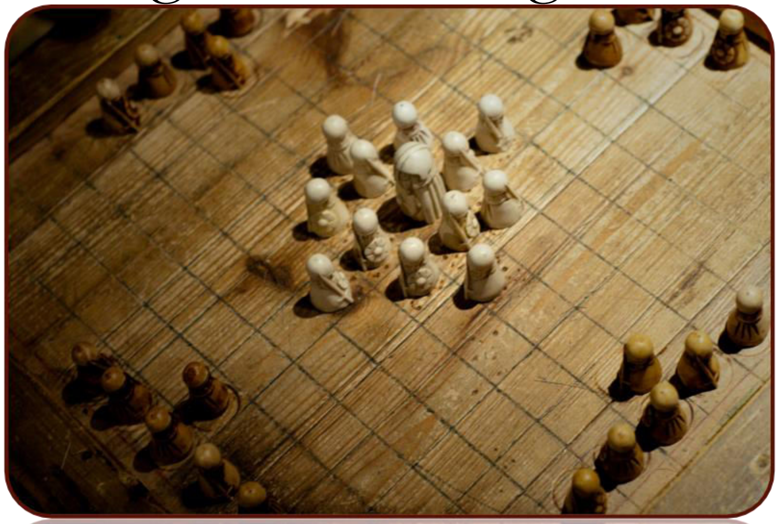
Viking Chess Using MCTS



Research Document

Abstract

Viking Chess (Tafl) is an ancient game played throughout Northern Europe from as early as 4th century B.C, until it was supplanted by modern Chess in the 12th century. Unlike Chess and other zero-sum games, it is difficult to build reliable heuristics for solving its domain space, making traditional algorithms ineffective. Monte Carlo Tree Search (MCTS) has shown promise in areas where traditional algorithms have failed. This makes MCTS an ideal candidate for a Tafl implementation. This document examines Tafl and its regional variants, surveys the current search algorithms for zero-sum games and evaluates their application in relation to Tafl games before finally considering the suitability of the Universal Windows Platform (UWP) in implementing Tafl for Windows 10.

Declan Murphy
B.Sc.(Hons) Software Development

Abstract

Viking Chess (Tafl) is an ancient game played through Northern Europe during the Middle Ages. Unlike Chess and other zero-sum games, it is difficult to build reliable heuristics for solving its domain space, making traditional algorithms ineffective. Monte Carlo Tree Search (MCTS) has shown promise in areas where traditional algorithms have failed. This makes MCTS an ideal candidate for a Tafl implementation. This document examines Tafl and its regional variants, surveys the current search algorithms for zero-sum games and evaluates their application in relation to Tafl games before finally considering the suitability of the Universal Windows Platform (UWP) in implementing Tafl for Windows 10.

Declan Murphy
B.Sc.(Hons) Software Development

Contents

1. Introduction

1.1 Abstract

Viking Chess (Tafl) is an ancient game played throughout Northern Europe from as early as 4th century B.C, until it was supplanted by modern Chess in the 12th century. Unlike Chess and other zero-sum games, it is difficult to build reliable heuristics for solving its domain space, making traditional algorithms ineffective. Monte Carlo Tree Search (MCTS) has shown promise in areas where traditional algorithms have failed. This makes MCTS an ideal candidate for a Tafl implementation.

1.2. Purpose

The purpose of this research document is to study the Viking Chess domain space and establish its variant and rules of play while researching the possible methods for creating a computer player for the game. This document will also examine the requirements for implementing a Viking Chess game on the Windows 10 platform and consider the existing applications of MCTS.

2. Sentiment Analysis

2.1. Viking Chess

2.1.1. History

Tafl games, colloquially known as Viking Chess, is a range of different games played from the 4th to the 12th Century across Northern Europe until it was supplanted by modern Chess (Murray, 1951). Tafl games spread with the Vikings as they travelled throughout Europe and broke into a number of variants in each region – Hnefatafl in Scandinavia, Brandubh in Ireland, Ard Rí in Scotland, Tawlbwrdd in Wales, and Tablut in Laponia (Modern-day: Lapland).

In 1732, during his expedition to Lapland, Carl Linnaeus documented a game (Tablut) played by the native Sami peoples. Linnaeus wrote all of his documentation in Latin and although his notes provide the most complete documentation of a Tafl game, they are somewhat incomplete due to his inability to speak the Sami language and his reliance purely on observation. In spite of this, his notes have become the prime source for the rules of all Tafl games.

2.1.2. Rules

As previously mentioned, the rules used in modern recreations of Tafl games come from the notes of the Swedish Botanist, Carl Linnaeus. The game that Linnaeus described was Tablut, a game still played at the time by the native Sami people of Lapland.

Linnaeus' description of the board and the layout of the pieces is common in other variants of the game, and has been found in older sources of information and archeological finds. From his description we get a clear view of the layout of the board and pieces. The board itself consists of rows and columns of uneven numbers – 7x7, 9x9, 11x11, etc. There is always a king piece in the centre of the board, the central square is referred to as the throne and can only be occupied by the king piece. The ratio of attackers to defenders is 2:1.

In his notes, Linnaeus went on further to describe the rules of Tablut. Though the description is brief and may exclude rules and nuances that the botanist missed in his observations, they still provide unprecedented detail on the rules for Tafl games. His notes give us the following information:

1. All pieces move in straight lines, any number of squares as long as there is no other piece obstructing movement. This is precisely the same movement of the Rook in Chess.
2. All pieces other than the King can be captured by enclosing them on two opposing sides.
3. The King piece is captured by surrounding it on all four sides.
4. The central square and corners of the board can be used to capture pieces.

5. The defenders win when the King escapes to an edge square.
6. The attackers win when they capture the King.

These six rules form the basis of every other modern recreation of Tafl games. In addition to these rules, modern recreationists have tried to determine other possible rules that may or may not have existed during the height of the game's popularity but which provide more balance to the game or resolve commonly encountered situations – such as preventing infinite repetition of a move.

In some descriptions of Tafl games and on some artifacts, the corners play some sort of role. In Tablut, these are referred to as the Burgs. It could have been the case that the goal of the game was not to escape to any edge but to one of these corners. This change provides a more balanced game. Without it, White (the player with the King) is vastly more likely to win than Black.

2.1.3. Variants

Hnefatafl

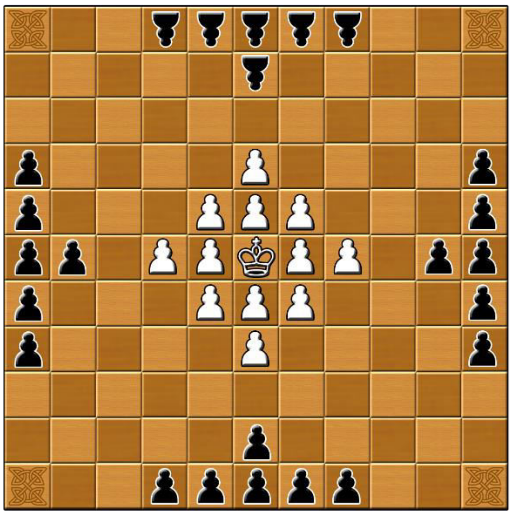


Fig 2. Hnefatafl Layout

History

It is possible that Hnefatafl is the original Tafl game. This variant was popular in Scandinavian countries before and during the Viking Age and followed the Vikings throughout Northern Europe as they raided and settled across the Continent. The game itself is mentioned in various Norse sagas and game boards have been found in a number of European countries, including Ireland and Ukraine.

Layout

Akin to other Tafl variants, Hnefatafl consists of a 2:1 ratio between the attackers (Black) and defenders (White), excluding the White King. This variant is played on an 11x11 board. In total, there are 24 black pawns, 12 white pawns and a single white king piece.

Brandubh

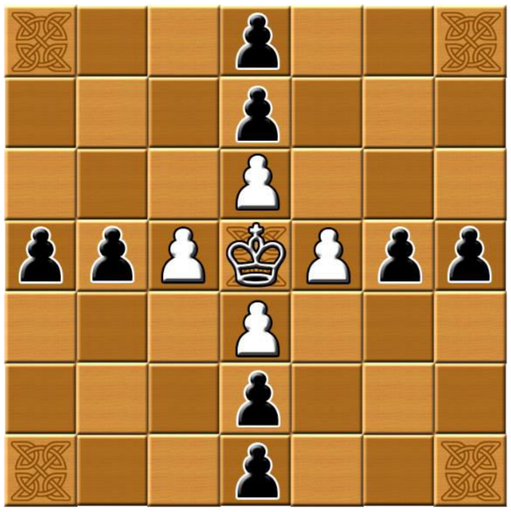


Fig 3. Brandubh Layout

History

Brandubh is the Irish Tafl variant and roughly translates to "Black Raven". It is believed that it came about as a result of contact and trade with the Vikings and dates back to at least the 9th century when the Vikings began to settle in Ireland. Boards have been found in various parts around Ireland, such as in Downpatrick, Waterford and in a peat bog in Antrim. The most famous Brandubh board was found in Ballinderry. The Ballinderry Board consists of a board with 49 holes, with clearly defined corners and a similarly defined centre.

Layout

Brandubh consists of a 2:1 ratio between the attackers (Black) and defenders (White), excluding the White King. This variant is played on a 7x7 board. In total, there are 8 black pawns, 4 white pawns and a single white king piece. It has been suggested from various sources that to win the game, the White player must get the King to the corners of the board as opposed to the sides due to the small size of the board and the low number of pieces.

Tablut

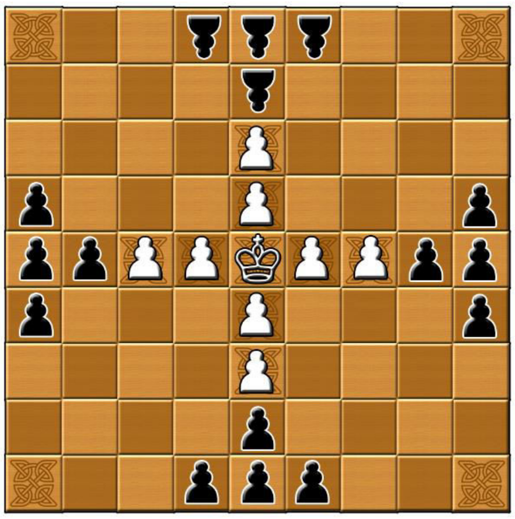


Fig 4. Tablut Layout

History

As previously mentioned, Tablut was documented in some detail by Carl Linnaeus and was still actively been played by the Sammi people at the time. As a result, it is the most reliable source of information on the general rules for Tafl games and is generally used as a base for the other Tafl variants.

Layout

Tablut consists of a 2:1 ratio between the attackers (Black) and defenders (White), excluding the White King. This variant is played on a 9x9 board. In total, there are 16 black pawns, 8 white pawns and a single white king piece.

Tawlbwrdd

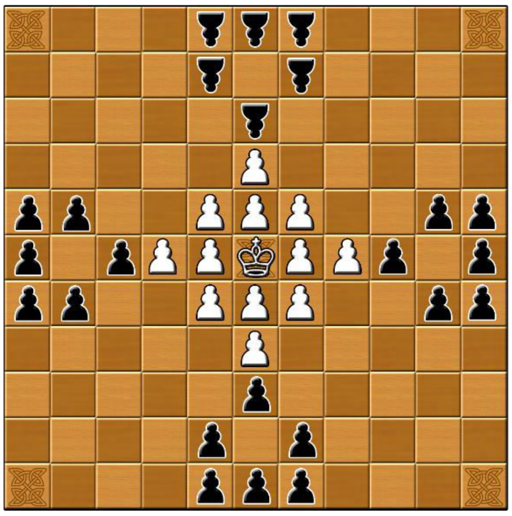


Fig 5. Tawlbwrdd Layout

History

Tawlbwrdd is the Welsh variant of Tafl and roughly translates to "Throw Board". It is mentioned numerous times in texts that date back to the time when Wales was ruled independently. For example, in the book "Ancient Laws and Institutes of Wales", it was stated that the Court Justice was to receive a Tawlbwrdd board from the King which he could never trade. (Hastings, 1841)

Layout

Tawlbwrdd consists of a 2:1 ratio between the attackers (Black) and defenders (White), excluding the White King. The layout is very similar to the standard Hnefatafl layout. This variant is played on a 11x11 board. In total, there are 24 black pawns, 12 white pawns and a single white king piece.

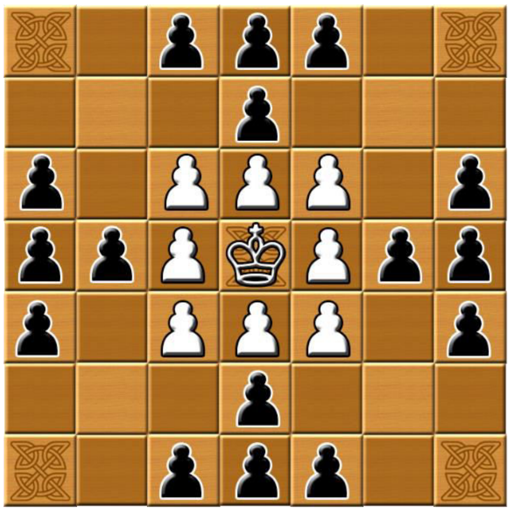


Fig 6. Ard Rí Layout

History
Ard Rí is the Scottish variant of Tafl and translates rougly to "High King". Although there is little historical reference to the board game, there have been various 7x7 board found and collections of pieces that almost match the number expected for the game. This may be the result of Ard Rí being a more modern version of an original Scottish variant.

Layout
Ard Rí consists of a 2:1 ratio between the attackers (Black) and defenders (White), excluding the White King. Like Brandubh, this variant is played on a 7x7 board. In total, there are 16 black pawns, 8 white pawns and a single white king piece. It is worth noting that the high number of pieces on the small board may call for changes to the general rules for playing Tafl games. One suggestion is that the pieces can only move one square at a time, preventing the attacker from overwhelming and boxing in the defenders too easily.

2.2. Two-Player Turn-Based Deterministic Zero-Sum Games of Perfect Information

Turn-based games are games where the order of actions rotates evenly between players. That is, once a player makes their turn, the game moves on to the next player in the game and continues in this fashion until it is the turn of the first player once again with the process repeating until the end of the game.

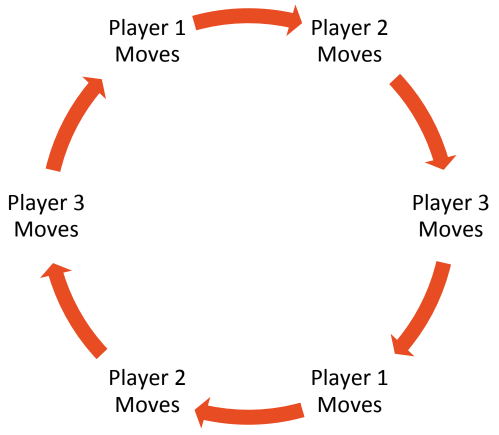


Fig 7. An example of the turn pattern for a three player turn-based game

A zero-sum game is a game in which each player's gains are balanced exactly to the losses of other players. By adding the total gains and subtracting the total losses of players, the sum will always be zero. The table below shows how this is true for the game of chess:

Outcome	Black	White	Sum
Black Wins	+1	-1	0
White Wins	-1	+1	0
Draw	0	0	0

Fig 8. Sum Table for Chess

A Deterministic game is one in which the result of a player's actions leads to entirely predictable outcomes. This is opposed to a game in which a player's actions lead to chaotic outcomes. An example of a deterministic game is Chess where the rules do not allow for variations in the outcome of a player's move. Poker, on the other hand is indeterminate as luck plays a large role in the outcome.

A game of Perfect Information is one in which all previous actions are known by all players. In other words, each player in a game of perfect information is completely informed of the events that have occurred previously when making a decision.

Tafl games can be described as two-person zero-sum games of perfect information i.e. A game of Tafl is played between two players moving alternatively with each player knowing all previously made moves and the gains/losses of each player are exactly balanced against the gains/losses of the other.

2.3. Searching in Two-Player Zero-Sum Games

2.3.1. Introduction

In two-player zero-sum games, players take turns making their moves. To represent this, we can create a game tree to describe the moves made by each player at each stage of the game until a winner is determined.

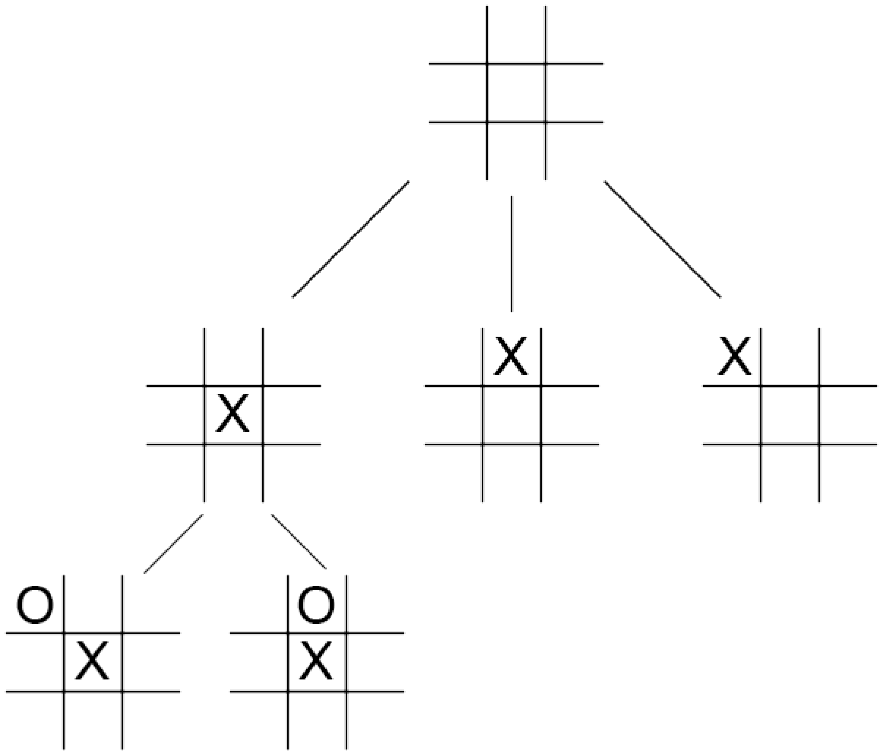


Fig 9. An incomplete game tree for tic-tac-toe

A simple algorithm to calculate the winning move can be created as follows:

1. From the current state, calculate all possible moves that can be made.
2. For each of these moves, expand the tree to include all possible moves for the other player.
3. Continue the expansion until the game reaches a win state for the required player.

This algorithm works to find the best move for a given player at a given game state. However, the algorithm is very inefficient when the number of possible moves that can be made by each player increases exponentially at each state and where the game tree goes several levels (plies) deep before a win state is found, such as in Chess. Even in terms of a simple game of Tic-Tac-Toe, the game tree will be too large and thus, the performance cost will be too high. (Strong, 2011)

Considering this limitation, a better method for finding the best move for a player at a given state needs to be determined. An algorithm that cuts down on the size of the game tree or otherwise reaches a win state quicker is required.

2.3.2. Minimax Search

Introduction

The Minimax Search algorithm offers a possible solution to reducing the performance cost of a search for the best move. In Minimax Search, the entire tree is searched to a specific depth as specified at run-time. This allows for searches to be limited and introduces the possibility of multiple skill levels for the computer player with shallow depths representing low skill players and greater depths representing high skilled players at the cost of time and performance.

Evaluation Function

As the goal of the Minimax Search algorithm is to reduce the time and performance cost associated with generating all possible moves to find the best move from a given position, it is essential that there exists an evaluation function f(P) that can return an approximate value for the positions at the specified depth.

The construction of this evaluation involves an analysis of the given game's rules followed by an extraction of possible evaluation principles. In chess, for example, there are a handful of principles that can be made about the game to assist in evaluating a given position. A handful of these are detailed below:

1. There is a relative value for each piece in chess. This is assigned based on its relative strength in potential trades but may also be affected by the state of the game or position of the piece in question. Traditionally, the standard valuation for chess pieces is detailed in *Fig 8*. Note that there is no standard value for the King as it cannot be traded or captured over the course of a game. For computer chess programs, a very high relative value is often assigned to the king to aid in evaluation – demonstrating that the loss of the king outweighs any other deliberations.

Symbol	p	n	b	r	q
Piece	Pawn	Knight	Bishop	Rook	Queen
Value	1	3	3	5	9

Fig 10. Chess Piece Relative Value

2. Pawns that are isolated, doubled or backwards are considered to be weak.
3. Rooks that are on an open file are considered more valuable due to the greater mobility. In general, the side that has greater mobility has a greater chance of winning.
4. Kings that are unprotected and exposed to the enemy are considered a weakness.

(Shannon, 1950)

Using these and other similar principles, a reliable albeit crude evaluation function can be constructed in which the total value of White's pieces are weighted against the total value of Black's pieces with any other considerations being given an appropriate value and added to each side's sum. In this way, the benefiting player of a given position can be determined to some degree of validity without traversing to a terminal state.

Algorithm

With the Minimax Search algorithm, it is assumed that each player will play their best possible move. Each possible move is assigned a value based on the likelihood that a move will result in a win, loss or draw based on an evaluation function.

When performing a Minimax Search, the player making a move is referred to as the MAX player, with the opponent being referred to as the MIN player. The goal of the MAX player is to choose moves that provide the highest value while the opponent seeks to minimize this value.

When searching to a specific depth, the next step after evaluating the positions at that depth is to propagate those resulting values back up the tree, allowing the maximizing player to select the best move to make from their given position.

```
Minimax(Position pos, Player player, Depth depth, Depth maxDepth){
   if(pos.gameOver() || depth == maxDepth)
       return pos.eval(player);

   bestMove = NULL;
   if (pos.currentPlayer() == player)
       bestScore = -∞;
   else
       bestScore = +∞;

   for (move in pos.generateMoves()){
       newPos = pos.move(move);
       score = Minimax(newPos, player, depth+1, maxDepth);
       if (pos.currentPlayer() == player):
              bestScore = score;
              bestMove = move;
       else if (score > bestScore):
              bestScore = score;
              bestMove = move;
       else (score < bestScore):
              bestScore = score;
              bestMove = move;
   }
   return bestScore, best Move
}
```

Fig 11. Minimax Algorithm Pseudocode

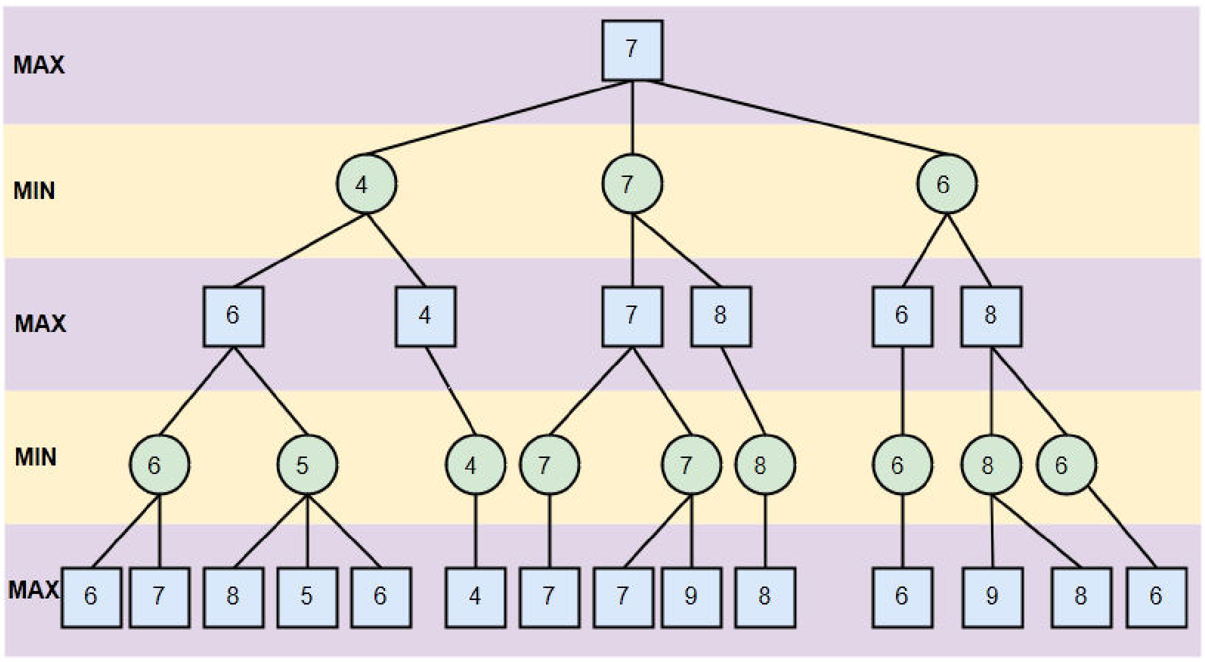


Fig 12. Minimax Game Tree

Fig 9 shows an example game tree that is 4 plies deep. Nodes drawn with a square represent the current (maximizing) player while those drawn with circles represent the minimizing player. The values in each node are obtained from the evaluation function.

Disadvantages

While the Minimax algorithm reduces the time and performance costs of a thorough search of the game tree, it still suffers relatively high costs in situations where the game tree has a high branching factor. Additionally, the reliance on constructing an evaluation function using consistent heuristics is difficult, especially in games such as Go where the best strategy to take at a given position is more esoteric

2.3.3. αβ Pruning

Introduction

αβ Pruning is an expansion upon the Minimax algorithm that seeks to reduce the time it takes to find an optimal move by cutting off branches of the game tree that appear to be disadvantageous to the player. It does this by searching the game tree until at least one move has been found that is worse than an already inspected move. These moves require no further evaluation. The algorithm returns the same move as the Minimax algorithm but cuts away branches that will not impact the final decision.

Algorithm

Since the αβ pruning algorithm is an expansion upon the Minimax Search algorithm, the functional make-up of the algorithm is much the same. However, with αβ pruning, we define two variables α and β to hold the values that will determine any cut-offs with α holding the maximum value found so far and β holding the minimum value found.

By removing branches of the tree for examination, a deeper search of the remaining tree can be performed in the same amount of time that it takes for the minimax tree to find the same moves. What this means is that the effective depth of the search can be reduced and thus the time it takes to find the optimal move can also be reduced.

```
alphaBeta(Position pos, Depth depth, Int alpha, Int beta, Player player){
      if(depth == 0 || pos.gameOver()){
            Return pos.eval();
      }
      if(pos.currentPlayer() == player){
            best = -∞;
            foreach (child : node){
                  best  =  max(best,  alphabeta(child,  depth-1,  alpha,  beta,
FALSE));
                  alpha = max(alpha, best);
                  if(beta <= alpha){
                        break;
                  }
            }
            return best;
      }
      else {
            best = ∞;
            foreach(child : node){
                  best  =  min(best,  alphabeta(child,  depth-1,  alpha,  beta,
TRUE));
                  beta = min(beta, best);
                  if( beta <= alpha){
                        break;
                  }
            }
            Return best;
      }
}
```

Fig 13. αβ Pruning Algorithm Pseudocode

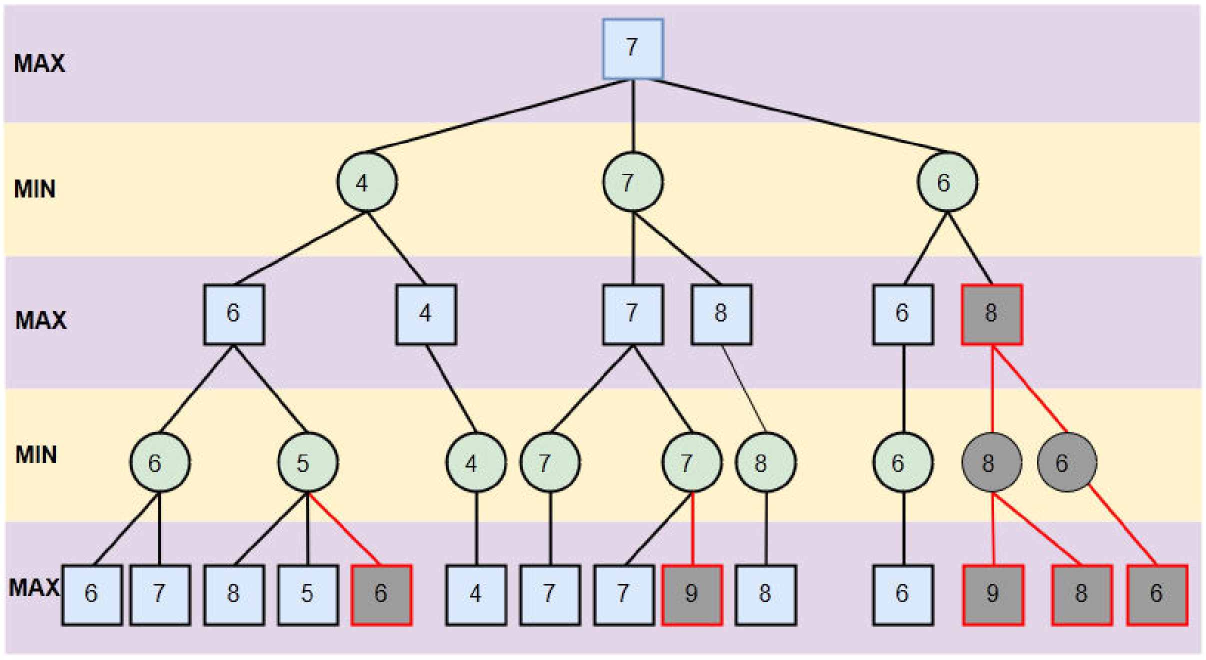


Fig 14. αβ Pruning example

Fig 10 shows an example of a game tree using αβ pruning. The square nodes represent positions where the maximizing player makes a move while circle nodes represent positions where the minimizing player makes a move. The greyed-out nodes are subtrees that have been cut-off. Subtrees are pruned when the value of a subtree is shown to be the same or worse than an already evaluated subtree. In the case of the above tree, the minimizing player prunes off the right-most subtree because it represents the worst outcome for the minimizing player in comparison to the other subtrees. Similarly, a couple of leaf nodes are pruned for the same reason.

2.3.4. Monte Carlo Tree Search

Introduction

Monte Carlo Tree Search (MCTS) is a more recent method for finding the best move for a given game. MCTS is a best-first search algorithm using Monte-Carlo simulations. In comparison to the aforementioned algorithms, MCTS does not require a heuristic evaluation function and is instead based upon randomized explorations of the game tree. The algorithm is of particular interest in areas where creating a concrete evaluation function is difficult or time-consuming, such as in the game of Go. The MCTS algorithm first appeared in 2006 in a number of different forms. Coulom first presented the MCTS algorithm in the program CRAZY STONE which went on to win the 9x9 Go tournament at the 11th Computer Olympiad. Then, Kocsis and Szepesvári created a variant called UCT that used the Upper Confidence Bounds (UCB) algorithm as a base.

Algorithm

MCTS comprises two components that are strongly coupled. The first component is a shallow game tree structure that determines the starting moves for the algorithm. The second component are deeply simulated games.

The structure of the algorithm is as follows. Each node represents a given position in the game. The node will contain at least two pieces of information: the current value of the position and the number of times it has been visited. Typically, MCTS starts with a game tree consisting of only a root node and progressively builds up a game tree in memory using the results of past simulations becoming increasingly better at estimating the values of the most favorable moves. (Chaslot, 2010)

The algorithm has four main steps:

1. **Selection:** The game tree is traversed from the root node until a leaf node is discovered.

2. **Expansion:** A new node is added at to the tree.
3. **Simulation:** During this step, moves are played in a self-play style until the game reaches a terminal state.
4. **Backpropagation:** The final step involves propagating the results of the simulation stage back up the previously traversed nodes.

```
while(timeLeft > 0){
        currentNode = rootNode;

        // traverse the tree
        while(currentNode != null){
                prevNode = currentNode;
                currentNode = select(currentNode);
        }
        // add a node
        prevNode = expand(lastNode);

        // Simulate a game
        result = simulateGame(lastNode);

        // Backpropagate Result
        currentNode = lastNode;
        while(currentNode != null){
                backPropagate(currentNode, result);
                currentNode = currentNode.parent();
        }
}
return bestMove();
```

Fig 15. MCTS Algorithm Pseudocode (Chaslot, 2010)

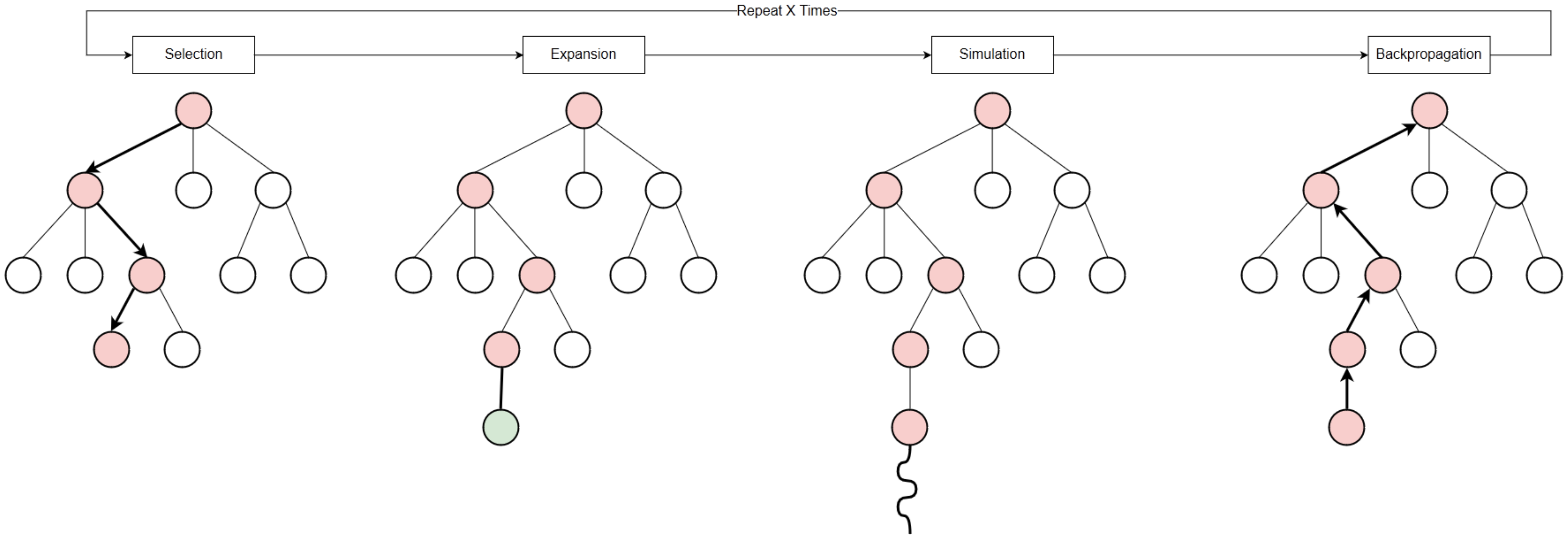


Fig 16. Monte-Carlo Tree Search Outline (Chaslot, 2010)

In Detail

In this section, each of the four steps of the MCTS algorithm will be explained in greater detail. For each step of the algorithm, the strategies available for implementing them will be examined. These strategies have been derived from previous research and implementations of MCTS.

Selection

Selection works in the following manner. From the root, we apply a selection strategy recursively until a position is found that is not yet a part of the tree. This selection strategy aims to control the balance between exploitation and exploration. For instance, one task of the selection strategy is to seek out and select the move that offers the best results thus far. This is exploitation. However, moves that show less promise must still be tried as a result of the ambiguity inherent in the evaluation. This is exploration. This balancing between exploitation and exploration has been previously examined in respect of the Multi-Armed Bandit (MAB) problem. The MAB problem is concerned with a gambling machine in which a player seeks to maximize the reward they gain from the machine. At each step, the player can select only one of several arms of the machine, which returns a reward. The reward is usually distributed randomly. The Selection step of the MCTS algorithm can be viewed as a similar problem for a particular node – the next move to play must be selected which will return a random reward (the result of one simulated game). Knowing the previous results, the selection strategy seeks to find the best move. There have been numerous selection strategies developed for MCTS. These include UCT and PBBM.

- **UCT**

  In 2006, Kocsis and Szepesvári suggested the UCT (Upper Confidence bounds applied to Trees) strategy which is easy to implement and is common across many programs. UCT alters the UCB (Upper Confidence Bounds) algorithm originally developed for MAB problems and works as follows:
  - $Q(p,m)$ is the average reward gained after making move m from position p
  - $n(p,m)$ is the number of times that move m was selected from position p
  - $n(p)$ is the number of times position p is discovered.

  $$\pi_{uct}(p) = argmax_m Q(p,m) + c\sqrt{\frac{\ln n(p)}{n(p,m)}}$$

  (Kocsis & Szepesvári, 2006)

- **PBBM**

  Also in 2006, Rémi Coulom proposed the Probability to be Better than Best Move (PBBM) selection strategy for the Crazy Stone program. The idea behind this algorithm is to select nodes based on its probability of being an improvement over the current best move. (Coulom, 2006)

  Making the assumption that each node has an estimated value of μ_i with a deviation of σ_i^2 and nodes are ordered such that $\mu_0 > \mu_1 > \ldots > \mu_N$, moves are selected with a probability relative to:

  $$U_i = \exp\left(-2.4\frac{\mu_0 - \mu_1}{\sqrt{2(\sigma_0^2 + \sigma_i^2)}}\right)$$

  The value of -2.4 is a constant selected under the assumption that the values obtained will match normal distributions. The values for μ and σ^2 are obtained during the backpropagation stage. For each node i, μ_i is the opposite of the μ value of the child node with σ_i^2 being its deviation.

Expansion

As it is impractical for the entire game tree to be stored in memory, a strategy for expansion is required that determines for a given node whether it should be expanded by storing one or more of its child nodes in memory. A common strategy is to add a single node for every simulated game that corresponds to the first unvisited node. Another strategy would be to expand the game tree to a specific depth before beginning the search.

Simulation

For the Simulation stage, there are two possible strategies to choose from. The first strategy is to play out the simulated games using entirely random moves. Alternatively, a pseudo-random approach can be taken if suitable heuristics can be created for the game. For instance, the pseudo-random strategy could take possible capture opportunities into account or favor certain established patterns over others.

It is important to balance the trade-offs that come with using a heuristic analysis with MCTS as opposed to a more random approach. As the heuristic function becomes more complete and effective, the computation time for the algorithm increases and thus, the number of simulations that can be carried out per second will decrease. If the simulation strategy allows for too much randomness however, there is a risk that the results of the simulation will be poor and as a result, the playing level of the MCTS program will decrease. However, if the strategy ends up selecting the same moves for a given position too often, there will be too much exploitation. This will result in the search space becoming too selective causing the simulation to become prejudiced and once again, the playing level of the MCTS program will decrease.

As a result of these trade-offs, creating a reliable simulation strategy becomes problematic. A possible solution to this problem is to create a number of different simulation strategies and pit them against one another. For example, if a program using simulation strategy A is consistent in defeating a program using simulation strategy B, it can be assumed that simulation strategy A is the superior strategy. (Bouzy & Chaslot, 2006)

Backpropagation

In the final step of the MCTS algorithm, the results of the simulated game is propagated back through the parent nodes of the leaf node from which the simulation was run. The results of the simulated game will be positive (+1) if the game resulted in a win, negative (-1) if the game resulted in a loss, or 0 if the game resulted in a draw. A backpropagation strategy is required to compute the value for nodes. A popular approach is to use an averaging strategy that took the average of the results of all simulated games from the node. (Kocsis & Szepesvári, 2006)

2.3.5. Evaluation

Of the aforementioned search algorithms, only αβ Pruning and MCTS offer a possible solution to creating a reliable computer player for Viking Chess. Minimax Search would be feasible but the time it would take to generate moves due to the high branching factor of Viking Chess rules it out as a practical option. Equally, the αβ Pruning algorithm may be problematic to implement if the aim is to achieve a computer player with a high degree of skill. This is due to the difficulty that arises in establishing quality heuristics for the algorithm's evaluation function.

On the other hand, MCTS is an appealing option due to its lack of reliance on heuristics to evaluate a given position. The skill level of the MCTS algorithm also has room for improvement by employing a heuristic method for choosing moves during the simulation stage. It is also possible to reduce the computational cost of the algorithm by storing previously visited positions in a lookup table and referencing the table when a position is reached that matches a previously discovered one.

The disadvantage of MCTS over Minimax and αβ Pruning is that the stages of the MCTS algorithm will require the establishment of specific strategies. If these strategies are poorly chosen or implemented, the overall effectiveness of the algorithm will be greatly diminished. However, one possible method for overcoming this risk is to create a number of strategies for each stage and setting them against one another from within the domain space of Viking Chess. The results of these experiments should result in a clearer picture of which developed strategy is the best.

In particular, attention needs to be paid to the selection strategy for choosing nodes as this seems to be where the greatest benefit can be gained. The UCT selection strategy is a popular choice and has shown to be applicable to a number of different domain spaces. Alternatively, the PBBM strategy has shown success in the area of computer Go but may require significant adjustments to be applicable to the domain space of Viking Chess.

3. Technologies

3.1. Universal Windows Platform

The Universal Windows Platform (UWP) is the result of Microsoft's efforts over the past number of years to unify their multiple devices. The need for unification arose due to the disparate development base and user experience across Microsoft platforms, requiring developers to target individual platforms (Such as Windows Phone, Desktop, Xbox) or develop several versions for a single application and relegating users to different experiences across Microsoft devices.

The UWP makes it more convenient for developers by providing a single API, application package and common store to target any Windows 10 devices. The effect of this is that a developer can focus their efforts towards targeting specific devices rather than concerning themselves with differences in the underlying operating system. The common application package additionally provides a dependable installation mechanism that ensures seamless deployment and updating of apps. The shared store further unifies Windows 10 devices by allowing developers to submit their app to a single store and make it available across all families of device or only specific ones.

In addition to these benefits, the UI elements for UWP applications are responsive and adapt to the resolution of the device the application is running on automatically. Additionally, they are designed to work with multiple types of input including but not limited to keyboard and mouse, touch, gamepads and stylus pens.

3.2. XAML

The UWP relies on the Extensible Application Markup Language (XAML) for the creation of User Interfaces. XAML is a declarative language developed by Microsoft that can initialize objects and set their properties using a hierarchical structure. XAML is a component of the Windows Presentation Foundation (WPF). WPF is a feature of the .NET Framework that deals with the visual presentation of applications.

WPF uses XAML for developing advanced UIs in markup instead of directly through the programming language. Where there is a need for code behind the scenes, a code-behind file needs to be created with the same name as the XAML document with the addition of the language's extension. For example, if the XAML document is named MainPage.xaml, the code-behind file will be named MainPage.xaml.cs if programmed in C#.

XAML applications can be created in Microsoft Visual Studio or Microsoft Blend. Microsoft Blend is a UI design tool for developing UIs for applications. It is built as an interactive What-You-See-Is-What-You-Get (WYSIWYG) front-end.

3.3. HCI (Human-Computer Interaction)

As the final application will be built as a Universal Windows Application for the Windows 10 family of devices, it will be important to design the user interface to account for different device types including the desktop, Xbox, windows phone and tablets. This will involve incorporating support for both keyboard & mouse, Xbox controllers and touch. In terms of the Look & Feel, adherence will be kept to the Windows 10 design guidelines as specified by Microsoft. (Microsoft, 2015)

3.4. .NET Framework

3.4.1. Introduction

The .Net Framework is a framework developed by Microsoft. It has a class library known as the Framework Class Library (FCL) and provides interoperability across a number of languages. Programs developed with the .Net Framework are executed in a software environment known as the Common Language Runtime (CLR), a virtual machine that provides security, memory management and exception handling services. The FCL provides UI, database connectivity and web app development in addition to network communications and mathematical algorithms. The .Net Framework is intended to be used for the development of new applications targeting the Windows platform and the Visual Studio IDE is designed largely to aid developers in producing .Net Framework applications.

3.4.2. Architecture

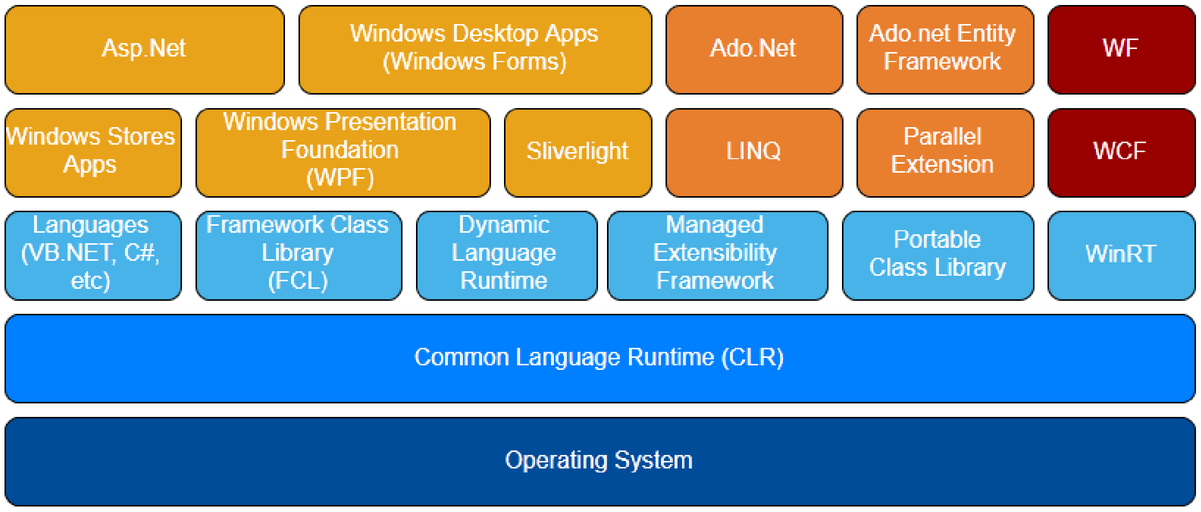

Asp.Net
Windows Desktop Apps (Windows Forms)
Ado.Net
Ado.net Entity Framework
WF
Windows Stores Apps
Windows Presentation Foundation (WPF)
Sliverlight
LINQ
Parallel Extension
WCF
Languages (VB.NET, C#, etc)
Framework Class Library (FCL)
Dynamic Language Runtime
Managed Extensibility Framework
Portable Class Library
WinRT
Common Language Runtime (CLR)
Operating System

Fig 17. .NET Framework 4.5 Architecture

Common Language Runtime (CLR)

The CLR is the run-time environment responsible for running the code developed using the .NET framework and provides services that simplify the development process. Compilers uncover the CLR's functionality, enabling the developer to benefit from a managed execution environment. The advantage of the CLR is gained from cross-language integration, a simplified component interaction model, security enhancements and support for deployment and versioning.

The CLR automatically handles the layout of objects and manages object referencing, releasing them from memory when they are no longer required. The garbage collector removes the risk of memory leaks and other errors.

In addition to garbage collection, the CLR provides performance improvements, extensible types, and support for multithreaded applications and exception handling. (Microsoft, 2015)

Framework Class Library (FCL)

The FCL is a class library that includes interfaces and types that provide access to the functionality of the system and includes multiple namespaces that expose system functionality such as input and output support, diagnostics, and basic GDI+ graphics functionality. (Microsoft, 2015)

Windows Presentation Foundation (WPF)

WPF provides the developers with a unified model for creating rich user experiences that integrate UI and media. WPF can create a large variety of standalone and browser-based applications. The fundamental element of WPF is a vector-based rendering engine that is resolution-independent and takes full advantage of contemporary graphics hardware. This core is extended with a broad collection of development features including XAML, data binding, templates and animations. (Microsoft, 2015)

4. Related Work

4.1. Applications Using MCTS

4.1.1. Single-Player Deterministic Games

One-Player Deterministic Games often come in the form of puzzles or scheduling problems. The first application of MCTS was to solve Production Management Problems (PMP) as found in airline scheduling. (Chaslot, et al., 2006)

Another application of the MCTS algorithm in One-Player Deterministic Games was to SameGame. In SameGame, the objective for the player is to remove tiles of the same colour from a board. The more tiles of the same colour removed at once, the higher the score obtained. The researchers used their MCTS implementation SP-MCTS to obtain the world record for the game (Schadd, et al., 2008). This was later broken by Cazenave with his Nested Monte-Carlo search, which also exceeded the top human score in Morphion Solitaire (Cazenave, 2009).

4.1.2. Two-Player Deterministic Games

Some of the greatest success and research of MCTS has been focus in this category of game. Traditionally, $\alpha\beta$ pruning with high quality evaluation functions were the standard for creating highly skilled computer players in games such as Chess and Draughts. Yet the $\alpha\beta$ pruning method achieved scant success in the game of Go.

It was not until 2006 that success in creating computer players for Go was achieved with the MCTS algorithm. Since 2006 and 2007, MCTS-based programs have won the 9x9 and 19x19 tournaments at the Computer Olympiads, respectively. The first program to achieve this success was MoGo Titan, defeating a human Go player in a match with a 9-stone handicap (Gelly & Wang, 2007). Aside from the triumph of MoGo, the best MCTS-based Go programs include CRAZY STONE (Coulom, 2006), MANY FACES OF GO (Fotland, 2009) and FUEGO (Enzenberger & Müller, 2009).

4.1.3. Multi-Player Deterministic Games

MCTS has also been applied to Hearts, Spades and Chinese Checkers. For Chinese Checkers, it was shown that MCTS was capable of exceeding the normal search methods. For Spades and Hearts, MCTS could compete with the highest quality standard algorithms for those games. (Surtevant, 2008)

4.1.4. Single-Player Stochastic Games

MCTS has been applied to The Sailing Domain. In The Sailing Domain, a boat must find the shortest path between two points of a grid with varying wind conditions. This was tackled using the UCT-based MCTS with the boat's position represented as a pair of coordinates on a grid of definitive size. (Kocsis & Szepesvári, 2006). When compared to other planning algorithms for the domain space, UCT was shown to require less simulations for equal performance.

4.1.5. Two-Player Stochastic Games

MCTS was applied with poor results to Backgammon using a UCT-based MCTS implementation. The program was capable of finding a proficient opening in a third of dice rolls but proved to be pointedly weaker than the top Backgammon programs. (Lishout, et al., 2007)

4.1.6. Multi-Player Stochastic Games

An interesting application of the MCTS algorithm was to the cult board game Settlers of Catan. For the MCTS-based SMARTSETTLERS, Chaslot *et al* used an implementation with some specific domain knowledge to create a program that could defeat the strongest available AI at the time, JSETTLERS. (Szita, et al., 2009)

4.2. Viking Chess Applications

4.2.1. Jocly Games

The Jocly Games website features a wide variety of browser-based board games. Included in this list are implementations of some of the Viking Chess variants including Hnefatafl, Tawlbwrdd, Brandub, Ard Rí and Tablut. These implementations feature a standard ruleset based on the most common rules for Viking Chess. They also include a computer player capable of playing to a relatively high skill but no information on the method used to create the computer player is detailed. http://www.jocly.com/

4.2.2 Lutanho.net

There is another browser-based version of Viking Chess in 9x9 and 11x11 variants available from a personal website. This implementation uses the corner escape rules for the King piece. Tests on the implementation revealed that the computer player was not adept at defeating a human player in either variant. No information regarding the methods used to create the computer player is detailed. http://www.lutanho.net

4.2.3 Aage Nielsen

Aage Nielsen's website features a Viking Chess implementation that allows for players to play online against each other. The site also features an archive of past games that can be used for research and a discussion forum. http://aagenielsen.dk/

4.2.4. Dragonsheel's Lair

At Dragonheel's Lair, there are numerous Viking Chess variants available to be played correspondence-style against other registered users. http://www.dragonheelslair.com/

5. Conclusion

From the research into the history of Viking Chess, it has become clear that the domain space is poorly documented. The lack of documentation arises from a number of factors. Due to the game all but disappearing as an actively played board game, reconstructing the rules with any degree of certainty is difficult. The only reliable source of information about the rules of the game come from Carl Linnaeus' description of the Tablut variant. However, this description is a second-hand report on what Linnaeus observed from watching the Sammi people play the game. While these rules can be applied across the other variants with some success, there remains a high level of uncertainty concerning how the game was played at the height of its popularity. It is also unknown whether the rules the Sammi people used for Tablut were the same rules that were implemented in the other variants.

Once the variants and rules of Viking Chess had been examined, the next step was to analyze the available methods necessary for creating a highly skilled computer player for a two-player turn-based deterministic zero-sum game of perfect information.

The Minimax Search algorithm is a well-documented and popular choice for creating computer players for this type of game. However, the base Minimax Search algorithm can rapidly degrade in performance when faced with a game with a deep or high-branching game tree such as in Chess or Viking Chess.

The αβ Pruning algorithm offers an improvement upon Minimax and can be built on top of it. These improvements mark it as a possible candidate for creating a computer player for Viking Chess but the need for a high quality heuristic evaluation makes it a challenging algorithm to implement for the board game when so little is known about the domain space in terms of making high value moves.

The MCTS algorithm, which is the planned method for implementation, is a more interesting choice for Viking Chess. There are a number of reasons for this. First and foremost, the lack of reliance on a heuristic evaluation function means that the algorithm can choose the best move to make from any given position with some degree of accuracy, foregoing the need for deep and specific knowledge of the domain space. Already, MCTS has shown to be successful in domain spaces where the customary methods have fallen short, particularly in the game of Go. Secondly, the game tree used by MCTS is created and expanded as play progresses and more simulations are run. This results in a relatively shallow tree that steadily becomes more accurate at predicting the best move from a given position. Finally, the comparative novelty of the MCTS algorithm to other more traditional methods is intriguing in terms of research and implementation.

As the final application is planned for release on the Windows Store, it was important that the requirements for a Windows Store application were examined. With Windows 10, Microsoft introduced the UWP. This forms the basis of applications designed to be run across the Windows 10 family of devices. With UWP, the UI is designed using XAML which removes the need for the developer to create everything from within their code and abstracts the UI layer to an easily readable and configurable markup language.

The UWP is built upon the .NET Framework. This framework provides a number of features to developers, including a class library and support for multiple languages. It was important that the architecture and the features of the .NET Framework were explored so that when implementing a solution, the usefulness of the available resources can be maximized.

To conclude, research was carried out to discover the existing applications that employ the MCTS algorithm. In addition, the current availability of Viking Chess implementations was examined. This research reinforced the incentive to carry out the project as no existing Viking Chess application appeared to implement the MCTS algorithm.

References

Bouzy, B. & Chaslot, G., 2006. *Bayesian Generation and Integration of K-Nearest-Neighbor Patterns for 19x19 Go.* Reno, USA, IEEE 2006 Symposium on Computational Intelligence in Games.

Cazenave, T., 2009. *Nested Monte-Carlo Search.* Pasadena, International Join Conference on Artificial Intelligence.

Chaslot, G. M. J.-B., 2010. *Monte-Carlo Tree Search,* Maastricht, Netherlands: Maastricht University.

Chaslot, G., Saito, J. & Uiterwijk, J., 2006. *Monte-Carlo Tree Search in Production Management Problems.* Hasselt, s.n.

Coulom, R., 2006. *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search,* Lille: Charles de Gaulle University.

Enzenberger, M. & Müller, M., 2009. *Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search,* Alberta: University of Alberta.

Fotland, D., 2009. *The Many Faces of Go, Version 12.* [Online]
Available at: http://www.smart-games.com/manyfaces.html
[Accessed November 2015].

Gelly, S. & Wang, Y., 2007. Mogo Wins 19x19 Go Tournament. *ICGA Journal,* 30(2), pp. 111-112.

Hastings, R., 1841. *The Chess Player's Chronicle.* Volume 2 ed. London: s.n.

Kocsis, L. & Szepesvári, C., 2006. *Bandit based Monte-Carlo Planning,* Budapest: Hungarian Academy of Sciences,.

Lishout, F. v., Chaslot, G. & Uiterwijk, M., 2007. *Monte-Carlo Tree Search in Backgammon.* Amsterdam, Maastricht University.

Microsoft, 2015. *.NET Framework Class Library.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/gg145045%28v=vs.110%29.aspx
[Accessed 2015 November].

Microsoft, 2015. *Common Language Runtime.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/8bs2ecf4%28v=vs.110%29.aspx
[Accessed November 2015].

Microsoft, 2015. *Guidelines for Universal Windows Platform (UWP) apps.* [Online]
Available at: https://msdn.microsoft.com/library/windows/apps/hh465424.aspx
[Accessed November 2015].

Microsoft, 2015. *Windows Presentation Foundation.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/gg145045%28v=vs.110%29.aspx
[Accessed November 2015].

Murray, H., 1951. *A History of Board-Games Other than Chess.* Oxford: Oxford University Press.

Schadd, M. et al., 2008. *Single-Player Monte-Carlo Tree Search.* Berlin, s.n.
Shannon, C. E., 1950. Programming a Computer for Playing Chess. *Philosophical Magazine,* 41(314).

Strong, G., 2011. *The Minimax Algorithm,* Dublin: Trinity College.
Surtevant, N., 2008. An Analysis of UCT in Multi-Player Games. *ICGA Journal,* 31(4), pp. 195-208.

Szita, I., Chaslot, G. & Spronck, P., 2009. Monte-Carlo Tree Search in Sellters of Catan. *Advances in Computer Games,* Volume 6048, pp. 21-32.