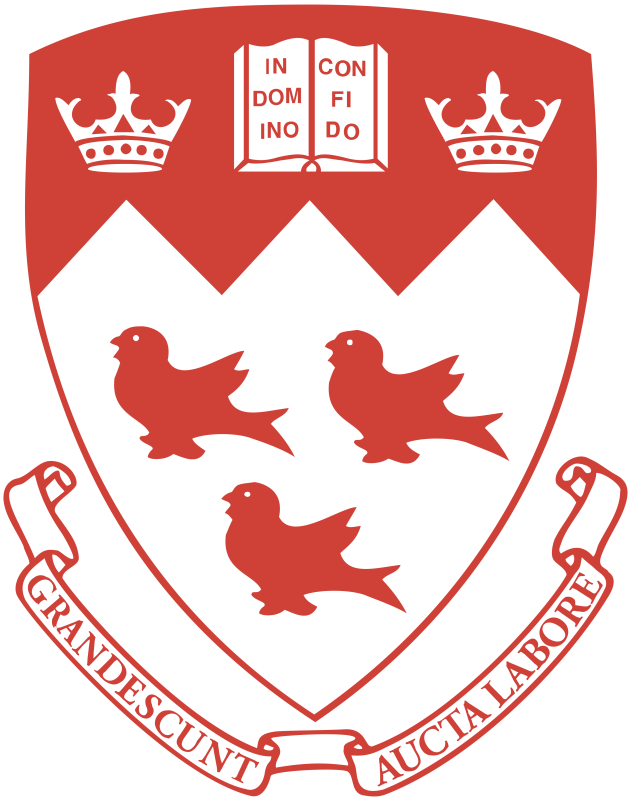
Final Project Report COMP 424 Artificial Intelligence

Nikita Rudin-260809135
McGill University

April 7, 2018



Abstract

This report explains the implementation of an artificial intelligence agent for the game "Tablut". The project was part of the course COMP 424 Artificial Intelligence of Winter 2018 at McGill University.

1. Introduction

The main goal of this project is to implement an AI algorithm, in the context of a game called Tablut, a Finnish variant of the Viking game Hnefatafl from over a thousand years ago.

Tablut is a two player turn-by-turn game played on a 9 by 9 board. One player is the Muscovites (black), and the other is the Swedes (white); the invading Muscovite horde seeks to destroy the Swedish king, while the noble Swedish knights seek to assist the Swedish king to safety.[1]

2. General Approach

The program is based on the min-max with alpha-beta pruning gameplaying technique. This has been chosen as the basic structure of the program as it is particularly suited for agents playing turn-by-turn games with a finite number of possible moves.

Since the Tablut game would require too much computation to search the entire tree, some heuristic is needed to get the value of the nodes at the maximum depth.

The choice a good heuristic is the most critical part and good human intuition is needed in order to understand how to evaluate a given state of the board.

The heuristic is the weighted sum of different scores, each associated to a particular feature of the board state. The features and the weights were mostly chosen iteratively after playing against the previously developed agents. This means that I started with a very basic feature (number of pieces) and then played against that agent. I then tried to see which moves I would have done differently and, most importantly, why. I then added another feature, which would make the agent play in a smarter way. Through trial and error, the agent became progressively better and better.

Quickly, I couldn't beat the agent anymore. Even though it used basic rules, the fact that it could plan every possible move a few turns in advance and not make mistakes was enough to beat me every time. At this point, I started to play each new iteration of the agent against the previous best. Repeating this approach lead to my final submission.

1 Project specification by Kian Kenyon-Dean. Precise rules can be found online.

3. Theoretical basis of the approach

As mentioned before, the basic theoretical method used for this this project is the alpha-beta pruning algorithm[2].

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

The terminal test is the maximum depth (set to 4) of the search. We will see that some minor adjustments were made to this algorithm in order to increase its efficiency in our particular setting.

The first step of the min-max search is done differently from the rest. It gets all possible actions and calls MaxValue if the agent is a *Muscovite* and the MinValue it is a *Swede*. It then compares the returned value with the current best (max for *Muscovite,* min for Swede). Finally the move that provided the best value is sent back.

4. The utility function.

The final utility function is determined for the black player. As such the "Muscovites" try to maximize it while the "Swedes" do the opposite. It combines the following features (other tested features are discussed later):

4.1 Win or loss.

If a state has a winner, it's score can be set without any other consideration and it must more extreme than any other possible score (positive for a black winner and negative for a white winner).

[2] Artificial Intelligence A Modern Approach, Third Edition
Stuart J. Russell and Peter Norvig

4.2 Score based on the number of pieces.

This score is straight forward. It simply defined as

$$S_{pieces} = N_b - N_w$$

Where N_b is the number of black pieces and N_w the number of white pieces.
This makes intuitive sense as it will push our agent to protect its pieces while trying to capture the opponent's ones.

4.3 Score based on the king's ability to escape.

This 2nd score is slightly more complicated in both its intuitive sense and implementation. The basic drawback of approach (2) is that nothing motivates the king to escape or the black player to stop him. This is partially solved by the fact that at some point the search will reach a win or loss situation. Nevertheless, it would be beneficial to encourage the players to move accordingly even when there are no winners at the maximum depth. Since the king must go through an edge before escaping, it seems reasonable for the black player to prevent him from reaching any of the edges and for the white player to look for positions where the king has access to multiple edges at the same time.
A simple solution that seems to work well is therefore to check if there is a piece (white or black) for each direction around the king (above, below, right and left). The score is then computed as the number of direction that are blocked.
The weight given to this score is lower than that of the number of pieces.

4.4 Score based on the king being surrounded

This last feature is an addition to (3).
It is needed because the agent resulting from the previous scores is too defensive. For example, as a black player it will not let the king escape but won't capture it either. This leads to many draws. To encourage it to be more aggressive, a bonus score is given when there are black pieces in the king's neighbor cells. This score has the lowest weight.

5. Practical implementation checks and adjustments

5.1 Maximum depth

The first practical question is how to choose the maximum depth. The constraint is given by the maximum allocated time (2s) between each move. Since the branching factor is high, the time required for an extra step grows quickly. For example, for a branching factor of 40, each step requires 40x more computations. In practice it seems that the branching factor is between 20 and 50. On a standard computer the maximum depth of 4 is easily reached every time, while a depth of five can work in some cases. To use the allocated time to the maximum. I set the maxim depth to 5 if the branching factor is low (<25) and to 4 otherwise. This approach has the issue of being dependent on the computer's processing power, but since an extra level of depth can provide much better results, it is a risk worth taking.

5.2 Timeout check

In order to avoid timeouts, the first step of the search keeps track of the elapsed time and if a timeout is about to happen, it returns the current best move. This move can be suboptimal, but it is still better than a random action. The most probable cause of a timeout is that the max depth was changed to 5, while the processor is not fast enough. To avoid getting this issue at each following step, if the timeout check is positive even once, the processor is categorized as slow and the maximum depth will be fixed to 4 for the rest of the game.

5.3 Efficiency improvements

At every step, if the board state has a winner, the Utility is called without expanding the node further. At the first step, if a move provides a win, it is directly returned.

5.4 Continue playing in case of guaranteed loss

If in a particular state a loss is guaranteed, the min-max algorithm will not provide the optimal way to play since all nodes will have the same value. It would still be useful to play optimally, because the opponent might make a mistake. The selected solution is to rerun the search, but with a lower maximum depth. This way the agent will select the best move, even though it will probably loose at the next step.

5.5 Add a non-deterministic component

The alpha-beta pruning algorithm is completely deterministic, this means that the agent will always respond the same way to a particular state. This is a problem when two of these agents play against each other, since the game will always be the same. Since there are often many moves that return the same maximum score. It would be useful to choose between these moves in a random manner. This was implemented by having a probability of selecting a new move as the new best when its score is equal to the current best.

6. Advantages and disadvantages of this approach.

6.1 Advantages

The first advantage of this approach is it simplicity. The actions taken by the agent can be easily understood and related to the utility function. It also robust to different players. To my knowledge the agents responds well to very different styles of opponents. There doesn't seem to be a simple strategy that would exploit my agent's technique to beat it every time.

Another advantage, is that the utility function is not computationally heavy. This means that the program can run on a slow computer and still respond in a far less than a second (if the max depth is set to 4).

6.2 Disadvantages

Of course, simplicity also becomes a disadvantage when playing against a more advanced AI. I have no doubt that a more complex utility function can be beneficial for this game and provide better results. The fact that my agent responds very quickly is beneficial in a game

played against a human, but in the context of our competition it means that more computations could have been made.

A second disadvantage comes from the min-max algorithm. As it assumes that the opponent will play optimally, it will not try some potentially beneficial moves. This can lead to many games finishing in a draw.

7. Other tested approaches

The main other approach that interestingly failed was to include opening strategies. When playing myself, I noticed that having black pieces in particular cells is highly beneficial to protect the corners. I tried adding a special score that push the agent to pursue these states at the beginning of the game. After a long trial and error period it seemed that the best weight for this score was 0, meaning that the strategy didn't help at all.

I have also considered using a Monte-Carlo Tree search technique, but I quickly abandoned the idea because the random moves wouldn't be representative of an actual game. For example, even if the king had the opportunity to win the chance of it taking this move would be very low.

8. Proposed improvements

8.1 More complex utility function

This improvement is the most obvious one. It would make sense to add more features to the utility function. These can include for example: corner protection, distance of the king from a corner, scattering of pieces on the board etc… The reason this was not done in this project is only the amount of work it would represent to design and tune these features and their weights.

8.2 Learning the weights and/or opening strategies

It would a good idea to use some learning technique to learn the weights of the different features. It would also be great to use a similar technique to learn various opening strategies. These techniques can range from a genetic algorithm to neural network.

8.3 Optimize code efficiency

One last improvement would be to optimize the code to make it faster, thus potentially increasing the maximum depth. First, this can be done by going through the code and removing any redundant checks and calculations. Then the alpha-beta pruning can be optimized by selecting the most promising nodes first. Finally, to go even further, the code can be optimized by moving away from object-oriented programming and implementing as many functions as possible on the most basic level.