm based on MCTS method developed for this project shows advantages and disadvantages that will be discussed in this part.
Firstly, it has been proven that the selection and evaluation of actions in MCTS search converges to Mini-Max. The program apply various heuristics to influence the choice of moves which accelerates the convergence of the search to the Mini-max solution. Additionally, MCTS does offer significant advantages in term of space complexity over algorithms that minimize the search space including Alpha-Beta pruning.
Secondly, due to the rollout phase deployed and the use of an effective statistic that makes an adequate balance between exploration and exploitation of nodes, MCTS does not require an explicit evaluation function for none-terminal game states. As such, MCTS is a straightforward algorithm that can be implemented in games without a developed playing strategy or prior specific knowledge about the game. In addition, controlling the number of lines of play allows the algorithm to always return an action in time (2 sec).
Furthermore, the game tree in the implemented algorithm grows asymmetrically as the MCTS method mainly focuses on the most promising subtrees, thus the

search can go much deeper if the branching factor is huge. However, classical algorithms will consider few steps ahead (2 to 3 steps) to select the next action, which might not be enough in games with a huge branching factor to make a suitable selection. As a result, algorithms based on MCTS performs much better in games with high branching factors (Eg. Go) than classical algorithms like Mini-Max and Alpha-Beta pruning.
The algorithm relies on randomness for a meaningful phase; and its search attempts to prune less relevant sequences. The drawback is that the search could not detect a single branch that leads to a loss and therefore does not take it into account. Moreover, even if a play leads to a very significant line of play, it turns out that this latter is overlooked when the tree is pruned, and consequently, not detected by the search.

5 Potential improvements

Although, the implemented program StudentPlayer.java beat optimally RandomPlayer.java and GreedyPlayer.java, the program can still be markedly improved. Hence, in this part we will discuss some of the potential improvements that can be considered in the future on the project.
Monte Carlo simulations used in the algorithm can significantly be improved by the addition of rules and improved heuristics to influence the choice of moves. Hence, the algorithm relies on heavy playouts, which leads to better performance in many games.
The use of other statistics in the algorithm might improve its performance. Basically, the program collects enough information to select the most promising action only after numerous played moves (deep in the search tree), until this specific depth in the search tree the algorithm essentially plays randomly. The use of other statistics indeed RAVE (Rapid Action Value Estimation) might significantly reduce the exploration phase for the considered game. Thus, finding the adequate statistic for Tablut might be a meaningful improvement for the algorithm.
Since, Monte Carlo tree search can be concurrently executed by many processes, then parallel execution is a huge improvement for our program. The following parallel execution might be considered in the future:

- Leaf parallelization: parallel execution of many playouts from one leaf of the game tree.
- Root parallelization: building independent game trees in parallel and selecting the most visited node on the root-level branches of all these trees.

Moreover, the implementation of neural networks and machine learning algorithms will allow us to use data sets of previously played games to train the algorithm by learning about the game strategies. In other words, the more we train the algorithm by playing numerous games against either humans or machines the stronger it becomes.

References:

- Lecture Notes(Comp 424)
- Wikipedia
- Artificial Intelligence: A Modern Approach (Second Edition), Prentice Hall, 2003