Imperial College London
Department of Computing

Determining Game Quality Through UCT Tree Shape Analysis

by

Gareth M. J. Williams

Submitted in partial fulfilment of the requirements
for the MSc Degree in Computing Science of Imperial College London

September 2010

Abstract

Upper Confidence Bounds for Trees (UCT) is an important new algorithm for tree searching that has shown itself to be very powerful. For example, in 2009, the Fuego software beat strong professional and amateur Go players by incorporating UCT and pattern recognition a feat which was previously thought to remain decades away. UCT has given rise to new fields of research such as Monte Carlo Tree Search.

The hypothesis of this project is that there is a machine learnable relationship between the shape of UCT search trees for a variety of games their quality – how engaging the games are to humans.

A statistically significant positive result was identified, representing a machine learning system that may be of great use when helping to classify automatically-generated board games in the future.

Acknowledgements

I would like to offer my thanks to the following:

My supervisor, Dr Simon Colton, for his continued time, energy and support during this project.

Dr Cameron Browne, for helping me to understand the UCT algorithm and for providing the inspiration for this research.

My second marker and personal tutor, Dr William Knottenbelt, for his constant energy and generosity with his time, and whose superb Introduction to C++ course laid the foundation for what has been a great masters.

My parents, Andy and Jan, for their continued love and support through hard and good times. I could not have done any of it without you.

My brother David, for cooking me a seemingly endless supply of gourmet food during the last year, and especially for his pièce de resistance – the power meal.

All my friends here on the course at Imperial College. You've made it a really special year.

Contents

Chapter 1

Introduction

Leibniz once said "human beings are never more ingenious than in the invention of games". We play games in a myriad of forms: games of skill and games of chance, games of logical reasoning, memory games, observation games, creative games. Sometimes we play them for the thrill of victory, at others we play them for sheer entertainment. Games are have been ingrained in our culture for millenia. In many ways it is our playing of games that makes us human.

Board games have existed for thousands of years. Examples of board games have been found buried in Egyptian tombs, dating as far back as 3500 B.C[1]. They retain their appeal today: despite the rise of computer games, many more people play board games than video games on a daily basis, even in Western cultures such as the U.S.A.[2]. Within board games, there are several sub-categories, including race games (such as Backgammon), war games (such as Risk), word games (such as Scrabble), roll-and-move games (such as Monopoly) and many more.

This thesis is concerned in particular with abstract strategy board games. These are games which are abstract in the sense that they have no particular theme (or that any theme they have is not of importance to their gameplay). It is likely that themes associated with any popular board game will dissolve over time, as players focus more and more on the abstract game rules and dynamics rather than their historical or cultural importance. Many of the oldest games we see today – such as Go, Chess and Oware – are highly abstract, but retain loose thematic links to their origins.

Abstracting games has allowed human players to ignore their decorations and focus on their gameplay. It has also allowed a great deal of academic study of games to take place, starting with the work of Neumann in the 1920s and 1930s. Since then game theory has blossomed into a huge field that has influenced other areas from biology to political science.

Game theory first really took hold as a field at around the same time as the first modern computers were being developed. This led to the birth of computataional game theory, focusing on compuational and algorithmic issues associated with playing games. In 1958, the NSS chess program became the first to implement alpha-beta pruning of its search tree. Theoretical advances combined with rapidly growing processor speeds led to huge leaps in this area in the late 20th century, culminating in the defeat of Garry Kasparov by the IBM supercomputer Deep Blue in 1997. This chess match was watched by millions of chess fans around the world and signalled the end of human superiority over chess computers. The strongest chess computers are now considerably above the level achieved by even top grandmasters.

Without the intuition that guides humans, computers inevitably have to search through a tree of [state, action] pairs to make intelligent decisions. AI players search systematically through a tree of possible future states of play to decide which move to play next. And despite their superiority in chess, computers still struggle with some games where this tree grows too quickly for them to make accurate analysis in reasonable time. Go is an example of such a game. Since the Deep Blue match in 1997, Go has taken over as the frontier of computer game-players. Despite its simple rules, its huge number of possible game continuations makes it very hard for computer players to play.

Upper Confidence Bounds for Trees (UCT) is a simple yet powerful new tree search algorithm which was developed only 4 years ago[3], which has shown itself to be very powerful. In August 2009, the Fuego software beat both strong professional and amateur Go players in games played at the 2009 IEEE International Conference on Fuzzy Systems by incorporating UCT and pattern recognition[4] – a feat which was previously thought to remain decades away. The success of the algorithm is not limited to the game of Go: the UCT-based CadiaPlayer software won the 2007 and 2008 AAAI General Game Playing competitions[5], where game-playing algorithms must do battle across a selection of games which are not revealed until the tournament begins. UCT can play a wide selection of games successfully because its algorithm does not require any prior knowledge or heuristics.

An intruiging aspect of the UCT algorithm is that its search trees exhibit asymmetry, a feature not present in classical tree search algorithms such as breadth-first search. This asymmetry emerges from the selective way in which UCT samples actions – it focuses its investigations on moves which look strong; sometimes it re-checks moves which look less promising in case their value estimate is erroneous. Each UCT tree has a unique shape defined by the current game situation.

Over the last few decades computers have been widely used for playing board games, but until recently they have never been used for *creating* board games. However, this area of research has been addressed by Browne[6], who, in his PhD thesis, also showed that board game quality may be assessed through self-play by a computer player. The fact that UCT tree shapes are depend uniquely on the different games and game positions it is presented with leads to the intruiging possibility that there is a link between this UCT tree shape and game quality.

The main hypothesis of this research can be stated as follows:

> A machine learner trained on statistics from UCT trees from of a variety of board games can accurately predict the quality of a new game it is presented with – that is, how interesting the game is to human players.

If such a link can be shown to exist, it may contribute to the creation of computer-generated board games in the future: a machine learner could separate games likely to be good from those likely to be bad. Under the assumption that most automatically-generated board games will be unplayable, it would be useful to find a machine learner simply capable of differentiating between playable and unplayable games. So a second hypothesis to be investigated was:

> Machine learners trained on UCT tree data can accurately predict whether a new game is playable or not.

1.1 Chapter summary

The chapters of this report are set out as follows:

Chapter 2: Background

This chapter covers some foundational material which will be referred to throughout this thesis. It describes the basic concepts of tree search, then moves on to discussing advances in this field and the definition of the UCT algorithm. It then covers concepts of game quality and the tree metrics which can be used to summarise the shape of trees. Finally, it gives some background on machine learning systems, which will be used to try to identify a link between tree shape and game quality.

Chapter 3: Implementation

This chapter covers the implementation necessary to complete the research. It discusses the implementation of the UCT algorithm itself, and of the games programmed for the algorithm to play. It also covers the creation of unplayable games, which would be used for addressing the second hypothesis. The means of extracting and aggregating data for the machine learning systems is covered. Finally, the approach to carrying out the training and testing of the machine learing techniques is described.

Chapter 4: Results

This chapter covers the results of the research, showing some of the UCT search trees created and the tables and charts relating to the performance of the machine learning systems. It reveals that there is one [data collection and aggregation, machine learning system] combination which performs extremely well on the task of predicting game playability.

Chapter 5: Evaluation

This chapter analyses the results from Chapter 4 from a statistical standpoint, to establish their significance. The preferred UCT data collection method and machine learning system combination is discussed in the context of predicting the quality of automatically-generated board games. The chapter then discusses the main features of the UCT trees used by the machine learning systems to make their predictions, and why these features proved relevant to game quality. Finally, areas for improvement are covered, including discussion of the reasons why even a strong UCT algorithm falls down when playing some games.

Chapter 6: Further work

This chapter discusses how future work could be used to confirm the findings of this research and address some of the areas for improvements mentioned in chapter 5. It then discusses how a major re-engineering of the UCT algorithm might lead to a stronger UCT player, less likely to succumb to the problems with some games brought up in Chapter 5. This stronger player would also be likely to created search trees with a stronger link to quality/playability.

Chapter 7: Conlusion

This chapter summarises the rest of the thesis and summarises the major findings.

Chapter 2

Background

While there are a huge variety of different board games in existence all over the world, many of them share similar "landscapes". For example, Chess and Connect4 are very different games, but they both take place by means of a game board, which has an arrangement of pieces on it, and players take turns to play. Both have winning conditions which are related to the arrangement of pieces on the board. By finding abstractions which iron out the differences between games and build on the similarities, academics can study them in a general way.

This project will focus on turn-based two-player abstract strategy board games. At any moment a game of this type is in a particular *state* – this state encapsulates the combination of the arrangement of pieces on the board, the player to play next, and other variables which change throughout the course of the game (such as a score or the amount of time each player has remaining). In each state, a player takes an *action* (makes a move) which changes the game state – this usually involves changing the arrangement of pieces on the board, and passing the turn to the other player.

So a turn-based two-player game can be abstracted in this way: into *states*, and *actions* which bring about changes in state.

Let us use the game of Noughts & Crosses as a concrete example. Let us label our two players Max and Min, with Max to play first. The state of the game on Max's first move is simply an empty board (see figure 2.1). For his move, he can place a nought anywhere he likes, so he has 9 available actions. Each action leaves the board in a different state, so there are 9 possible states of the game after one ply (when Min is about to take his first move, see figure 2.2)[1]. For each of these possible states, there are 8 possible replies.

So from one initial state, there are 9 possible states of the board after 1 ply, and several more possible states after 2 plies (see figure 2.3). We can think of this growth from one current state to many possible future states as a tree. The current state sits at the root, with possible future states connected to it via branches. Each branch represents an action which brings about a change from one state to another.

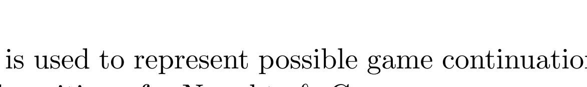


Figure 2.1: An example of how a tree is used to represent possible game continuations. The root of the tree here is the initial position of a Noughts & Crosses game.

[1]ignoring symmetries

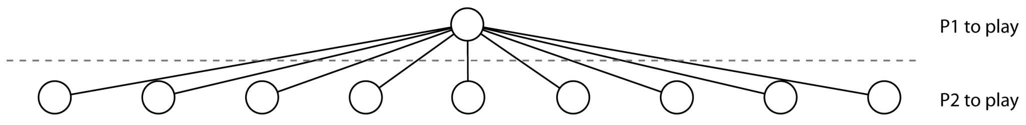


Figure 2.2: Looking one move ahead, the tree has "grown": there are 9 possible game states.

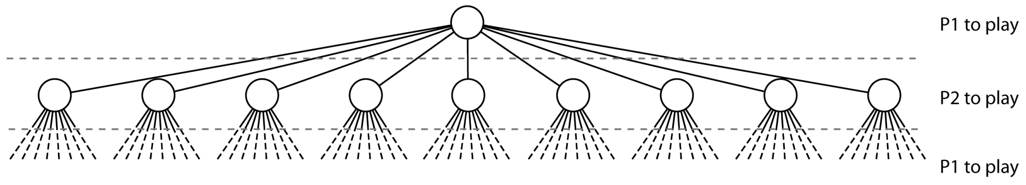


Figure 2.3: Looking two moves ahead, the tree continues to grow: there several more possible game states.

Artificial intelligence calls these structures search trees. In playing a game strategically, humans intuitively search through these potential future states of play as they decide which move to make. Computers, too, plan and calculate the best moves to play by analysing these search trees. As stated in the introduction, this paper aims to find a link between the shape of search trees created by the UCT algorithm and game quality – how interesting and engaging a game is to human players.

From an abstract perspective, trees can be thought of as simply a collection of vertices, connected by edges. This abstract approach allows mathematicians to study trees and understand their properties in a general way. Below are some preliminary definitions relating to trees. A reader familiar with the basics of tree theory can move onto the section 2.1, which discusses tree search in general, before the introduction of the UCT tree search algorithm in section 2.2.

Definition 1. Graph

A graph is a collection of vertices, V, and edges, E. Each edge is an ordered pair of vertices (v_1, v_2), and defines a link between the vertices.

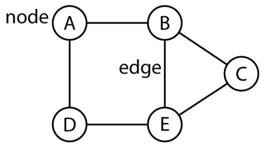


Figure 2.4: An example of a graph.

Definition 2. Path

Let P be an ordered subset of the edges E of a graph, and let p_i be the edges in P, for $i = 1...n$. Let $v_1(p_i)$ and $v_2(p_i)$ be the vertices of edge p_i.

P is a path from vertex V_1 to V_2 if all of the following hold:

$$\begin{aligned} v_1(p_1) &= V_1 \\ v_2(p_i) &= v_1(p_{i+1}) && \text{for } i = 1...n-1 \\ v_2(p_n) &= V_2 \end{aligned}$$

Definition 3. Cycle

Let C be a path in a graph, and let c_i for $i = 1...n$ be the edges in C.

Then C is a cycle if $v_1(c_1) = v_2(c_n)$.

Definition 4. Connected

A graph G is connected if for any pair of vertices V_1 and V_2 in G, a path exists between V_1 and V_2.

Definition 5. Tree

A tree is a connected graph without cycles.

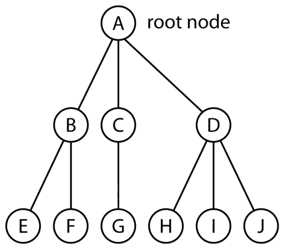


Figure 2.5: An example of a tree. Throughout this paper trees will be drawn with their root node at the top.

2.1 Tree search

Tree search can be considered a special case of Markov Decision Process (MDP), which in turn is an extension of a Markov Process. A Markov Process is one where an agent moves randomly from one state to the next, and the probability of moving from a current state s to a next state s' depends only on s (not on any previous states that might have been encountered in the process of reaching s).

An MDP is similar to a Markov Process, but for each current state s, the agent has a selection of actions available – hence the "decision". The choice of action affects the probability distribution of the next state. Thus, for an MDP, the next state s is decided by a probability distribution which depends only on the current state s and the chosen action a. MDPs may also associate a reward with each transition from one state to another. This reward may be deterministic or have a probability distribution – but it too will only depend on s and a.

Definition 6. Markov Decision Process

A Markov decision process M on a set of states S and with actions $\{a_1, ..., a_k\}$ consists of:

- Transition probabilities: For each state-action pair (s, a), a next-state distribution $P_{sa}(s')$ that specifies the probability of transition to each state s' upon execution of action a from state s.
- Reward probabilities: For each state-action pair (s, a), a distribution $R_{sa}(r)$ on real-valued rewards r for executing action a from state s.

Markov Decision Processes define many commonly encountered decision and planning situations, including games. Imagine playing a game of chess. The likelihood of facing a given board position in one turn's time is dependent on the current board position (the current state of the MDP, s), and the move made in this position (the action chosen, a). There is no explicit reward when a move is played, but if winning the game is the eventual goal, then there is an implicit reward associated with playing a strong move, since it leaves the game in a state which is more likely to lead to a win.

Theoretically, the best action to take in a given MDP can be calculated from the probability distributions for each (state, action) pair. Imagine that you were playing a game of chess against a computer chess player, and knew the probabilities associated with each of its possible replies in every board position. With enough processing power, you could take the current board position and use your knowledge of the computer player to calculate your best possible move. At time t, in state s_t, by analysing each possible move a you could work out the probability of being presented with any board position at time $t+1$. Continuing this process, you could calculate the likelihood of being in any given position for all time steps. You could then make a simple (albeit monumentally huge) calculation to work out which action maximises your expected payoff or minimum payoff.

However, in practice, this is rarely possible. For one thing, large state spaces and limited computing power mean that even if the exact probabilities associated with an MDP were known, the time required to make such a calculation would be too large. Further, in many situations these probabilities are not known.

One approach to this problem is to use a generative model. Although the agent facing an MDP may not know the MDP's probability distributions, a generative model allows the agent to obtain samples from these distributions. By sampling from the generative model repeatedly, the agent can build up a picture of the likelihood of each future game state.

Let us use the analogy of playing a game of chess against a computer chess player again – imagine you are faced with the task of making a move against the computer in a given position. Using a generative model is like having a second copy of the chess program on a separate computer. You can turn to this second computer and play each available move several times, each time recording the computer's reply. While you may not know the exact probability distributions of the chess program, this repeated sampling approach allows you to build up a more and more accurate picture of what the true probability distributions might be. Finally, when you have enough information about the computer's possible replies to each of your moves, you can work out what your best move is, go back to the main game, and make a move accordingly.

Definition 7. Generative model

A generative model for a Markov Decision Process M is a randomized algorithm that, on input of a state-action pair (s,a), outputs a state s' and a reward r, where s' and r are randomly drawn according to the transition probabilities $P_{sa}(s')$ and $R_{sa}(r)$ respectively.

Definition 8. Policy

A (stochastic) policy is any mapping $\pi : S \to \{a_1, ..., a_k\}$. Thus $\pi(s)$ may be a random variable, but depends only on the current state s.

The above definition of a policy is perfectly natural – given any state s in S, a policy returns an action to play in state s.

Thus a policy can be executed from any game state, and, allowing for replies determined by the probability distributions of the MDP, future game states, until the game reaches an end. The expected reward of executing a policy π from a game state s until the game ends is called the value function of the policy from state s. It is calculated by simply taking the expectation (over all randomisation in the policy) of the future rewards of playing this policy.

Definition 9. Discount rate

The discount rate γ for an MDP is a factor by which rewards at time step $t+1$ are worth less than rewards at time step t.

Definition 10. Value function

The value function V_π for any policy π is defined as:

$$V_\pi(s) = \mathbf{E}\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_i \mid s, \pi\right]$$

where r_i is the reward received on the i^{th} step of executing the policy π from state s, γ is the discount rate of future rewards, and the expectation is over the transition probabilities of the MDP and any randomization in π.

In the context of a game, γ is usually taken to be 1 (a win in two turn's time is as good as a win in one turn's time). The rewards r_i are only non-zero when playing a move which wins ($r = 1$) or loses ($r = -1$) the game. The value function is the expected reward of playing policy π until the game's conclusion – an estimate of how likely policy π is to eventually lead to a win or loss. $V_\pi = -1$ indicates that a policy π will inevitably lead to a loss; $V_\pi = 1$ indicates that policy π will with certainty lead to a win.

Definition 11. Optimal value function

We define the optimal value function and the optimal Q-function as

$$V^*(s) = \sup_\pi V_\pi(s)$$

So $V^*(s)$ is the maximum expected reward attainable starting from a game state s. A natural measure of the quality of a policy is how close it gets to attaining this maximum reward – the "regret" of the policy.

Definition 12. Regret

The regret of a policy π is defined as:

$$|V^*(s) - V_\pi(s)|$$

So regret of a policy π is the expected loss as a result of playing π instead of the best policy.

Kearns et al.[7] have shown that for any $\epsilon > 0$, a *sparse sampling* algorithm exists which can find a policy whose regret is less than ϵ. Sparse sampling means that the algorithm works in an online manner - it does not need to have any overall knowledge of the whole state space S. It works from the current state, sampling actions and using a generative model to calculate the rewards for these actions.

Kearns' algorithm works as follows: based on the desired accuracy ϵ, it calculates a required number of calls to the generative model per state, C, and a lookahead depth, H. Given a starting state s, each available action a is attempted C times. By using the generative model, this gives C successor states for each action a. This process is repeated for each successor state, trying each available action C times, up to a maximum depth of H.

The value of an action state s is estimated to be the maximum value of the actions available from s, and the value estimate of an action a is its reward plus the average value of each state returned by the generative model when a is played. Essentially, this is just a minimax search of the search space – the only special feature is that a generative model is used to gauge the likely replies from the MDP.

Kearns et al.'s research showed that sparse sampling could lead to accurate value estimation in a search space too large to be solved by classical methods, as is the case with most playable games. UCT draws upon the same sparse sampling technique, but uses a selection algorithm to decide which moves to sample at each point in its execution.

2.2 UCT

Upper Confidence Bounds for Trees (UCT) is an important new algorithm for tree searching that has shown itself to be very powerful. It was first described in a paper called "Bandit-based Monte-Carlo Planning" published by Levente Kocsis and Csaba Szepesvári, in 2006[3]. Their algorithm is based on Kearns et al.'s work, in that it uses sparse lookahead sampling to calculate a near-optimal policy. However, instead of using a breadth-first approach (sampling each action exactly C times, up to depth H), UCT samples actions *selectively*, slowly expanding its search tree as it does so.

The challenge of finding the best move in a state s boils down to accurately estimating the value function for each available action a. A good algorithm for playing a game is one that can estimate the true value of each available action (move) accurately in a reasonable amount of time. This estimation is done by making several calls to the generative model in order to build up an accurate picture of the likely payoff of each action a. Testing a given action more times will result in a more accurate estimate, but how many times should each action be sampled? Kocsis and Szepesvári express this issue very succinctly:

> "In order to achieve [accurate estimation], an efficient algorithm must balance between testing alternatives that look currently the best so as to obtain precise estimates, and the exploration of currently suboptimal looking alternatives, so as to ensure that no good alternatives are missed because of early estimation errors."

Finding the right balance in this situation is known as the exploration-exploitation dilemma. It is instructive to consider research on the most basic form of this dilemma, which occurs in *multi-armed bandit problems.*

Multi-armed bandit problems suppose that a gambler is presented with K one-armed bandits, each with its own payoff distribution. So some machines might be more profitable than others, but initially the gambler has no way of knowing which machines these are. The gambler is given the opportunity to play the bandits a given number of times, choosing a different machine on each play if desired, and must find the best policy to maximise their winnings. Auer, Cesa-Bianchi, and Fischer[8] proposed playing a strategy which initially plays each machine once, then plays the machine j which maximises the quantity

$$\overline{X}_j + \sqrt{\frac{2\ln(n)}{n_j}} \tag{2.1}$$

where $\overline{X}_j$ is the average observed payout so far from machine j and n_j is the number of times machine j has been played. Note that formula (2.1) can also be expressed as

$$\overline{X}_j + c_{n,n_j} \tag{2.2}$$

$$\text{where } c_{n,n_j} = \sqrt{\frac{2\ln(n)}{n_j}}$$

The second term c_{n,n_j} is known as a *bias term.* Note that for a machine j which is not played often, as n increases but n_j remains small, the bias term grows. Thus after a certain number of plays, even a machine with a very low expected payoff value will have a large bias term, and so will be played once more. It is this bias term, therefore, that defines how the algorithm balances exploitation (of machines which currently look good) and exploration (of currently poor-looking machines). It is also sometimes called the *exploration* term for this very reason. Auer et al. referred to this strategy as "Upper Confidence Bounds", or simply UCB.

This strategy is actually adopting a very human approach: play each machine once, then keep playing the machine that has, on average, given you the best payout so far – but from time to time try a machine you have not played for a long time, in case this machine is actually the best one. Auer et al. showed that their strategy has regret which is $O(\ln n)$ as $n \to \infty$. This is clearly a powerful strategy, and Kocsis and Szepesvári found a way to adapt it to tree search.

As mentioned above, UCT samples actions in its search tree selectively. When presented with a node, it first attempts each available action once, then chooses to sample action a that maximises the quantity

$$Q_t(s,a,d) + c_{N_{s,d}(t),N_{s,a,d}(t)} \tag{2.3}$$

where

$Q_t(s,a,d)$ is the estimated value of action a in state s at depth d at time t;

$N_{s,d}(t)$ is the number of times state s has been visited at depth d up to time t;

$N_{s,a,d}(t)$ is the number of times action a has been chosen in state s at depth d up to time t;

and

$$c_{t,s} = C_p\sqrt{\frac{2\ln(t)}{s}}$$

with C_p a constant

Note the similarity between equations (2.2) and (2.3). The algorithms are intimately connected. The current estimate of a node's value, $Q_t(s,a,d)$, is equivalent to the average payout so far from a bandit, $\overline{X}_j$. The bias terms are also very similar, up to a multiplicative constant. UCT is effectively using the UCB algorithm at each node in its search tree to choose which action to sample next.

2.2.1 Monte-Carlo Tree Search

One question remains: how does UCT place an estimated value upon each node in its search tree? When playing a game, there is no explicit reward in moving from one state to another; only finally winning or losing the game provides a definitive reward in this sense. So any value estimates will not come about as a result of specific rewards associated with making moves, but from rewards propagated back up the search tree from nodes representing terminal positions (won, lost or drawn positions).

But to keep track of all sampled moves continuously until a game terminates would be hugely impractical. The end of the sample game might not occur for 50 moves or more. As shown by Kearns et al., accurate estimation of the best move to play in the current game state can be determined by sampling only near-future actions, so this is where UCT should be focusing its efforts.

UCT's approach to this problem is to make a Monte-Carlo simulation every time it reaches the edge of its current search tree. When it reaches a leaf node, it plays a completely random playout of the rest of the game from the position represented by the leaf node. The result of this game is recorded, and propagated back up the tree along the path back to the root node. Each node keeps track of the number of times it has been visited, and its score (the total payoff of each playout game associated with it). The fraction $\frac{\text{score}}{\text{visits}}$ is used as the Q-value referred to in the formula 2.3.

This technique of performing random playouts as part of value function estimation is now known as Monte-Carlo Tree Search. It first appeared with the advent of the UCT algorithm, and has now spawned its own area of research. Used on their own, these Monte Carlo simulations would simply be an indication of how likely a given position is to lead to a win with *random* play. However, with the slow expansion of its search tree, recording of score and visit counts, and use of the UCB formula to select which moves to sample next, UCT builds up a more and more accurate estimation of the value of near-future moves – allowing it to act intelligently in the current board situation.

So UCT samples actions according to its current estimation of the payoff for those actions. If an action a available in a state s in the search tree currently looks to be strong (has a high current Q value), it will be sampled often when the algorithm reaches state s in the future. However, it will also from time to time sample moves which do not currently look strong, in an effort to ensure that its current value estimation for these moves is correct.

This is a very human approach. A human playing chess will rarely spend any time considering a move which results in him losing his queen after 2 plies with no compensation. The immediate detrimental effect of being a queen down means that the move simply does not warrant further investigation to 4, 6 or more plies. The UCT algorithm would also be unlikely to investigate this move deeply, since the early estimation of the move's value is likely be very low.

Another interesting feature of the tree searches made by UCT is that they are *asymmetrical.* This asymmetry is related to its intelligent sampling method. Actions with high payoff values will be sampled often, and hence will likely be at the top of large, deep sub-trees. Actions with low payoff values will be sampled rarely, so will link to far smaller sub-trees. Longer-established search algorithms such as iterative deepening simply sample all actions and return a search tree that is featureless – these trees simply sample each move available in each position, up to a certain depth. UCT, on the other hand, returns search trees that are asymmetrical and uniquely related to the exact game situation being searched.

This paper aims to answer the question: can the shape of UCT trees somehow be linked to the quality of the game being played? If this question is to be addressed, two important issues present themselves – how to judge the quality of a game, and how to encapsulate the shape of a UCT tree. These issues are discussed further in the next two sections.

Overleaf is an example tree from the game of Kalah, for a UCT algorithm of 5,000 iterations. Note the asymmetry, with UCT probing some continuations more deeply than others. If the hypothesis stated above is correct, a machine learner should be able to find a relationship between tree shape and game quality, to the point where it can predict the quality of new games accurately, based only on a set of example UCT trees for those games.

Pseudo-code for the UCT algorithm is shown in figure 2.6, and the algorithm is explained diagramatically in figure 2.7.

```
int get_best_move_via_UCT(Game g, int maxIterations) {

    Node* rootNode = new Node(g);
    Node* currentNode = rootNode;

    for(int i=0; i<maxIterations; i++) {

        // Descend through tree using UCB algorithm, updating
        // visit counts for each node encountered.

        while (currentNode->timesVisited > 0) {
            currentNode->timesVisited++;

            if (currentNode->is_terminal_position()){
                break;  // Cannot descend any further
            } else {
                currentNode = currentNode->descend_via_UCB();
            }
        }

        // Play out game with completely random moves from leaf
        // node and record game score. Immediately returns game
        // score if node is already a terminal position.

        int playoutScore = currentNode->playout_random_game();

        // Move back up through search tree, updating
        // nodes with the score from the playout game

        while (currentNode != NULL) {
            currentNode->score += playoutScore;
            currentNode = currentNode->parent;
        }

    }

    // Return child node with best score:visits ratio

    return rootNode->get_best_child_node();

}
```

Figure 2.6: Pseudo-code for the UCT algorithm. It takes as arguments a game position and a desired maximum number of iterations, and returns an integer representing the index of its chosen move. descend_via_UCB() simply returns one of the child nodes of the calling node, according to the UCB algorithm. get_best_child_node() simply returns the index of the child node with the hightest average score ($\frac{\text{score}}{\text{visits}}$ ratio).

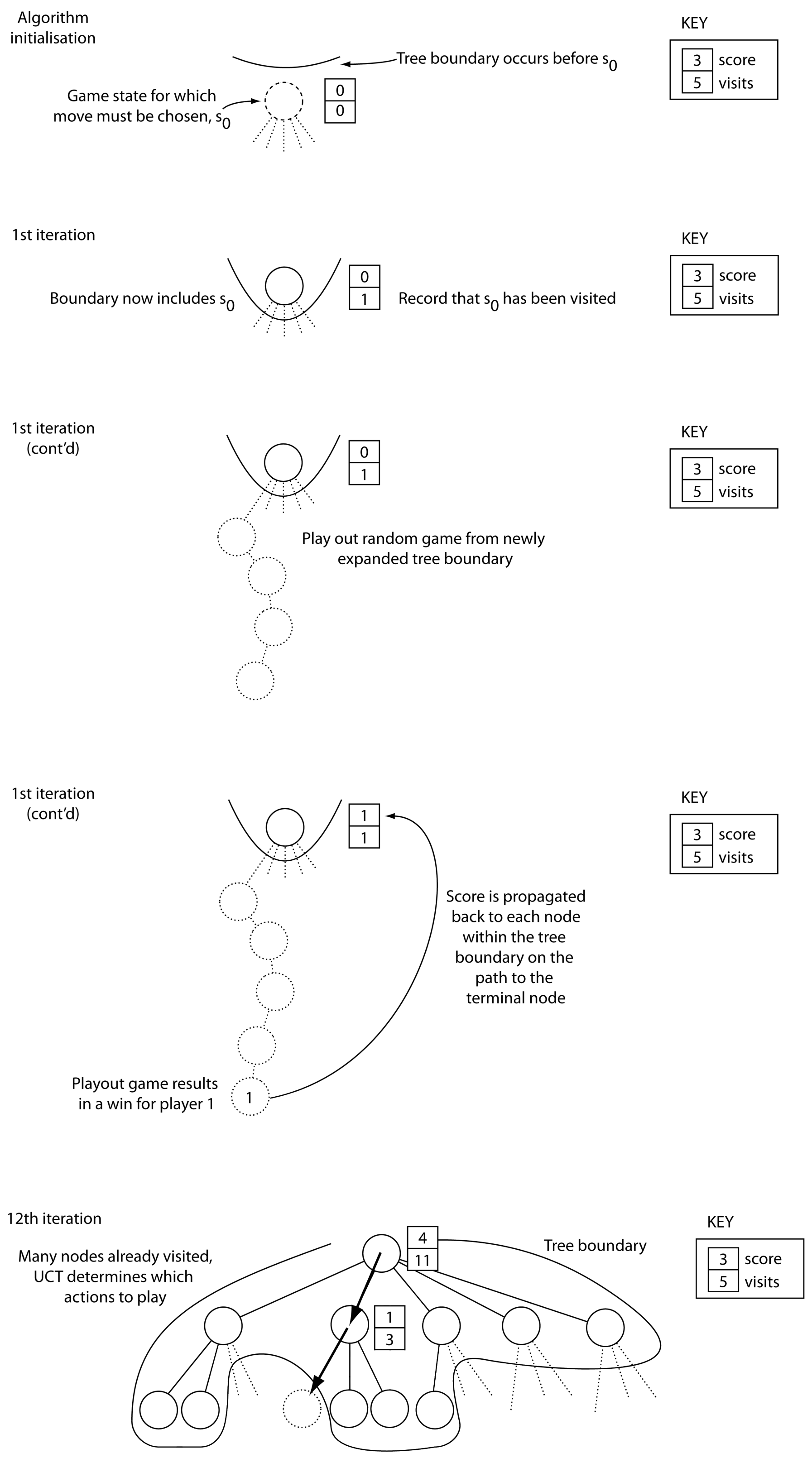
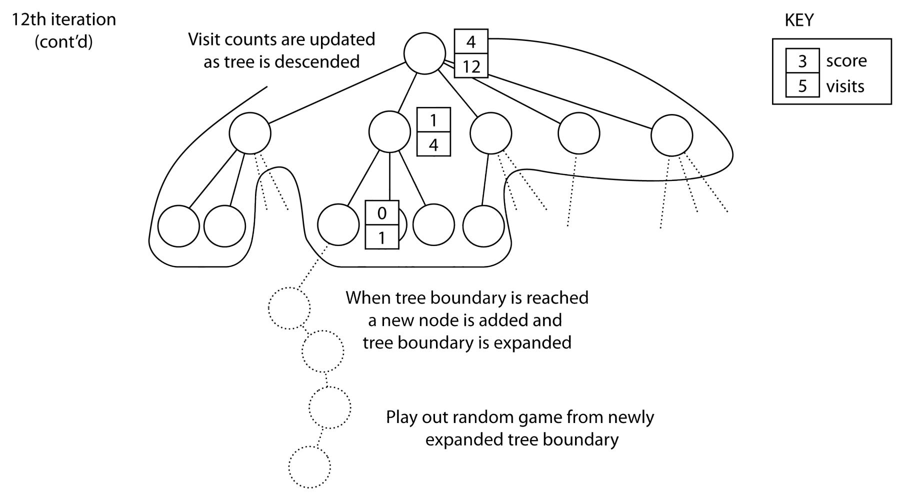
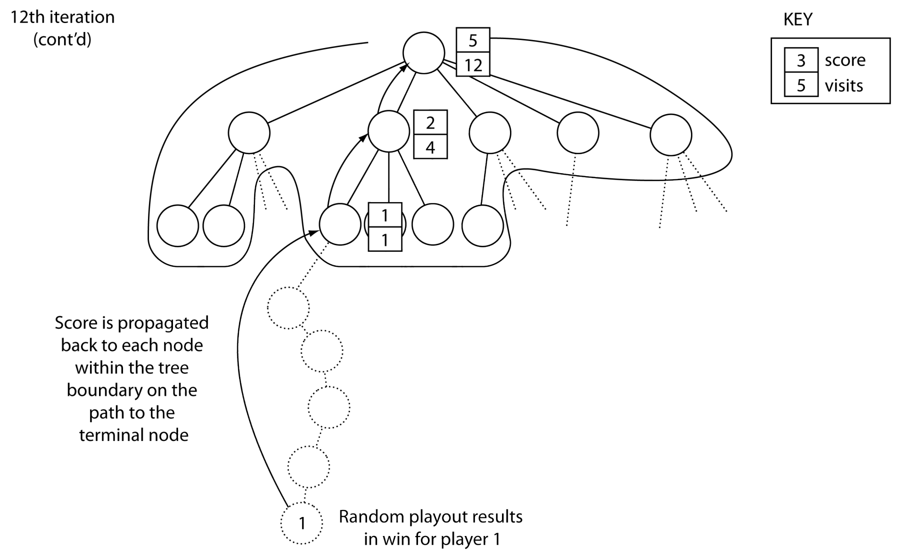

Algorithm initialisation
Tree boundary occurs before s_0
Game state for which move must be chosen, s_0
0
0
KEY
3 score
5 visits
1st iteration
Boundary now includes s_0
0
1
Record that s_0 has been visited
KEY
3 score
5 visits
1st iteration (cont'd)
0
1
Play out random game from newly expanded tree boundary
KEY
3 score
5 visits
1st iteration (cont'd)
1
1
Score is propagated back to each node within the tree boundary on the path to the terminal node
Playout game results in a win for player 1
1
KEY
3 score
5 visits
12th iteration
Many nodes already visited, UCT determines which actions to play
4
11
Tree boundary
1
3
KEY
3 score
5 visits

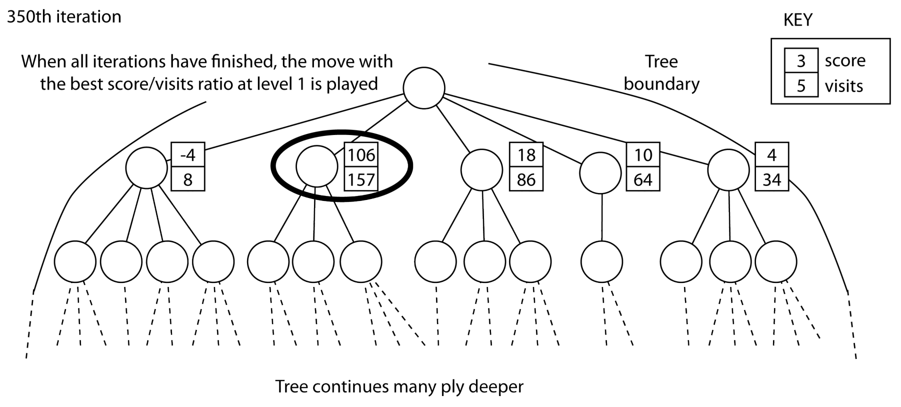
Figure 2.7: Diagrams explaining the execution of the UCT algorithm



12th iteration (cont'd)
Visit counts are updated as tree is descended
4
12
1
4
0
1
KEY
3 score
5 visits
When tree boundary is reached a new node is added and tree boundary is expanded
Play out random game from newly expanded tree boundary



12th iteration (cont'd)
5
12
2
4
1
1
KEY
3 score
5 visits
Score is propagated back to each node within the tree boundary on the path to the terminal node
1
Random playout results in win for player 1



350th iteration
When all iterations have finished, the move with the best score/visits ratio at level 1 is played
Tree boundary
KEY
3 score
5 visits
-4
8
106
157
18
86
10
64
4
34
Tree continues many ply deeper

Figure 2.7 (cont'd): Diagrams explaining the execution of the UCT algorithm

2.3 Game quality

Many of the games we play today have existed since time immemorial, and have been refined and perfected over thousands of years. They represent a fundamental part of

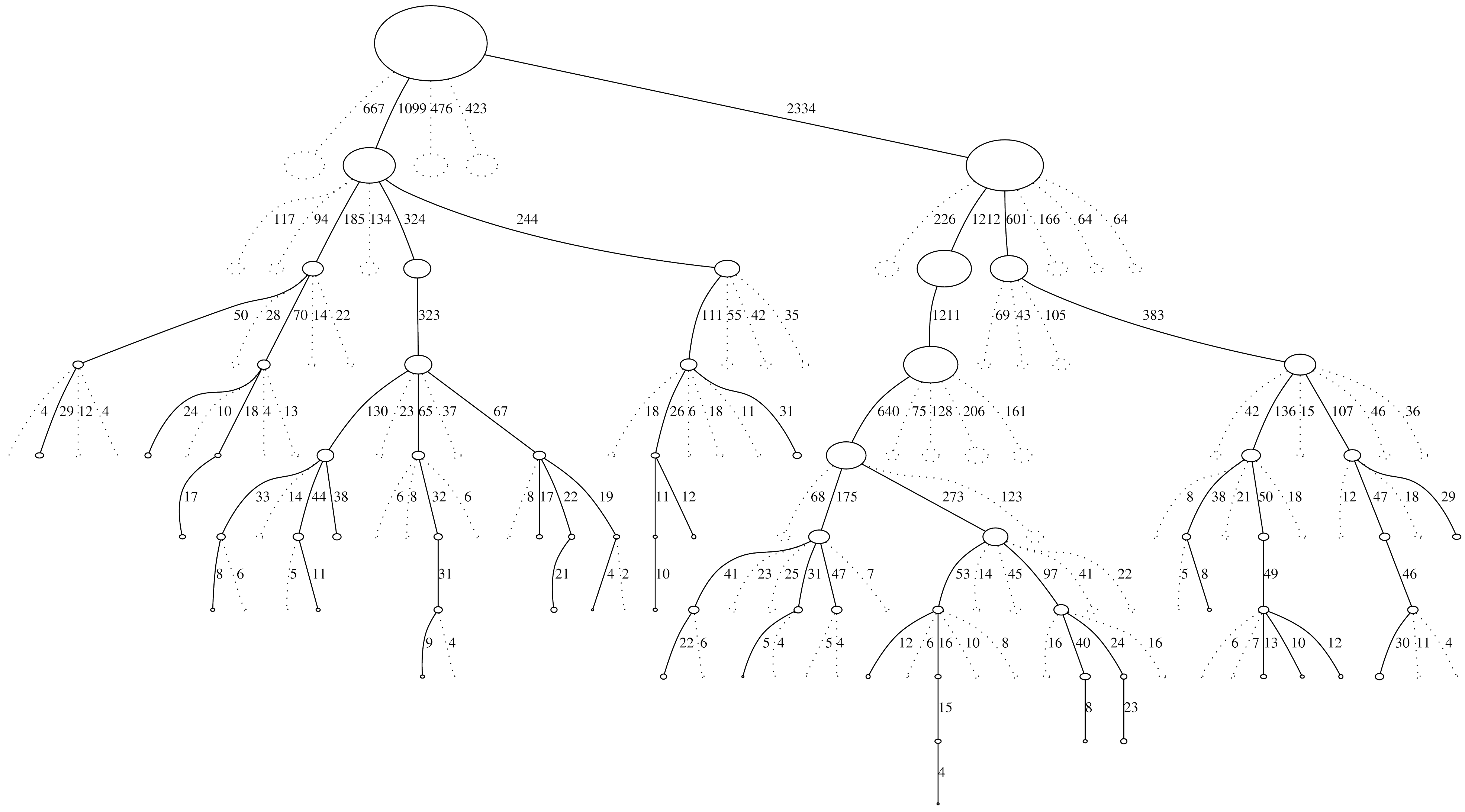


Figure 2.8: An example game tree from the game Kalah (an Oware variant). Note the asymmetry of the tree, corresponding to some moves being investigated more deeply than others. This asymmetry is a distinguishing feature of the UCT algorithm, and means that each UCT tree has its own unique shape. The hypothesis of this paper is that the shape of the UCT tree is related to game quality, that the relationship can be learnt by a machine learner and this machine learner can subsequently predict the quality of a new game based on UCT trees for that game.

our culture, and some may say that it is our playing of abstract games that most makes us human.

Yet, in many ways, we take the games we play for granted. What is it that makes one game timeless and magical and another tedious and uninspiring? These are questions that often go unasked, but they have been addressed by keen games players and games designers. Thompson, in his article "Defining the Abstract"[9], defines the merit of a game in terms of four key qualities: depth, clarity, drama and decisiveness.

Depth indicates that a game has enough complexity to be played at different levels of expertise. Part of the enjoyment of playing a game is the learning experience, learning to play the game better and better. If a game does not possess depth, there is no room for this improvement to happen, and experienced players will be little better than rookies. If a game does not possess depth, players will quickly begin to feel they have got everything they can from the game-playing experience.

Clarity expresses that the gameplay and rules of a game should be simply and concisely quantifiable. This simplicity allows humans to immediately make a judgement about what is a good move in a given position. If a game does not have clarity, it will be longwinded and frustrating to learn for human beings.

Drama expresses that a game should offer chances for counterplay for a player in a weaker position – they should still have a chance to come back and win the game. In a game with no drama, any slight advantage for one player will inevitably lead to victory; this makes the game boring and pointless for humans. Part of the attraction of playing a game comes from the unexpected twists and turns that occur as the game's story unfolds.

Decisiveness measures the ability of one player to achieve an advantage from which the other cannot recover. If a game has no decisiveness, then even perfect play might not lead to a strong enough position to win the game. This will clearly lead to a frustrating and futile game situation. A game must be winnable, or else it is not really a game at all.

These four key qualities can be grouped together into two pairs: depth and clarity, drama and decisiveness. Depth and clarity are both desirable qualities, but in many ways are contraposed. For example, a game with complex rules may possess great depth, but the very rules which give it this depth may mean it lacks the clarity to be enjoyable. A game such as Noughts & Crosses may possess great clarity, but no depth. Equally, drama and decisiveness are in many ways opposed to each other. A game with great drama might be difficult to bring to a conclusion if the player in the weaker position can constantly recover. A good game brings together both these pairs of qualities in exactly the right balance.

Wolfgang Kramer, a highly decorated game designer from Germany, also speaks of the concept of "tension" in an article entitled "What Makes a Game Good?"[10]. The tension in a game can be thought of as the *importance of playing the right move* at any point. If a game maintains high tension throughout, every move is important, and so the game will be engaging from start to finish. In a way, this relates to the decisiveness that Thompson describes: each moment in the game should have the *potential* to be decisive, otherwise the moves will hold little interest for the players.

Another basic feature of a good game is that it should avoid excessive repetition – a new instance should lead to different situations and challenges to a previous one. Inevitably similar patterns and endings will crop up from time to time, but a game in which all instances are alike (or nearly alike) will never hold much interest for human players.

2.4 Tree metrics

This research aims to find out if a machine learning system, can successfully analyse the *shape* of trees created by the UCT algorithm as it plays a variety of games, and find a link between the shape of these trees and the qualities of the games. In order for a ML system to do this, it must be presented with numerical data which summarises the shape of the trees produced for each game. To quantify this shape it is necessary to understand some of the fundamentals of tree theory and the measures it uses to describe trees.

2.4.1 Branching factor

In terms of the visual representation of a tree, recall that a state s is represented by a node. Labelling this node n, each action a available in s is represented by an edge (a "branch") emanating from n. If this edge connects n to another node n', then n' represents a state s' which is the result of playing action a in state s.

Definition 13. Branching factor of a node

The branching factor of a node n representing a state s is simply the number of actions available in state s. Diagramatically speaking, the branching factor of a node is the number of edges ("branches") emanating from node n to its child nodes.

Definition 14. Branching factor of a tree

Since the number of actions available may vary from state to state, the (mean) branching factor for a *tree* is the mean of the branching factors of each of the nodes in the tree.

UCT always tries each action once before starting to use its selection algorithm, so the branching factor of a state in the UCT tree will simply be the number of moves available in that state[2]. This branching factor gives us an immediate link to Thompson's concept of clarity. If a game has a very high branching factor, it may directly result in it being difficult for humans to understand. Equally, if a game has a very low branching factor, forward thinking strategies will become rather "narrow". This has two consequences: choosing the best move from few options may be too easy a decision, so the game may lose tension. A small branching factor may lead to the game following predictable patterns, making it repetitive and hence dull.

However, an alternative measure presents itself: perhaps a more interesting quantity would not be simply how many legal moves are available, but how many of those moves are plausible. Some moves will usually be instantly recognisable as weak, and hence not really worth considering at all – this leaves a remainder of plausible moves out of which the best must be selected.

The score:visits ratio used by UCT in deciding which move to play may be a good measure of plausibilty. For example, the branching factor of a node, K, may be large, but if $K - 3$ of the child nodes have negative scores associated with them, while the remaining 3 child nodes have positive scores, only these 3 moves are plausible in this position. Hence, in this example, while the node s has a branching factor of K, it has an *effective branching factor* of 3.

2.4.2 Tree depth

Branching factor can heuristically be thought of as the width of the tree, but another important metric is the *depth* of the tree, which is also sometimes referred to as tree

[2]providing the node has been visited at least as many times as it has moves available

height. In this paper trees will be represented with their root node at the top, so depth shall be used.

Definition 15. Depth of a node

The depth of a node n is the distance between n and the root node of the tree, n_0 (the number of edges on the unique path between n and n_0).

Definition 16. Depth of a tree

The depth of a tree T is the maximum depth over all the nodes in T.

The depth of a search tree may once again be related to Thompson's ideas of clarity and depth. If search trees for a game are very deep, the UCT algorithm is looking far ahead into the possible game continuations. This may indicate a game with good clarity – if UCT can see far ahead into the possible game continuations, perhaps even inexperienced human players will also be able to understand the likely future course of the game and make judgements about which is the best move to play. However, if the trees are too deep, this may imply that the game is too easy to analyse, and so the game may lack depth – inexperienced players may be just as good as experts.

2.4.3 Asymmetry

It is important to remember that a UCT search tree is by no means a box – it cannot be described by its width and depth alone. UCT trees have a unique asymmetry brought about by the algorithm's selective sampling approach, as shown in figure 2.8. How can this asymmetry be quantified?

Recall that UCT always adds one new node to its search tree per iteration, slowly expanding its tree boundary[3]. Whether this new node is added deep down an already-explored path or very near the root node depends the algorithm's current payoff estimates.

So the shape of the UCT search tree can be thought of as the combined effect of each of these node additions. This, in turn, means that the tree shape is a reflection of the current game position, and near-future potential game positions: it depends strongly on the number of moves available and their relative strengths.

For a game in which all moves are equally good in every board position, each of these moves will be inspected an equal number of times, and the UCT tree will closely resemble a breadth-first tree (see figure 2.9). For a game with only one good move in each position, UCT will focus on the single best continuation, so the UCT tree will be very thin and deep (see figure 2.10).

One approach to quantifying this asymmetry is to inspect the tree metrics for the level-1 sub-trees. Note that the level-1 sub-trees in figure 2.9 are all identical in shape. Those for 2.10 are very different in shape: one has been explored a lot, and is very deep; the others have been explored very little (because UCT estimates the moves to be weak), and will be very shallow.

Measures such as the branching factor and depth of each of the level-1 sub-trees go some way to quantifying the asymmetry and inner structure of the UCT tree. So will recording the mean and variance of the average score and number of visits to each child node at level 1. Garnering as many statistics as possible about the UCT trees gives the machine learners the greatest possible chance of successfully finding a link between tree shape and game quality.

The measures used to classify trees are discussed in more detail in the next chapter, and a full list is contained in appendix B.

[3]unless its sampling ends at a terminal node within the tree boundary, when the tree cannot be expanded further

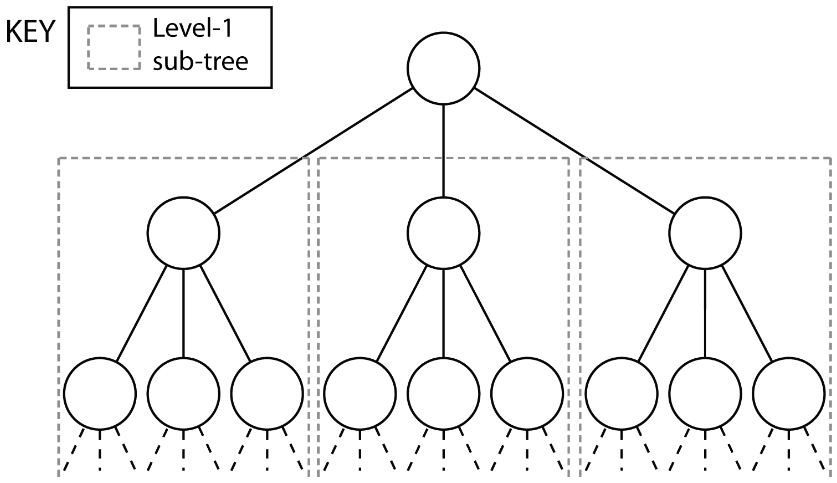


Figure 2.9: An example of a UCT search tree where each move looks equally good in each board position. All moves will be tested equally often, and the tree resembles a breadth-first search tree. Note that each level-1 sub-tree (indicated by grey dotted lines) is identical.

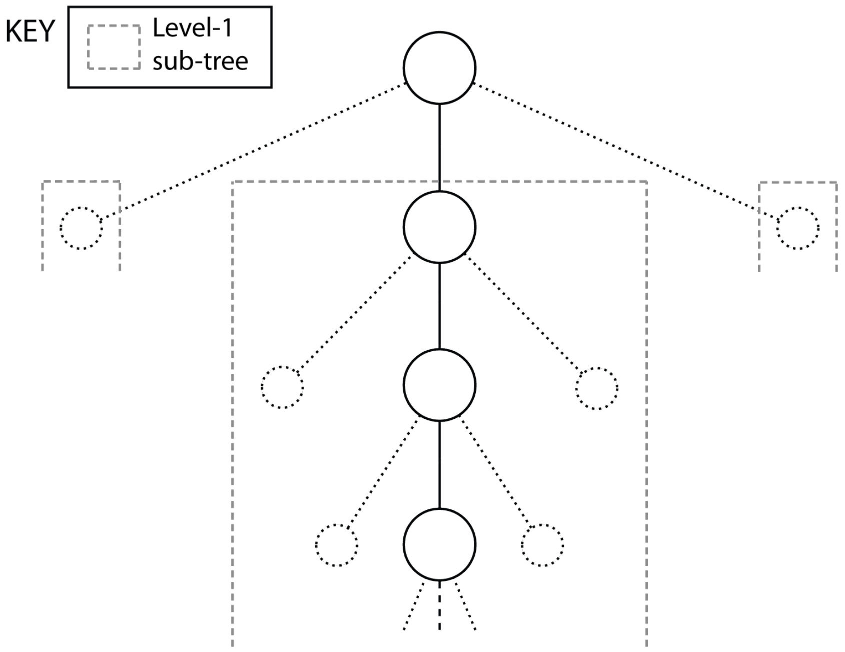


Figure 2.10: An example of a UCT search tree where only one move is worth playing in each board position. Poorer alternative moves (dotted nodes) are not visited often, and the tree becomes long and skinny. Note the difference in shape between the level-1 sub-trees (indicated by grey dotted lines). The sub-tree for the preferred move is very deep, while those for the alternatives have no depth, because these poorer moves are ignored by the UCT algorithm.

2.5 Machine Learning

Machine Learning (ML) focuses on how to create computer programs which can learn from experience. This ability is undoubtedly one of the things which makes us most human, and as such, machine learning represents a huge, important field of AI research. Great strides have been made in the field; machine learning systems have been applied in areas as wide-ranging as pneumonia mortality prediction[11], vehicle driving[12] and speech recognition[13].

In general, machine learning systems function by taking in a set of training data, comprised of attributes and targets. They then proceed to learn a relationship between these attributes and the targets. Once trained, they can make a prediction about the classification of a new case based on its attributes alone. Note that the prediction task may be a simple classification into one of many possible groups, or it could be a task of a more "continuous" nature, such as steering a car correctly based on video camera images of the road ahead. In the case of this research, the machine learning systems will be inspecting attributes in the form of numerical data about UCT tree shapes, and making a prediction about game quality or playability.

There were two key advantages of using a machine learning system for the task of predicting the quality/playability of board games:

- They can analyse large amounts of numerical data quickly and find patterns that humans might not be able to detect.
- Once trained, they can often analyse new cases based on their attributes and make a prediction far quicker than a human could (especially when working with large amounts of numerical data such as is the case for this research).

The second point will be vitally important if, as mentioned in the introduction, the ML system is used for analysing the playability of automatically-generated board games. It may be inspecting a very large test set (potentially several thousand games) which it will be able to analyse far quicker than a human could.

2.5.1 Finding a system which generalises well

To simply find a relationship between a set of attributes and targets can be done in a large number of different ways, and indeed, if working with numerical data, methods such as Taylor series can do this with as much accuracy as desired. The key feature of a machine learning system is not only that it should be able to learn a relationship, but that it should then be able to *extend this knowledge to new data it encounters* by inspecting the attributes of the new cases and making accurate predictions about their classifications.

When there is plenty of data available, the best way to train a machine learning system and understand its ability to generalise well to new cases is to have separate training, validation and test data. Learning is done on the training data; while this is done, the machine learner is constantly tested using the validation data. This gives a good indication of how the system's predictive performance would extend to new data, because the testing set upon which its performance is measured is not used for training at all.

Unfortunately, with the time constraints on this research, it would not be possible to create a large enough sample of games to keep back a validation set – all data would be necessary for training. Instead, leave-one-out cross validation was used to compare the performance of the various machine learning systems. This involves repeatedly removing one of the games from the dataset, and training a network on the remainder. Then the trained network's predicted classification for the left-out example is recorded. By repeating this as many times as there are samples in the dataset, a cross-validation matrix of the predicted against true classification values is formed. Learner performance is measured by the accuracy of this cross-validation matrix.

2.5.2 Avoiding overfitting

When training a machine learning system it is important to try to avoid overfitting. Overfitting occurs when the error on the training set becomes very small but the network performs poorly when classifying new data. It occurs because the network has been trained so much on the current dataset that it has become tuned to the exact details of this particular data, rather than learning rules and trends that generalize well to all cases.

When undergoing training, a machine learning system will usually improve its performance, as measured by its ability to predicting the classifications of the training data, as training time increases. However, after a time its predictive performance in terms of its ability to classify new data accurately will start to decrease – this is the point at which overfitting has occurred (see figure 2.11).

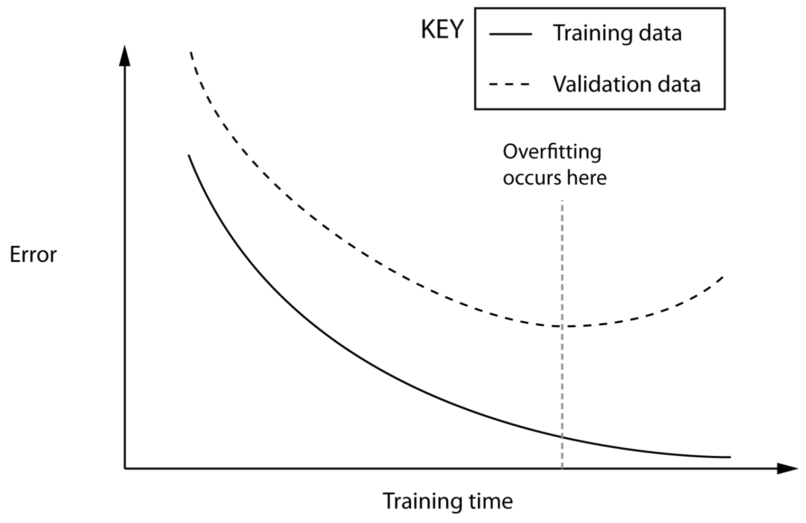


Figure 2.11: Chart demonstrating overfitting. Although error continues to decrease on the training set as training time increases, at a certain point error begins to increase on the validation set. At this point overfitting has occurred (the machine learner is starting to learn particulars of the training data rather than general classification rules) and training should be stopped.

A major advantage of using a training, validation and test set approach, as described above, is that it allows overfitting to be detected and avoided. The error on the validation set can be tracked as training goes on. At first, error will usually decrease as a function of training time on both the training and validation sets. When error on the validation set starts to increase, overfitting has occurred. Training is stopped at this point and the performance of the machine learner can be measured by its predictive ability on the validation data (or sometimes, if possible, further test data can be used to measure its performance).

Using leave-one-out cross-validation meant that this method of avoiding overfitting would not be possible. The specific approach used to avoiding overfitting is detailed in the Implementation section.

2.5.3 Measuring performance

The first question being investigated by this research was whether a machine learner trained on UCT data can accurately predict the quality of new games it encounters. In order to measure the performance of this machine learner, a measure of how close its predictions A machine learning system attempting this will output predictions on a continuous numerical scale. In order to measure the performance of a predictor of this kind, the correlation coefficient between the predicted and true target values will be used. A perfect classifier would have a correlation coefficient of 1, and the closer to 1 the correlation coefficient of the network, the better its predictive capability.

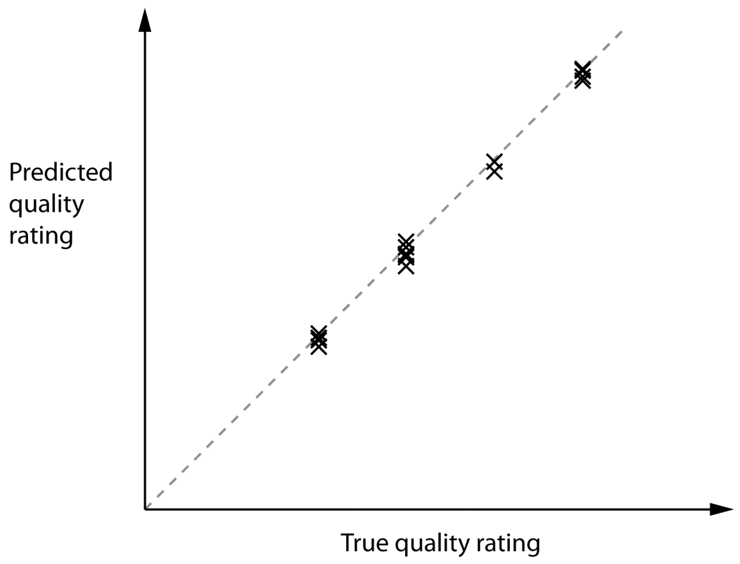


Figure 2.12: Example of the predictive performance that would be expected from a near-perfect machine learner in a continuous target space (such as is the case when predicting game quality). All points lie on or very near a straight line through the origin. This gives a correlation coefficient very close to 1.

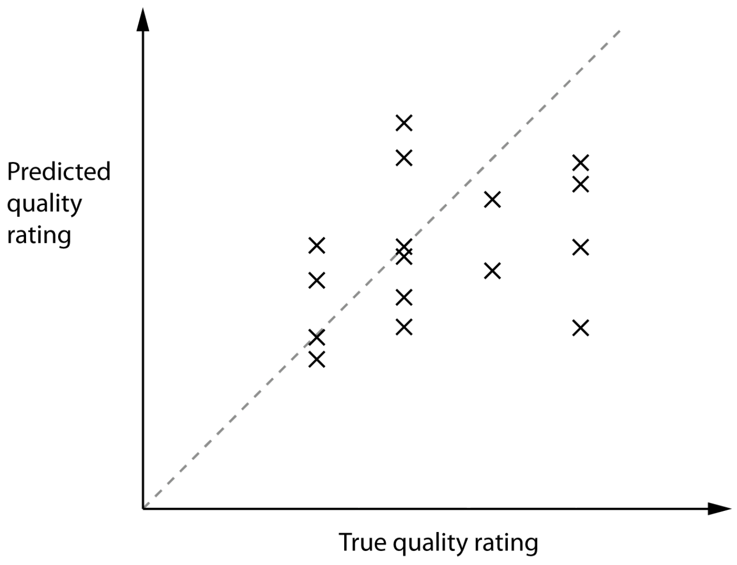


Figure 2.13: Example of the predictive performance of a poor machine learning system making predictions on a continuous scale. Predicted quality values do not reflect true game quality ratings, so the points do not lie on a straight line through the origin. The correlation will therefore be close to zero. Thus the correlation coefficient between true quality ratings and predicted quality ratings gives a good indication of the quality of the machine learner.

The second hypothesis being tested was: can a machine learner trained on UCT data differentiate between a real, playable game and a "broken" (unplayable) one? The second hypothesis represents what is called a decision problem, because the target function for the machine learner was now a binary classification (playable or unplayable) rather than making a prediction on a continuous numerical scale. The predictive ability of a machine learning system in a decision problem such as this is usually judged in terms of three measures: predictive accuracy, precision and recall.

Predictive accuracy simply measures the following: when presented with a new game, what is the likelihood that the machine learning system will classify it correctly? At first sight this would seem to be the only measurement of interest when quantifying the accuracy of a machine learner. However, several very different predictors can show exactly the same predictive accuracy, which is why precision and recall rates are also necessary to fully quantify the performance of a machine learner.

An example of different predictors exhibiting the same predictive accuracy is shown in table 2.1. This example supposes that playable and unplayable games occur in equal measure. Predictor ML1 simply predicts that every game is unplayable; predictor ML2 simply predicts that every game is playable. Finally, predictor ML3 predicts that a game is playable or unplayable with equal probabilty.

Note that each predictor has the same predictive accuracy – they each classify exactly 50 of the 100 examples correctly, and so if the ratio of unplayable to playable games in the population as a whole is also 50:50, they can each be expected to classify a new game correctly with probability of exactly 50%. However, the chance of ML1 classifying a playable game correctly is 0, as is the chance that ML2 classifies an unplayable game correctly. In contrast, the chance that ML3 classifies a playable game correctly is exactly 50%, because in 50% of cases it will (at random) choose to predict that the game is playable.

ML1:
predicts all games
to be unplayable

		Predicted		
		Unplayable	Playable	
True	Unplayable	50	0	50
	Playable	50	0	50
		100	0	100

ML2:
predicts all games
to be playable

		Predicted		
		Unplayable	Playable	
True	Unplayable	0	50	50
	Playable	0	50	50
		0	100	100

ML3:
randomly predicts
games to be playable
or unplayable with
equal probability

		Predicted		
		Unplayable	Playable	
True	Unplayable	25	25	50
	Playable	25	25	50
		50	50	100

Table 2.1: An example of predictors that use very different prediction rules, but exhibit identical predictive accuracy.

To recap, we have three prediction techniques here which have very different behaviour, but the same predictive accuracy. The difference between them can be quantified by their precision and recall rates, which are calculated separately for each of the possible classifications (in this case, for playable and unplayable games).

Precision measures the following: imagine that our machine learning system (using UCT tree data) classifies a previously-unencountered game as playable; there is always the chance that our machine learner has made a mistake – so what is the chance that this game is actually playable? The answer is given by the precision of the machine learner for playable games.

Conversely, recall measures the following: suppose a previously-unencountered example is presented to the machine learning system, and that this example is already known to be a playable game. Using only UCT tree data, what is the chance that the system correctly classifies the game as playable? This is given by the recall of the machine learner for playable games.

Similar statistics can be calculated for unplayable games. Precision and recall rates for the example predictors ML1, ML2 and ML3 are shown in table 2.2.

ML1		Unplayable	Playable
	Precision	50%	-
	Recall	100%	0%

ML2		Unplayable	Playable
	Precision	-	50%
	Recall	0%	100%

		Unplayable	Playable
ML3	Precision	50%	50%
	Recall	50%	50%

Table 2.2: Precision and recall rates for the three predictors shown in table 2.1. Precision and recall rates fully classify prediction performance.

A predictor with 100% predictive accuracy would show 100% precision and recall across all classifications, but this is often not possible in practice. Often a choice must be made between a classifier with high recall or high precision for the particular feature of interest. Which one is preferable depends on the application.

For example, in medical testing, it is vitally important that tests pick up all positive cases of a disease so that treatment may begin. So an initial testing technique must have a very high recall rate for positive cases of the disease. It does not matter if it has a low precision rate (i.e. it picks up several false positives) because often an initial test which shows a positive result is followed by a more expensive but more accurate test to confirm/refute the result. False positives can be picked up at this second test stage. However, to *miss* a positive case initially could be catastrophic, because the patient would be dismissed and the disease may be untreatable by the time they are seen again.

2.5.4 Balancing recall for playable and unplayable games

The likely application of this research is to assess automatically-generated board games for playability. In many ways the machine learning system will take on the same role as an initial medical test: it will be presented with several (possible many thousand) automatically-generated games, and must give an initial prediction of the quality/-playability of each. Its role will be to pick out potentially playable games this test set, and these games will then be checked by humans to confirm whether they are playable, and how interesting they are.

In one sense, high recall rate for playable games is preferable in this situation, since this will mean that few genuinely playable games will be rejected at this early stage. It would be very unfortunate for brilliant game to go undiscovered because it is misclassified by the machine learning system and hence ignored.

However, in another sense, high recall for *unplayable* games is important, so that the machine learning system accurately removes as many unplayable games as possible from the test set. Machine learner ML2 above, if it were given this task, would certainly not miss any playable games, but it would not remove any unplayable games from the set, so in that sense it would be useless. A classifier with good recall for unplayable games will allow for many unplayable games to be identified accurately and removed from consideration, leaving less work to do during the human checking process.

Further discussion of this balance between recall for playable and unplayable games, and which is more important for a machine learning system identifying playable automatically-generated board games, is contained in chapter 5, Evaluation.

Chapter 3

Implementation

With the background to the UCT algorithm, the game quality concepts and the tree metrics that would be used in place, the implementation of the research could begin. The coding required to perform the main body of the work falls into two parts – the programming of the UCT algorithm itself and of the games upon which the UCT algorithm would be used. All implementation took place in C++.

A complete game-playing system was created which was able to work with UCT or human players. It was designed to be as extensible as possible – other AI techniques could easily be added as extensions of the ComputerPlayer class. It collected data on all trees produced, outputting these to the terminal or into text files in a suitable format for aggregation. If desired, it could translate each tree structure into ".dot" format, meaning that the tree could be viewed using the visualisation software GraphViz. A UML diagram for the system is shown in figure 3.1.

3.1 UCT Algorithm

As described in the previous section, UCT works by iteratively growing a search tree; for each node in the tree its number of visits and score is recorded. Each node in the tree is associated with a board position, and each edge can be associated with the move that takes the game from one board position to another.

To implement the algorithm, a Node class was defined with the necessary member variables to construct the tree and record UCT data. Pointers to child, sibling and parent nodes are used for navigating through the tree, and variables record the number of visits and the total score for the node. Pseudo-code for the algorithm is shown in figure 2.6.

One design choice should be explained here. In the context of playing games, search trees are usually described as shown in figure 3.2. A node can have several children, and from a programming perspective the links between a parent and its children can be represented using pointers. This is a perfectly natural and sensible way to understand a search tree. However, to implement a tree in this way is inefficient. An array of pointers to child nodes must be maintained for each node in the tree. Given that at any moment several of the tree's nodes will be leaf nodes with no children, this setup wastes large amounts of memory.

It is more efficient for each Node object to contain just three pointers: one pointing to its parent node, one for a child node, and a nextSibling pointer which points to a node's younger sibling if one exists. Children of any node are effectively grouped into linked lists by using this setup (see figure 3.3). If a node has more than one child, its first child is accessed through its child pointer, its second child is accessed by using the

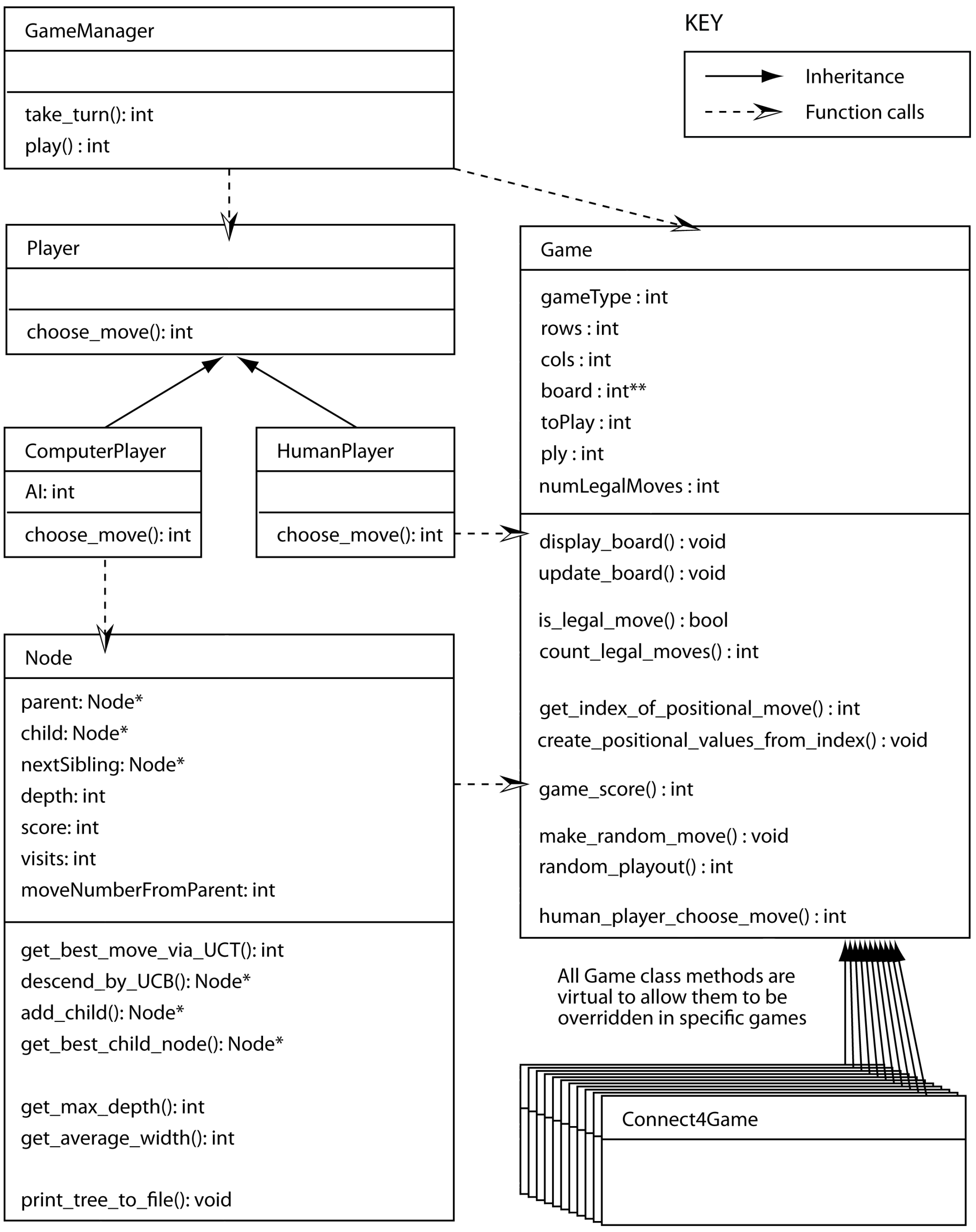


Figure 3.1: UML diagram for the game-playing system created for the project. All individual games inherited from a parent Game class, which implemented its functions virtually to allow them to be overriden as necessary.

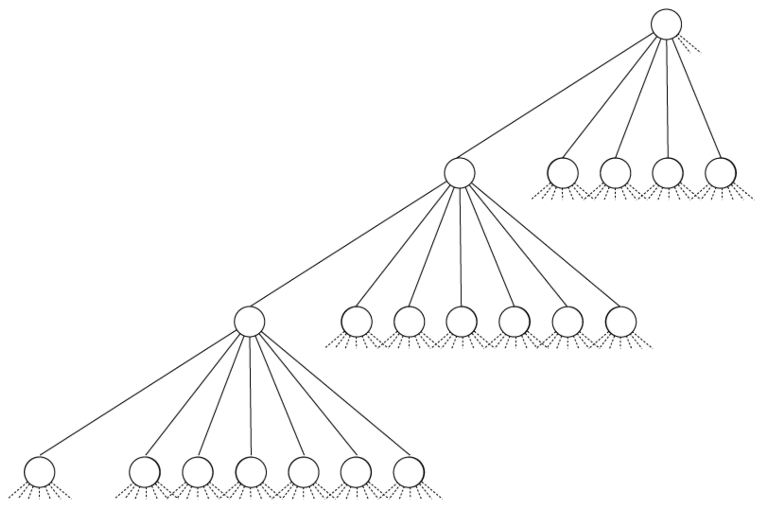


Figure 3.2: The classic implementation of tree search. The tree is formed of nodes; each node has an array of pointers to child nodes, and one pointer to its parent node. This leads to a great deal of wasted memory and reduced performance. Note that in this example, this setup has 19 pointers in use (solid lines) and 123 unused pointers (dotted lines)

nextSibling pointer of the first child, its third child is accessed by using the nextSibling pointer of the second child, and so on.

This saves large amounts of memory for games such as Go where there are more than 350 legal moves in some positions. The implementation used guarantees that each node only has three pointers (one to its parent, one for a child node, and one for a sibling node) and hence wasted memory is kept to a minimum.

It was important that the algorithm should run as quickly as possible. The experimental phase of the research would involve playing out 100 instances of each game with UCT playing against itself, and the UCT algorithm would be using up to 100,000 iterations to make each move. If a move takes 1 minute to execute, then a typical instance of 100 plies would last 100 minutes, meaning that 100 playouts would last approximately a week. Even with large amounts of computing power available, this was not acceptable, so targets were set for its execution speed.

The target for the implementation of the UCT algorithm was 10,000 playouts per second on any game, meaning that even a 100,000-iteration UCT player would play a move in 10 seconds. As shown in the pseudo-code in figure 2.6, the algorithm can be split into three major parts: descent through the tree, performing a playout game, and updating scores for nodes. It would be important to understand which of these parts of the algorithm was taking the longest in order to know where to focus efforts to keep its execution time low.

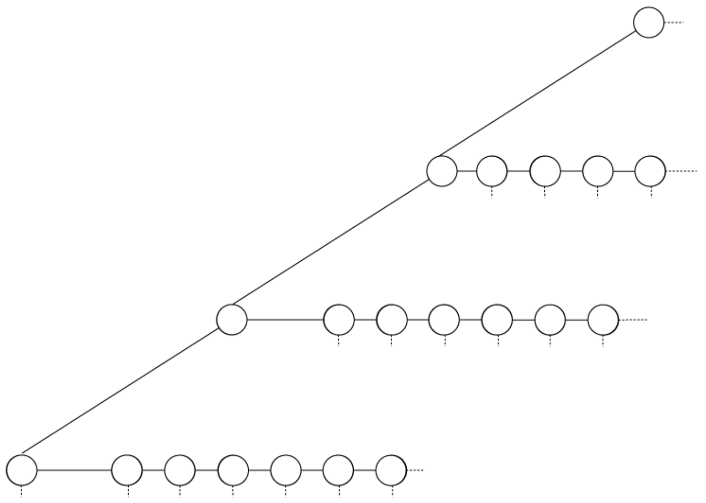


Figure 3.3: More efficient implementation of tree search. Each node has just three pointers: one to its parent, one for a child node, and one for a sibling node. This allows the same tree structure to be created with vastly reduced memory usage. This figure represents the same tree as figure 3.2, yet here 19 pointers are in use (solid lines), with only 21 unused pointers (dotted lines)

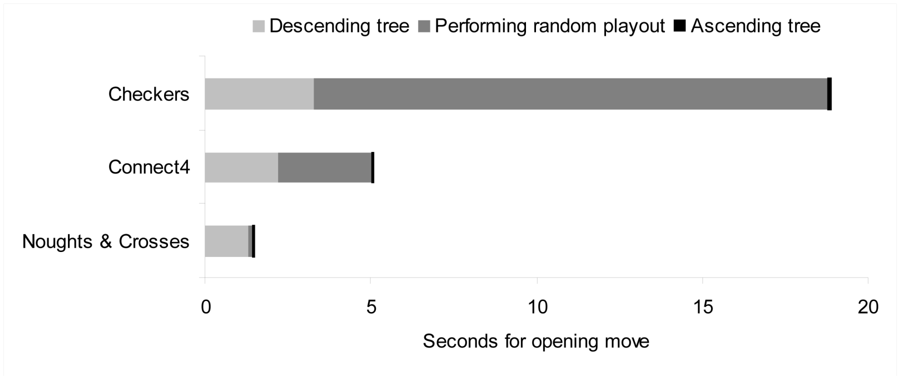


Figure 3.4: Chart showing times taken by different parts of the UCT algorithm, for the opening moves of a short, medium and long game (Noughts & Crosses, Connect4, and Checkers respectively). The random playouts are clearly the part of the algorithm that takes the longest, and so were focused on when trying to improve the algorithm's speed.

As shown in figure 3.4, early tests showed that the random playouts took far longer than any other part of the UCT algorithm for medium to long games. These would be the games where the most focus would need to be placed on algorithm speed in order to keep game times low. The random playout games were therefore the main area where improvements would bring about quicker UCT play. This part of the algorithm calls the random_playout() method of the game class, which in turn makes several calls to the count_legal_moves(), update_board() and game_score() methods; so all of these methods were examined for possible efficiency gains when quicker playouts were needed. Explanation of the programming of the individual game subclasses and the techniques used to keep playouts as brief as possible is contained in the next section.

3.2 Games to be analysed

The majority of the work associated with the implementation was associated with coding up a variety of games for the UCT algorithm to play. To make the analysis worthwhile it would be necessary to code up coding up several games so would it be possible to collect enough data to give the ML systems a chance of identifying a relationship between UCT tree shape and game quality. Luckily, the UCT algorithm is very flexible: it can play any game, provided it is presented with a few key functions for playing that game, as described in the previous section.

Encapsulating the rules and dynamics of a game in code can be difficult, but the mechanics of the UCT algorithm itself provided a guide as to how this should be done. Recall the information needed by the UCT algorithm for each step in its execution:

- To choose a move in a given board position it must know what legal moves are available.
- To link board positions together it must know how to update the game board between moves (for example, removing a captured piece in the game checkers).
- To generate its scores it must know when a playout game has reached a conclusion, and whether the game has ended in a win, loss or draw.

So it was necessary for each game to have member variables and functions available to achieve these three tasks: finding legal moves; updating the board position based on a given move; recognising when the game has finished and what the result is. Presented below is an abstract base class Game which contained these functions (some virtual) – all individual games were derived from this base class.

A summary of the methods implemented in the base class Game is shown overleaf.

```
int gameType;    // integer to identify the game

int rows;        // the size of the board
int cols;

int** board;     // 2D array of integers
                 // which holds the board state

int toPlay;      // the player whose turn it is

int ply;         // how many plies have passed
                 // in the game

int numLegalMoves;  // the number of legal moves
                    // available in the position
```

Figure 3.5: Member variables in the Game class. These variables are common to all games regardless of type, and are used to keep track of the game state and facilitate playouts for the UCT algorithm. Further game-specific member variables were implemented for some games.

The update_board() method updates the game state when a player makes a move. This involves updating the board positions; it may also require updating some member

```
// Functions vital to the UCT algorithm

// Implemented pure virtual,
// must be defined in subclasses
void update_board();
int is_legal_move();
int game_score();

// Implemented in parent Game class
// but can be overriden in subclasses
bool count_legal_moves();

// Only implemented in parent Game class
void make_random_move();
int random_playout();
```

Figure 3.6: Methods vital to the UCT algorithm defined in the parent Game class. Some were implemented as pure virtual functions, which ensured that they were properly defined in each specific subclass

variables in order to implement the mechanics of the game properly.

The is_legal_move() method takes in a proposed move described by positional values (x and y coordinates) and returns whether the move is legal in the current game state.

The count_legal_moves() simply counts the number of legal moves available in the current board position. This number is required by the UCT algorithm to build its search tree and to perform its random playouts. The moves available in a given state are calculated deterministically and are always returned in a given order.

The game_score() method looks at the current game state and calculates if the game has terminated. If so, the method returns the game score (1 if player 1 has won, 0 if a draw, -1 if player 2 has won).

The random_playout() method is used for executing random playout games; it calls the make_random_move() function several times during each playout. Each iteration of the UCT algorithm makes one random playout.

```
// Other functions defined in Game class

// Can be overridden as necessary in subclasses
void display_board();

int get_index_of_positional_move();
void create_positional_values_from_index();

int human_player_choose_move();
```

Figure 3.7: Other methods implemented in the parent Game class.

The display_board() method simply displays the current board position on the screen.

The middle two methods are for converting a move described by positional values (in terms of x and y locations) into a unique move index (between 1 and the number of legal moves available), and vice versa. This indexing is used by the UCT algorithm to interface with the other functions defined above. For example, the UCT algorithm returns a move index in the form of an integer, but the update_board() function takes positional arguments.

The x and y locations may simply describe a single chosen square (e.g. Noughts & Crosses, Othello); it may be necessary for the functions to take arguments for both "to" and "from" locations for games when pieces are moved across the board (e.g. Checkers, Breakthrough); sometimes a further argument may be used to specify any options associated with the move (e.g. removing an opponent's piece in Nine Men's Morris).

The human_player_choose_move() method is to allow human players to take part in games against UCT via the terminal. This was vital for debugging the system during development.

Some of the functions in figure 3.6 had to be implemented separately in every subclass (in the code for each game). For example, the is_legal_move() function depends uniquely on the game being played, as does the game_score() function which calculates if the game has reached a conclusion. These functions would be implemented in a pure virtual manner to ensure that they were correctly implemented for each individual game in its source code.

However, it was important to try to minimise the amount of coding required as far as possible, by defining functions in parent Game class which could be called by several specific games. This allowed new games to be added quickly. The count_legal_moves() function is an example of this: the code for this function as defined in the game class is shown in figure 3.8. This function definition shown could be used by any game where a move is defined by a single (x,y) location, such as Noughts & Crosses, Connect4 and Othello.

```
// Counts the legal moves for player
// to play in the current position
int Game::count_legal_moves() {

    int number = 0;

    for(int i=0; i<maxCols; i++) {
        for(int j=0; j<maxRows; j++) {
            if(is_legal_move(i, j)) {
                number++;
            }
        }
    }

    return number;
}
```

Figure 3.8: Definition of the count_legal_moves() function in the parent Game class. This code is very extensible, and could be called by any of the games where a move is defined by a single (x,y) coordinate, provided the is_legal_move() function is defined properly. However, in order to achieve acceptable playout speeds, for some games this function was overriden in the subclass with of a more specific, quicker method.

When speed improvements were required for a particular game, functions such as count_legal_moves() could be overriden in its source code. For example, in the game of Checkers, counters only reside on the dark squares on the board, so a simple improvement to this "vanilla" count_legal_moves() function is to only inspect dark squares when counting legal moves.

Some games required much more than overridden functions to achieve acceptable playout speeds. Nine Men's Morris is a game played on square board, with connections between the different positions on the board as shown in figure arrangement of linked nodes as shown in figure 3.9. Each player begins the game with 9 stones, which are placed on the game board alternately until none remain. Then players proceed to move their stones from one location to a neighbouring location (along one of the lines drawn on the board). The object of the game is to create lines of three stones in a row; when this done, the player may remove one of their opponent's stones. When a player has fewer than three stones remaining, they lose the game.

The game dynamics of Nine Men's Morris meant that when using a basic 2-D array of board positions (which worked prefectly adequately for other games) a 100,000-iteration UCT player would take 9.4 seconds to make a move. Game playouts with 100,000 iterations per move would simply not be possible under this setup.

Efficiency gains had to achieved, but the approach to doing so required more thought than for games such as Checkers. In checkers, pieces are only allowed to occupy dark squares on a standard chess board, which is simple to quantify: pieces may only lie in positions (x,y) where $x + y = 1 \mod 2$. The positions where a piece may legally be located on a Nine Men's Morris board have a strange symmetry. There is no easily quantifiable pattern determining legal piece locations. Further, each legal piece location has its own unique group of neighbouring locations to which it may moved.

The game board for Nine Men's Morris is a graph – the same sort of graph that was described in definintion 1 of chapter 2. The board can be thought of as a collection of nodes and edges, and in fact this is by far the best way to represent the game in code.

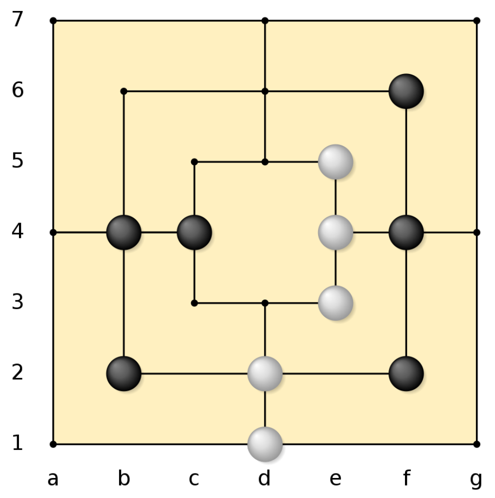


Figure 3.9: The game board used for the game Nine Men's Morris[14]. The unusual layout and game dynamics meant that the game had to be programmed very differently from others. An adjacency matrix was defined to allow quick calculation of available legal moves

The board was defined in terms of an array of 26 nodes. An adjacency matrix could be used to quickly determine all neighbours of a given node, facilitating the process of counting legal moves.

While this approach meant there were some overheads when instantiating a game of Nine Men's Morris, the speed advantages during game playouts were considerable. The difficulty of dealing with the board layout disappears when treating it as a graph. This change in implementational approach reduced the time taken for a UCT player to make a move by a factor of approximately 30, allowing even a 100,000-iteration player to make its first move of the game in around 25 seconds.

In total, 17 playable games were coded, details of which are contained in appendix A. The unplayable games created are described in the next section.

3.3 Defining unplayable games

To give the machine learner several examples of unplayable games, adjustments were made to the already-programmed group of games to "break" them (make them unplayable). For example, the well-known game of Connect4 had its winning condition changed to six counters in a row (instead of four). Since the original board size of 7x6 was retained, a win became so difficult to achieve that the game became unplayable.

Other games were adapted in similar ways to make them unplayable. This adaptation process allowed several broken games to be created quickly with minimal extra coding. A game was considered unplayable if one of the following conditions held:

- Drawish: the game is too hard to win for either player
- Biased: the game is too easy to win for one of the players
- Too long or too short: the game lasts too long or finishes too quickly

After "breaking" some of the games, 11 broken and 17 playable games were presented to the machine learning techniques. Shown in table 3.1 is a list of the unplayable games created, how they were broken and the reasons they were considered unplayable.

Game broken	Reason	Rule change and description
Noughts & Crosses	Short, biased	Winning condition changed to two-in-a-row. Player 1 can always win after two moves.
Achi	Short, biased	Board size changed to 4×4, with winning condition still 3-in-a-row. Player 1 can always win after 3 moves.
Oware	Drawish	Players only harvest if they place seeds in an opponent's field, bringing the total to 5 seeds (usually 2). This made it very difficult to capture seeds and win the game. 20/20 games played ended in draws.
Kalah	Biased	Players cannot use their own store for harvesting. This was expected to bring about a drawish game, but instead made the game biased in favour of player 1. Player 1 won 20/20 games played.

Table 3.1: A list of the games which were broken for the purposes of analysing game playability. Data on game results is based on the 20,000-iteration UCT algorithm playing itself unless stated otherwise.

Game broken	Reason	Rule change and description
Alak	Long, drawish	Moves which immediately repeat the position of the previous turn are allowed. This made it impossible to force a win, since the player behind could simply repeat the previous turn's position rather than moving into a weaker one (20/20 games ended in draws).
Dipole	Biased	The relationship between the size of stack and the number of squares moved by the stack was inverted. This led to a futile game of damage limitation where it was difficult to avoid moving counters off the board, leading to a loss. Player 2 won 20/20 games played.
Checkers	Long, drawish	All pieces start as kings. There is little benefit to moving pieces forward, since all pieces are already kings. Since kings are more manoeuvrable than pawns, it is very difficult to capture an enemy king without giving one away too. 20/20 games ended in draws.
Nine Men's Morris	Short / Long, drawish	The player who makes the first mill wins. This was intended to make the game short, which was successful: games played between lower-strength UCT players ended quickly (20/20 games lasted less than 10 turns for 1,000-iteration UCT). However, games between higher-strength UCT players would reach stalemates due to strong defense (20/20 games ended in draws for 100,000-iteration UCT)
Othello	Short, biased	Stones may only be laid next to the opponent's most recently placed stone. This reduced the number of available moves considerably. Since Othello ends when neither player can make a legal move, it shortened the game (20/20 games ended in less than 10 turns). The game also became biased (20/20 games ended in a win for player 1).
Connect4	Drawish	Players must achieve 6 counters in a row to win. The same board size was used as for standard Connect4, and so the game became very difficult to win (19/20 games ended in draws).
Qubic	Drawish	Players must achieve 2 distinct lines of 4-in-a-row to win. The same board size was used as for standard Qubic; this made the game very difficult to win (17/20 games ended in draws, compared with 0/20 draws for standard Qubic).

Table 3.1 (cont'd): A list of the games which were broken for the purposes of analysing game playability. Data on game results is based on the 20,000-iteration UCT algorithm playing itself

3.4 Data extraction

3.4.1 Statistics used to quantify UCT tree shape

Each move played by the UCT algorithm produces one search tree. So once the algorithm was working for the selection of games described in the previous chapter, it was simple to produce large numbers of search trees very quickly – playing the UCT algorithm against itself several times produced one tree for each move of each game. As discussed in the previous chapter, the shapes of the trees created by the UCT algorithm had to be distilled into numerical data in some way for this shape to be analysed by the machine learning systems. So each tree had to be summarised by a set of statistics.

In one sense it was sensible to use summary statistics which might be related in some way to the game quality concepts described in section 2.3. However, it was important not to be too selective: it was also desirable to present the machine learning systems with as many statistics as possible. This would give the machine learners as much information as possible about the shape of the UCT trees, increasing the chance that they would find a relationship between tree shape and game quality.

Sixteen distinct statistics were used to summarise the trees. To collect all this data, functions were added to the Node class which recorded the desired statistics for each tree produced by the UCT algorithm. Once this was done, 100 UCT versus UCT instances of each game were played out. Each instance provided several trees, so in total, statistics from several hundred trees were collected for each game. The features most closely related to the game quality measures mentioned in chapter 2.3 are discussed below; a full list is available in appendix B.

Number of legal moves available (branching factor)

This may relate to the ideas of clarity and depth. A game with only two moves available in each position is likely to lack depth and be repetitive, and hence may be boring to human players. On the other hand, a game with over 100 legal moves may be too complicated for human players to enjoy.

Tree depth

The depth of the search tree measures how deeply into the game position UCT is seeing. This is once again related to clarity. If UCT can only see a couple of moves ahead in the search tree, will the game lack the clarity desired by human players?

Recall that UCT's asymmetric search means that the tree depth is not only a consequence of the branching factor, but is also affected by the relative strengths/weaknesses of each move analysed. If a tree is shallow despite having a low branching factor, this indicates that all moves look equally promising, which is undesirable since it removes tension from the game.

The score:visits ratio of the best available move

This is measured on a scale of 1 (immediatedly winning move) to -1 (immediately losing move). When one player is in a better position and another is in a worse position, the scores of their best available moves will be positive and negative respectively. So games which end quickly in a win will exhibit large variance in best score from positive (winning player) to negative (losing player).

More balanced games will not have such a high variance: in the extreme case, a game with no possible win will show no variance at all – all moves including the best move will have a score of zero. A good game will be somewhere in between: best scores remaining near zero for some of the game, but then diverging as one player gains the upper hand. As such, keeping track of this score may give clues as to the drama and decisiveness of the game.

The spread in score:visits ratio between the best and worst available moves

A game where the worst move's score is often very close to the best move's score will certainly lack decisiveness and be boring for human players – in this situation choosing the worst move may not result in sufficient penalty to make the game interesting.

Another similar measure used was the spread between the best move and the $\frac{n}{3}$th move, where n is the number of legal moves available. This measure analyses the difference between the move which is "top of the list" of available moves and the move which is "a third of the way down the list". This top third of the moves is where UCT and strong human players will focus their thought during their decision process. If there is a lot of difference in score, the game may lack drama, because choosing a move which is not the best will lead to a quickly lost position. If there is a very small difference in score, the game may lack tension, since choosing the 3rd best move instead of the best will have little difference on the outcome.

Mean of level-1 score:visit ratios

If this mean is too close to the best move's score:visits ratio, we are dealing with a long-tailed distribution where there are, on average, several good moves available in a given position. A game with this feature will not be interesting because there may be little to choose between the good moves: it may be too easy to play good moves, and hence the game will lack tension.

Fraction of leaf nodes and fraction of UCT nodes

For the purposes of this analysis a leaf node is defined to be one which has been visited less often than the number of legal moves available in its game state (branching factor). This means that it still has children unexplored by the algorithm, so in this way it is at the edge of the search tree. Conversely, a UCT node is any node which is not a leaf node – so named because once a node has been visited more times than its branching factor. Heuristically, these UCT nodes tell us how much "trunk" the tree has, in comparison with the amount of leaves.

These two fractions give more clues as to the shape of the search tree. If several of the nodes in the tree are leaf nodes, this can be thought of as a "spindly" tree where each branch contains many leaves. This may indicate a game with low clarity, since UCT has a lower understanding of leaf nodes than it does of its internal UCT nodes. Conversely, if the tree has a low proportion of leaf nodes and a high proportion of UCT nodes, this is akin to a strong oak tree where the majority of the mass is contained near the base. Games with this sort of tree are the ones UCT is likely to understand best, and may be an indication of games which are highly playable.

Various statistics for each sub-tree of the child nodes at level 1

As discussed in chapter 2.4, these statistics may give key information about the internal structure of the search trees. Insight into the asymmetry present among these sub-trees may give information about the levels of drama and tension within the game. Does choosing a slightly different move result in a radically different game? If so, we would expect a lot of variation among the sub-trees at level 1. This may indicate a very playable game with high drama and little repetition.

The node class, whose methods were responsible for the creation of each UCT tree, was the natural place to put the methods which would extract data about these trees. Trees are very important data structures in computing that are used for all sorts of purposes, and many good methods exist for iterating over their nodes. UCT trees are no different in this sense, and their structure could often be leveraged to extract data efficiently with minimal extra coding by using techniques such as recursive functions. The function for calculating the average branching factor of a tree is shown in figure 3.10.

```
// RETURNS THE AVERAGE WIDTH OF THE TREE,
// I.E. THE TOTAL NUMBER OF LEGAL MOVES
// AVAILABLE OVER ALL NODES DIVIDED BY
// THE NUMBER OF NODES.
// TO USE, CALL FROM ROOT WITH ARGS (0,0)
double Node::get_average_width(int& branches, int& nodes) {

    nodes += 1;
    branches += numLegalMoves;

    if(nextSibling != NULL) {
        nextSibling->get_average_width(branches, nodes);
    }
    if(child != NULL) {
        child->get_average_width(branches, nodes);
    }

    return double(branches)/nodes;
}
```

Figure 3.10: Recursive function used to calculate the average width of a tree. The function iterates over every node in the tree, keeping track of the total number of branches and number of nodes. Finally it returns the quotient of the two.

Once calculated, all the statistics describing each tree were output to Comma Separated Value (CSV) files. This created many megabytes of data which had to be aggregated into a set of attributes for each game before the machine learning could begin. The next step was to aggregate this data. For each game, the maxima, minima, means and variances of each tree statistic were calculated. It was this data that would finally be used as attributes in the machine learning task. If desired, further splits could be incorporated at this aggregation stage. For example, splitting the data between moves for player 1 and player 2 might give an indication of a biased game.

3.5 Data analysis

Matlab was used for data aggregation and training each of the machine learning systems. It provides a comprehensive statistical analysis package that was perfectly suited to the requirements of this research, and has a comprehensive API which allows the user to define their own functions as necessary. Functions were written for importing and aggregating the data from CSV files, training and testing each of the machine learning systems.

Three machine learning systems were tested: Artificial Neural Networks (ANNs) were used to learn and predict game quality; for the task of predicting game playability, ANNs, Decision Trees and Support Vector Machines (SVMs) were used.

3.5.1 Identifying a best network shape

One machine learning system was tested for its ability to predict game quality: neural networks. One aspect of working with neural networks is that there are a large number of parameters that can be adjusted, including network shape, training time and transfer functions. To exhaustively test all combinations would not be possible. Neural networks can also be difficult to interpret – it is not always possible to distill their classification process down into a functional or axiomatic form that humans typically use for such classification tasks. This is especially inconvenient in the current line of research since it is hoped not only to find a link between UCT trees and game playability, but for that link to be understandable in order that further investigations may be made into it.

To try to increase the chances of finding a network which could be understood from a human point of view, focus was placed upon finding the best shape of network for this learning task. If a particular shape of network could be shown to have strong predictive capability, and to be distinctly better than other network shapes, time could be spent interpreting the weights of this "best" network. With enough investigation, some rule-based interpretation of the network's inner workings might be achievable.

So wrapper functions were written in Matlab to test a variety of different network shapes. These wrapper functions also offered a way of avoiding overfitting. Neural Networks are trained over a number of epochs, after each epoch a negative feedback system (using feedback from their predictive performance on the training data) called backpropagation is used to adjust their internal weights and reduce prediction error. If this process continues for too long though, they can easily become overtrained to the point where their ability to classify new cases is reduced. When using a training and validation sets, overfitting can be avoided by ending the training of the neural network when prediction performance begins to increase on the validation set.

Due to the low number of games coded up, the training and validation set approach was not viable, so leave-one-out cross-validation was used instead. Overfitting was avoided by training each each network shape using variety of different training times. If overfitting was occurring for some networks (which were being trained for too long), then the cross-validation process should show better results for the same network shape trained for a lower number of epochs.

Finally, each [network shape, training time] combination had its cross-validation repeated 10 times, and performance was measured on average over these 10 repetitions. This was to negate the effects of random initial conditions used for each network, with the aim of finding a network setup which consistently outperformed its peers. Network performance was measured by inspecting the correlation of the predicted and true game quality values, as described in section 2.5.3. A perfect classifier would have a correlation coefficient of 1, and the closer to 1 the correlation coefficient of the network, the better its predictive capability.

3.5.2 Identifying most relevant UCT tree features

Three techniques were tested for their ability to predict game playability: neural networks, SVMs and decision trees. Using three techniques increased the chance of successfully finding a relationship. Furthermore, it increased the chance that if a relationship were found, it could be deciphered and understood. A decision tree is far easier to understand than a neural network; its decision-making process can easily be boiled down to a first order logic formula. Such a formula might provide an interesting springboard for further work and might even contribute to techniques for better automatic game generation in the future.

Matlab also contains functions for printing out decision trees created during the cross-validation process. These trees were then inspected in order to ascertain which UCT tree features were the most commonly used for making classification. If some UCT tree features are strongly related to game quality, training machine learning systems on *only* these features in the future might produce strong predictive capabilities. Ideally, these key UCT tree features might even be used in a generative capacity to help in the creation of interesting board games. Further, each node's decision process was studied to see if it made sense, specifically in relation to the game quality concepts discussed in section 2.3. The results were very interesting, and are discussed in more detail in the next two chapters.

Chapter 4

Results

Chapters 2 and 3 have given background to UCT, game quality concepts and Machine Learning systems, and described the implementation of the algorithm. This chapter contains the results of the research, beginning with some visual examples of the UCT trees being produced. After that follow charts showing the cross-validation data produced by the machine learning systems. Full tables of all the results are shown in appendix C.

4.1 Examples of UCT trees

The object of the research is for a machine learning system to be able to distinguish between good and bad (or playable and unplayable) games by UCT tree data alone. Some learning problems that humans find easy are very difficult for ML techniques to understand; others are far easier for a machine learner than a human. Could a human succeed at this game quality task – identifying good games by inspecting UCT trees?

Below are some examples of UCT trees produced by the game playing system. These examples all have the same format: they show UCT trees based on 5,000 iterations, with the size of a node representing how many times it was visited by the algorithm. This number is identical to the number of times the action leading this node was attempted, which is written next to that branch in the tree. The nodes representing weak-looking moves are represented by dotted lines, and the children of these nodes are omitted due to lack of space.

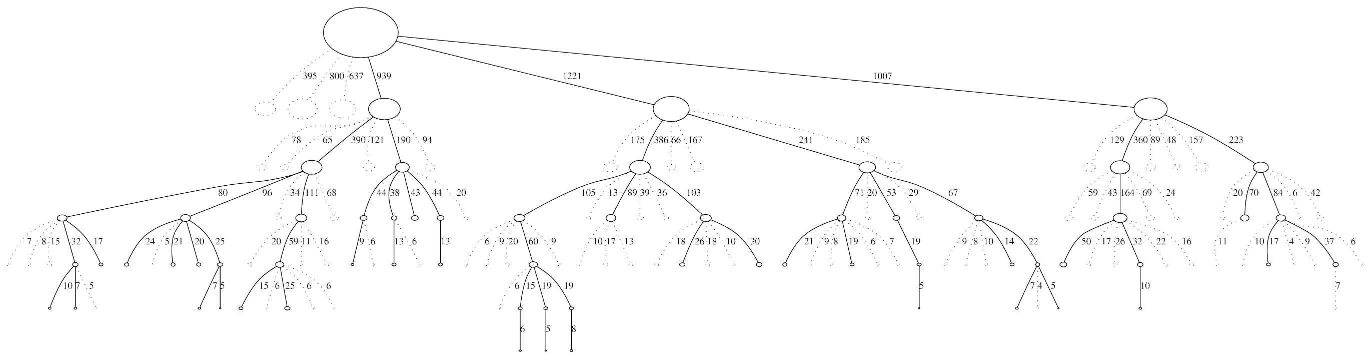


Figure 4.1: Example of a UCT search tree produced for the game of Oware, a game rated as very good quality.

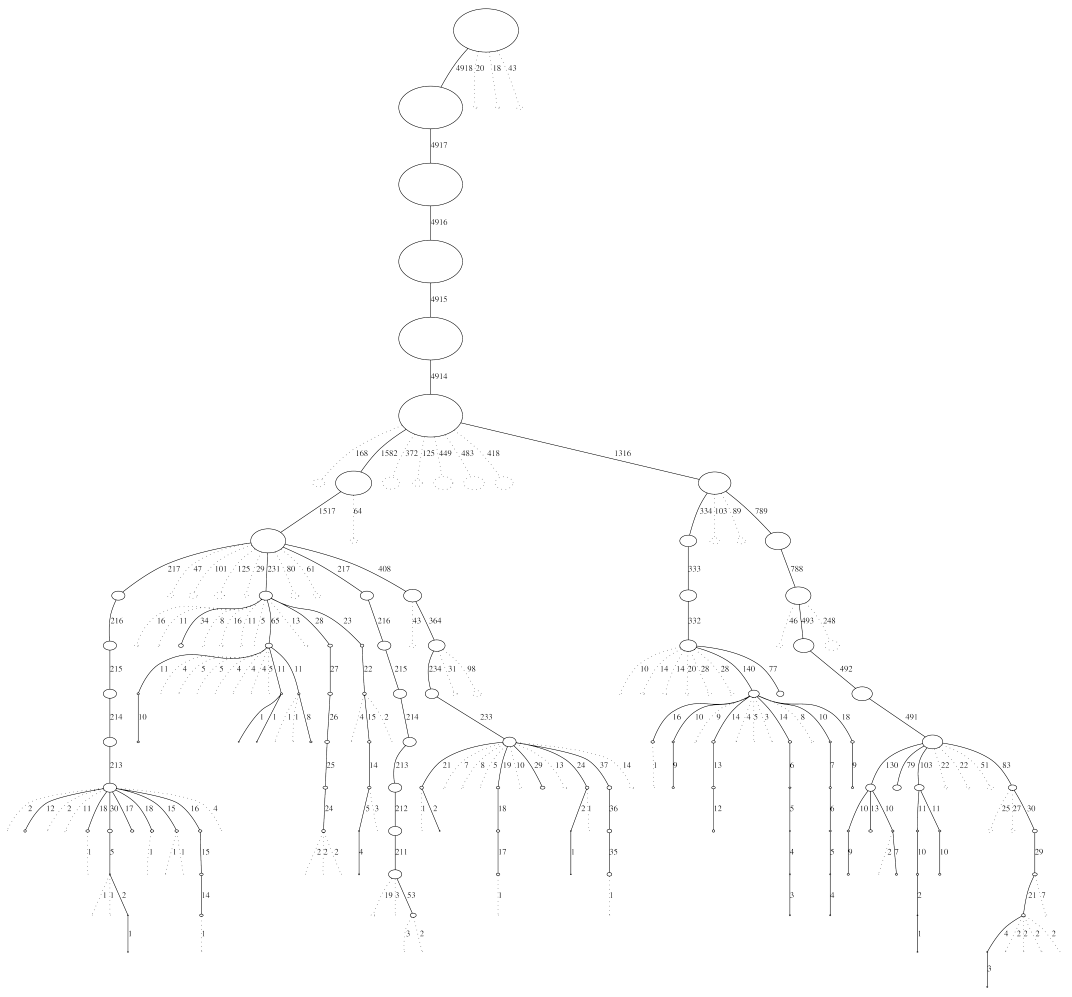


Figure 4.2: Example of a UCT search tree produced for the game of Qirkat, a game rated as medium quality.

The UCT tree for the game of Qirkat (figure 4.2) is very deep, due to the fact that in Qirkat certain moves are forced. The tree for the game of Oware (figure 4.1), rated a much higher quality game, is shallower and more “balanced”. However, it is unlikely that one would, in general, be able to make judgements about the quality of a game from these UCT trees.

One might suppose that it would be easier to differentiate between a playable game and a broken game based on its UCT trees. Below are shown two UCT trees for the game of Othello: in its original form (figure 4.3) and broken form (figure 4.4). There is a stark difference between the two trees. Reducing the number of available moves in the broken game has created a much narrower tree structure that has almost no resemblence to the original game. However, it is important to remember that many of the games coded had very different gameplay and mechanics, and that they were broken in a variety of different ways, so the difference between a broken game and a playable one would not always be easy to detect. Further, each move from each game would have its own unique search tree; these might have very different shapes, even within one game.

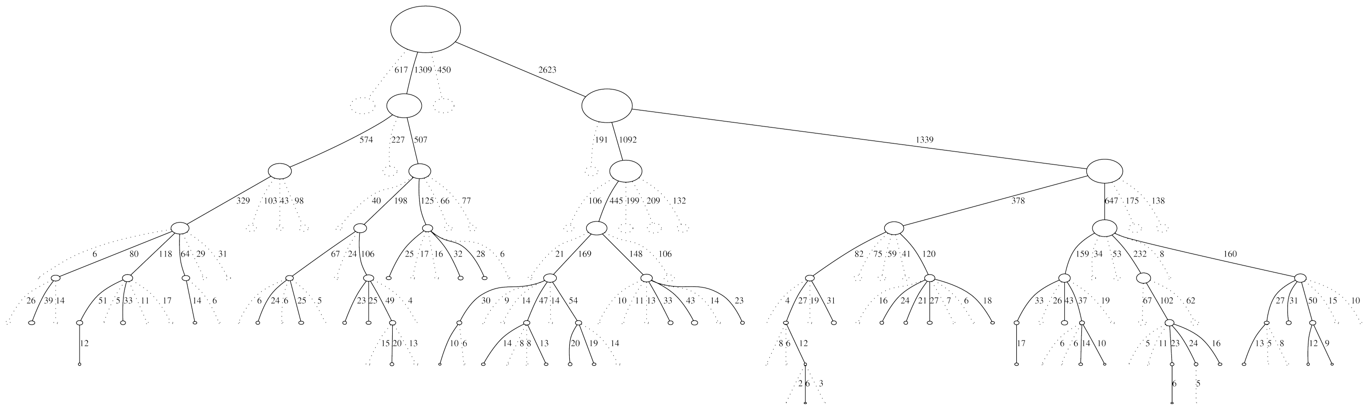


Figure 4.3: Example of a UCT search tree produced for the game of Othello.

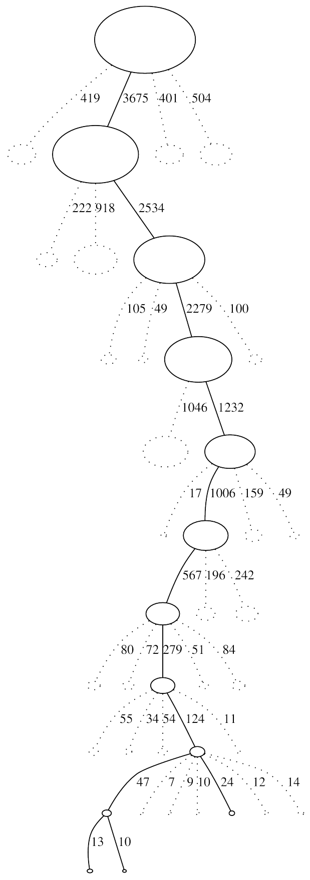


Figure 4.4: Example of a UCT search tree produced for the game of Othello when it was broken by only allowing players to play their stone next to their opponent's most recently laid stone.

4.2 Game quality prediction with neural networks

There was little evidence to support the initial hypothesis – despite testing a variety of network shapes using several different training times, no neural network was consistently able to predict the quality of games using UCT tree data. The results for the best-performing network are shown in table 4.1

		Predicted game quality					
		1	2	3	4	5	
	(Very poor) 1	0	0	0	0	0	0
	2	0.1	0.3	3.3	0.3	0	4
True game quality	3	0	0.7	5.7	0.6	0	7
	4	0	0	1.8	0.2	0	2
	(Very good) 5	0.1	0	3.8	0.1	0	4
		0	0.2	14.8	2	0	17

Table 4.1: The cross-validation matrix for the neural network setup which performed best at the game quality prediction task. Even this network has a low correlation coefficient and seems unable to identify very good games based on UCT data

4.3 Game playability prediction

The second hypothesis, regarding game playability prediction, produced more encouraging results. Decision trees, SVMs and neural networks were all tested for their predictive ability on this decision problem. Two separate datasets were tested – one where the attributes were created by simply aggregating data from all trees for each game, and another where trees for moves with player 1 to play were separated from trees for moves with player 2 to play before aggregation.

The predictive accuracy of each [UCT player, data aggregation, ML system] combination is shown in figures 4.5 and 4.6. As discussed in section 2.5.3, predictive accuracy alone is not enough to determine which machine learner will be the most preferable for the purpose of analysing automatically-generated board games. The recall of the ML system for both playable and unplayable games will influence how well it is able to identify a group of playable games from many potentials.

The precision and recall of each technique is shown in figures 4.7 to 4.10. They are compared to the expected results of a naive classifier[1]: this is a classifier that makes no inspection of attributes whatsoever, but simply predicts that a game is playable with probability $\frac{17}{28}$, since there are 17 playable games in the training set out of 28, and unplayable with probabilty $\frac{11}{28}$. A machine learning system must outperform this naive classifier to be considered to have performed well. The naive classifier is explained in more detail in section 5.1.1.

[1]not to be confused with a naive Bayes classifier, which is a more advanced predictive technique based on Bayesian probability theory

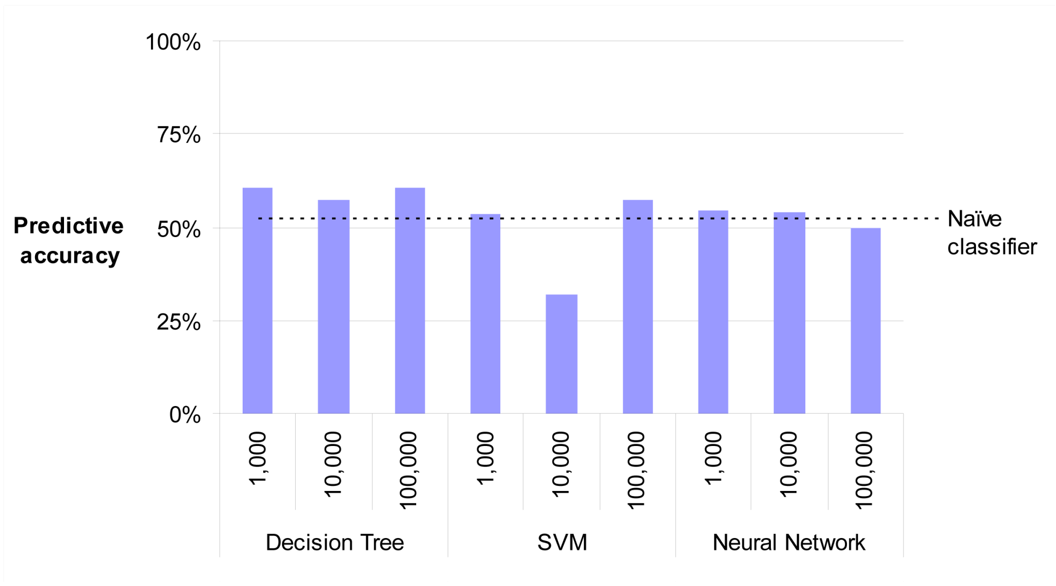


Figure 4.5: Predictive accuracy rates split by learning system and number of UCT iterations used to create the dataset. Predictive accuracy for the naive classifier, which simply predicts that every game is playable, is indicated by the dotted line.

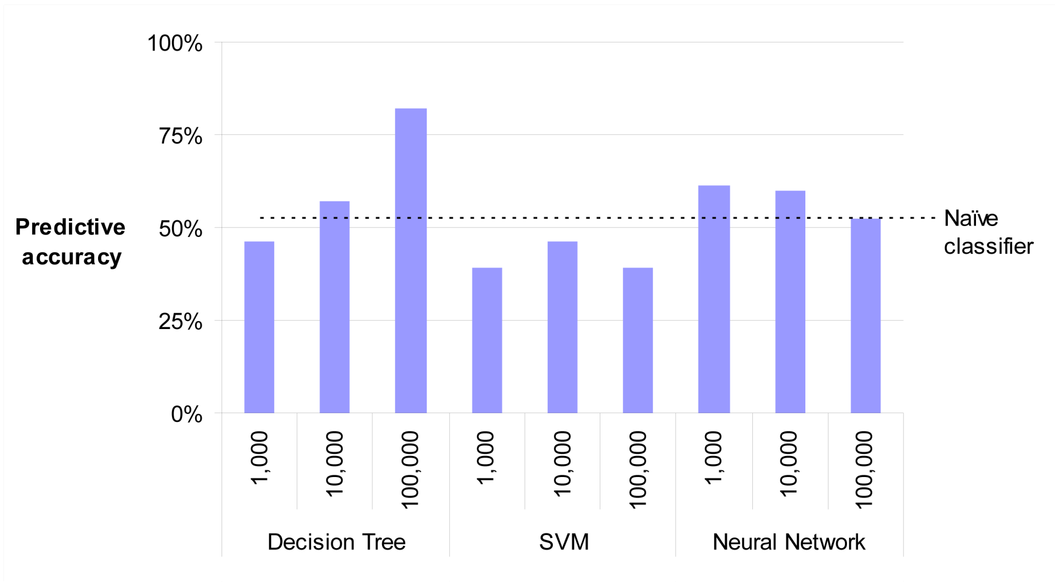


Figure 4.6: Predictive accuracy rates split by learning system and number of UCT iterations used to create the dataset (when the dataset was split by player number – trees for moves with player 1 to play were split from those for moves with player 2 to play). Predictive accuracy for the naive classifier, which simply predicts that every game is playable, is indicated by the dotted line.

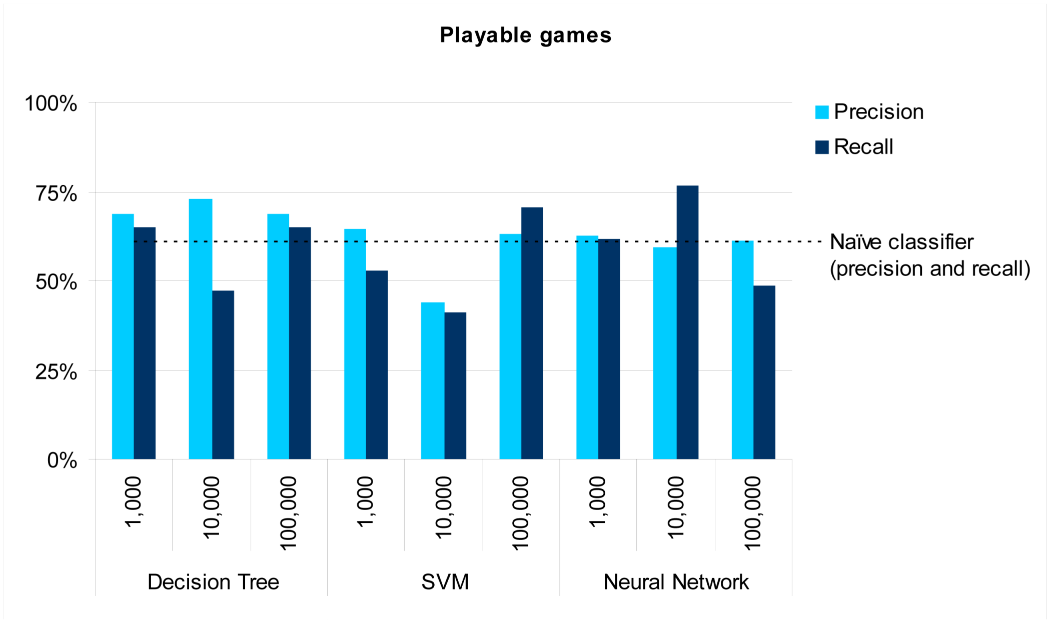


Figure 4.7: Precision and recall rates for playable games, split by learning system and number of UCT iterations used to create the dataset. The precision/recall rate for playable games for the naive classifier is indicated by the dotted line.

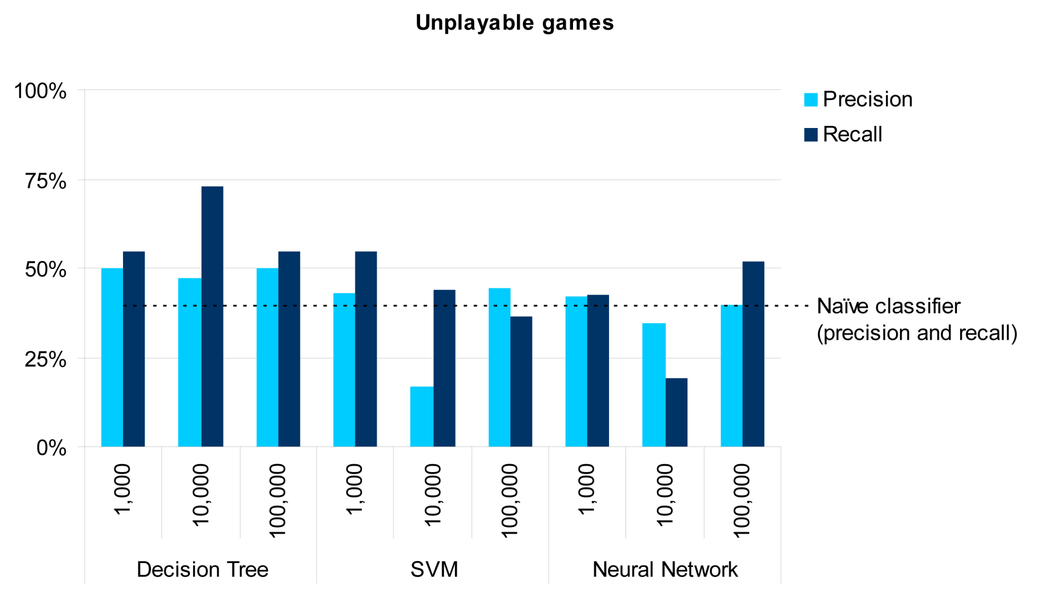


Figure 4.8: Precision and recall rates for unplayable games, split by learning system and number of UCT iterations used to create the dataset. The precision/recall rate for unplayable games for the naive classifier is indicated by the dotted line.

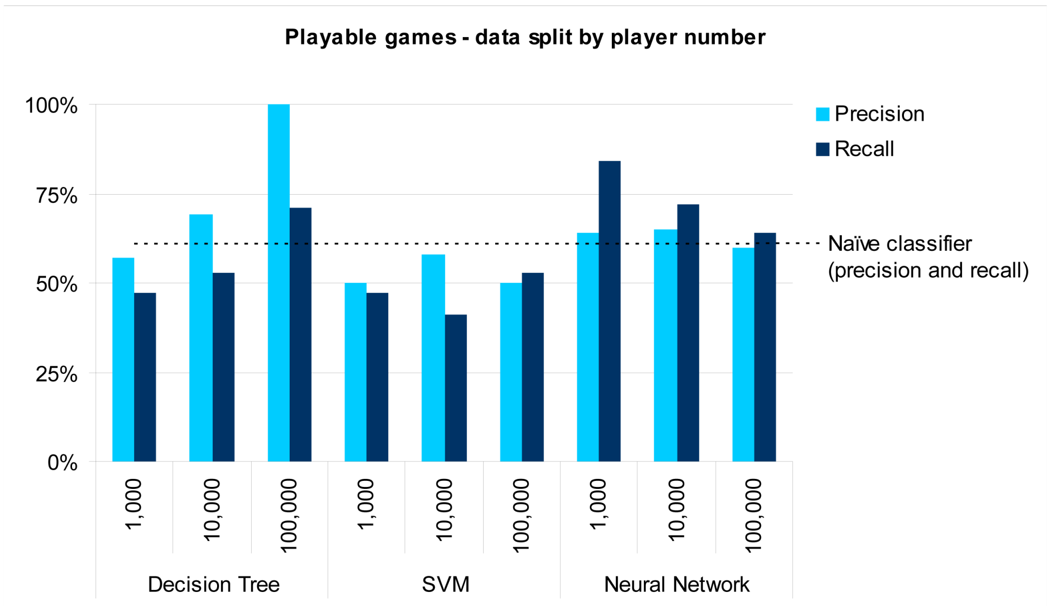


Figure 4.9: Precision and recall rates for playable games, split by learning system and number of UCT iterations used to create the dataset (when the dataset was split by player to number – trees for moves with player 1 to play were split from those for moves with player 2 to play). The precision/recall rate for playable games for the naive classifier is indicated by the dotted line.

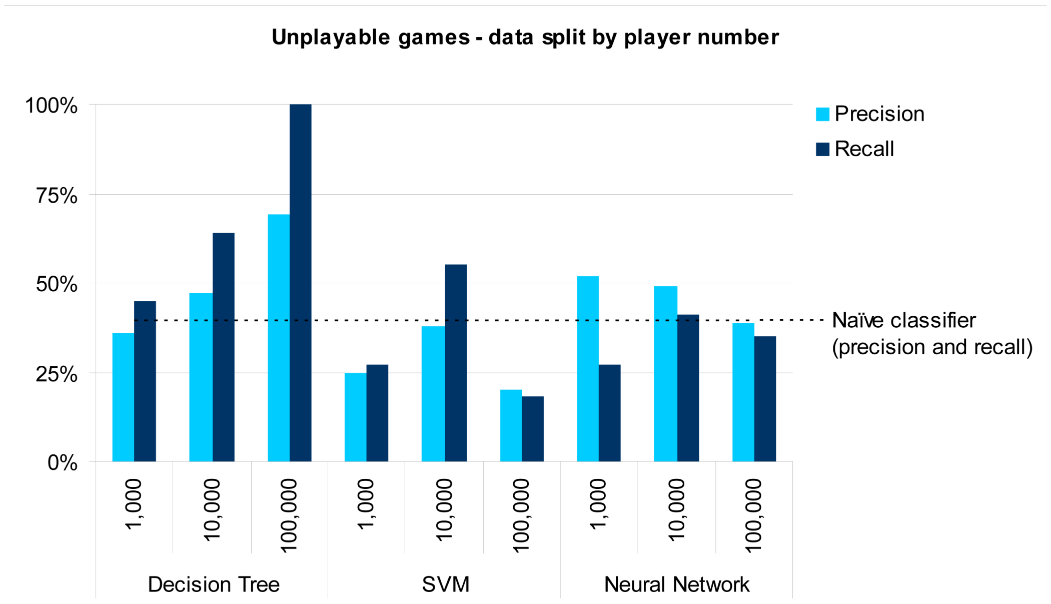


Figure 4.10: Precision and recall rates for unplayable games, split by learning system and number of UCT iterations used to create the dataset (when the dataset was split by player to number – trees for moves with player 1 to play were split from those for moves with player 2 to play). The precision/recall rate for unplayable games for the naive classifier is indicated by the dotted line.

Chapter 5

Evaluation

This chapter analyses and expands upon the results set out in chapter 5. It begins with some statistical anaylsis of the ML systems' predictive accuracy figures to assess their significance. It then moves onto analysing the UCT tree features used by the decision trees in making their game playability predictions, inspecting individual decision tree nodes in the light of the game quality concepts introduced in section 2.3. Section 5.3 covers the importance of recall for playable and unplayable games, and how they interact to determine how effective a machine learning system is when helping to analyse automatically-generated board games. Finally, possible reasons for some of the negative results are discussed.

5.1 Statistical significance of results

5.1.1 Game quality

Table 4.1 represents the best neural network shape and training time combination, measured using an average correlation coefficient over 10 repetitions of the cross-validation process. It seems quite intuitively clear that the machine learner has failed to learn the target function accurately, and indeed if the average correlation coefficient of 0.28 is in no way statistically significant. It has a p-value of 0.86, meaning that there is an 86% chance that a relationship as strong as this could have come about as a result of random chance.

This is a clear-cut negative result: the neural networks were unable to learn how to predict the quality of a game based on UCT tree data. Possible reasons for this failure are discussed later in the chapter.

5.1.2 Game playability

The machine learners' predictive ability with regard to game playability was encouraging, with one decision tree learner achieving predictive accuracy of 82%. It is important to establish the statistical significance of the results, though. Could these figures be the result of chance alone? To establish the success or otherwise of the machine learning systems, they shall be compared to a "naive classifier". For the purposes of this analysis, the naive classifier will be defined as follows: it simply calculates the proportion of playable and unplayable games in the known population, then classifies new games according to these proportions.

More specifically, since this research was conducted with 17 playable games and 11 unplayable ones, a naive classifier would predict that a new game is playable with probability $\frac{17}{28}$, and unplayable with probabilty $\frac{11}{28}$. The expected cross-validation results of this naive classifier are shown in table 5.1. It has expected recall rates of $\frac{17}{28}$ ($\frac{11}{28}$) for playable (unplayable) games, and identical expected precision rates.

Naive classifier

		Predicted		
		Unplayable	Playable	
True	Unplayable	4.3	6.7	11
	Playable	6.7	10.3	17
		17	17	28

	Unplayable	Playable
Precision	39%	61%
Recall	39%	61%

Table 5.1: Expected cross-validation results and precision/recall rates of the naive classifier.

In cases where the observed predictive accuracy of a machine learning system was higher than these naive figures, a t-test can be used to test for statistical significance.

Let P be the predictive accuracy rate of the machine learning system being examined, and P_n be the expected predictive accuracy rate of the naive classifier (i.e. $P_n = \frac{14.6}{28} = 52\%$, as shown in table 5.1).

Then the null (H_0) and alternative (H_1)hypotheses are:

H_0 : The ML technique is no more accurate than the naive classifier, $P = P_n$

H_1 : The ML technique is more accurate than the naive classifier, $P > P_n$

The test statistic is

$$t = \frac{(p - P)}{\sqrt{\frac{P(1-P)}{n}}}$$

where p is the observed predictive accuracy, n is the number of trials (i.e. 28) and under the null hypothesis, $P = P_n = 52\%$. The results of this significance test are shown in table 5.2.

There is a clear stand-out result here – the decision tree machine learner trained on 100,000 iteration UCT data (split by player number before aggregation) has a near-zero p-value. This is very strong evidence that it is has a higher predictive accuracy than the naive classifier. The next section explores the features used by this and the other decision trees to make their classifications.

5.2 Reasoning used by the decision trees

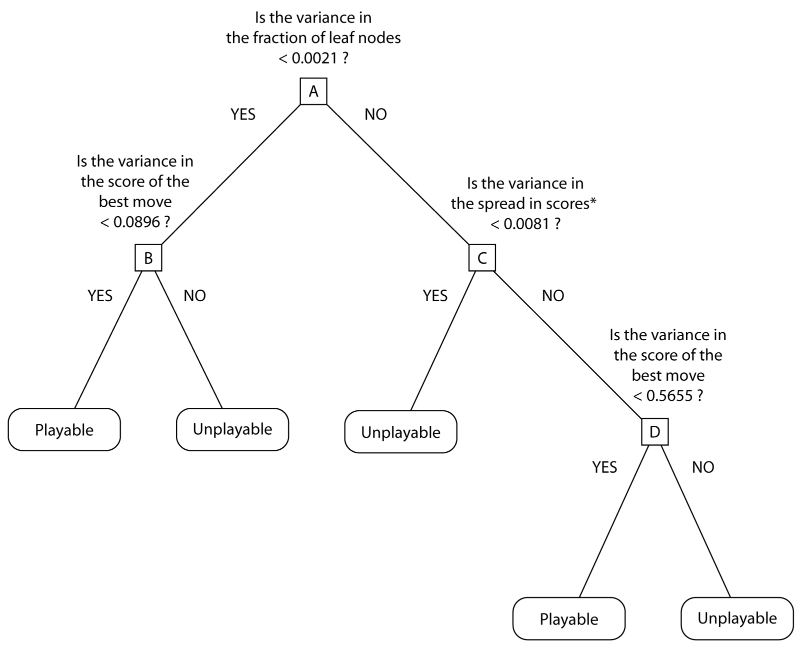
One of the advantages of using decision trees for this machine learning task is that they are simple to understand for humans, in contrast to methods such as neural networks. The features used by the decision trees in making their classifications can be easily identified. As described in chapter NN, sixteen features were used to describe the UCT trees in the course of this research. Are some of these features more closely related to game quality than others? If so, this might open new areas of research, and allow future work in this area to focus more closely on these most relevant features.

		Data not split before aggregation		Data split by player number before aggregation	
		Observed predictive accuracy	p-value	Observed predictive accuracy	p-value
Decision trees	1,000	61%	0.19	46%	-
	10,000	57%	0.31	57%	0.31
	100,000	61%	0.19	82%	0.00
SVMs	1,000	54%	0.45	39%	-
	10,000	32%	-	46%	-
	100,000	57%	0.31	39%	-
Neural networks	1,000	54%	0.42	61%	0.17
	10,000	54%	0.43	60%	0.21
	100,000	50%	-	53%	0.49
Naive classifier (expected)		52%		52%	

Table 5.2: Results of performing a t-test on the machine learners' predictive accuracy rates. With a p-value of nearly zero, there is very strong evidence that the decision tree learner trained on 100,000-iteration UCT data split by player number is a better predictor than the naive classifier.

So the next step in evaluating the results was to look at the tree features being used by these decision trees. How often various UCT tree features were used by the decision trees could be analysed, and efforts could be made to explain why some features were more relevant than others, and why particular decision nodes. This analysis would be done in reference to the game quality measures discussed in section 2.3 – do the decision tree classifications make sense in relation to these game quality concepts? This was especially true for the decision tree trained on 100,000-iteration UCT data split by player number, which showed significantly better predictive accuracy that would be expected of a naive classifier. It would be very interesting to see if some sense could be made of this classifier's strong predictive accuracy results.

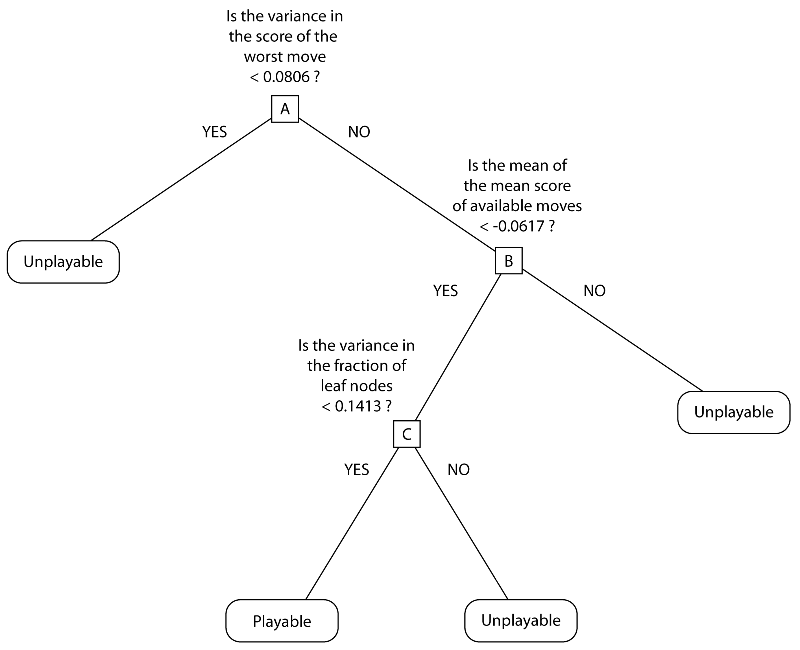
Shown below in figures 5.1 to 5.6 are the decision trees created for 1,000, 10,000 and 100,000 UCT iterations, with and without splitting by player number before aggregation. It should be noted that these are the overall decision trees learned when using all 28 example games as training data, rather than those learned during the cross-validation process.


Is the variance in the fraction of leaf nodes < 0.0021 ?
A
YES
NO
Is the variance in the score of the best move < 0.0896 ?
B
YES
NO
Is the variance in the spread in scores* < 0.0081 ?
C
YES
NO
Is the variance in the score of the best move < 0.5655 ?
D
YES
NO
Playable
Unplayable
Unplayable
Playable
Unplayable

Games classified to node	Unplayable	1	7	2	0	1
	Playable	3	0	0	13	0

* between the 1st and the $\frac{n}{3}$th move, when moves have been ordered by score (where n is the number of legal moves available)

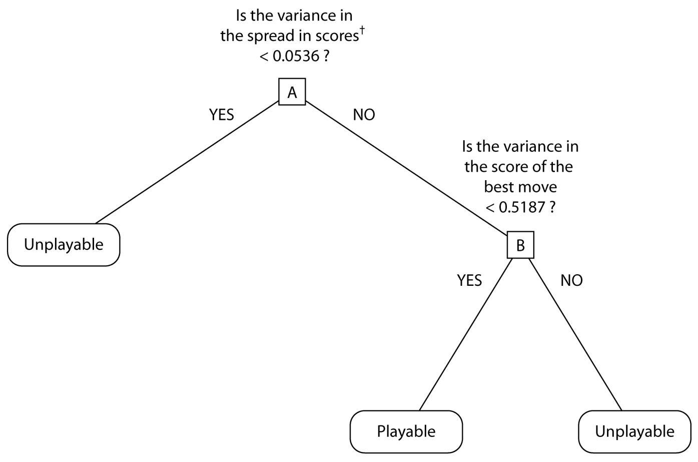
Figure 5.1: The decision tree learnt from the 1,000-iteration UCT data. The number of games classified to each node is shown in the table at the bottom of the figure.



Is the variance in the score of the worst move < 0.0806 ?
A
YES
NO
Unplayable
Is the mean of the mean score of available moves < -0.0617 ?
B
YES
NO
Unplayable
Is the variance in the fraction of leaf nodes < 0.1413 ?
C
YES
NO
Playable
Unplayable

Games classified to node					
	Unplayable	7	0	1	3
	Playable	1	15	0	1

Figure 5.2: The decision tree learnt from the 10,000-iteration UCT data. The number of games classified to each node is shown in the table at the bottom of the figure.



Is the variance in the spread in scores† < 0.0536 ?
A
YES
NO
Unplayable
Is the variance in the score of the best move < 0.5187 ?
B
YES
NO
Playable
Unplayable

Games classified to node				
	Unplayable	6	1	4
	Playable	0	16	1

† between the best and worst available moves

Figure 5.3: The decision tree learnt from the 100,000-iteration UCT data. The number of games classified to each node is shown in the table at the bottom of the figure.

Looking at the decision tree for 1,000-iteration UCT data in figure 5.1, node B differentiates between 7 unplayable games and 3 playable ones (although it does predict one unplayable game to be playable in doing so). It splits games according to the variance in the score of the best move. Those with larger variance are classified as unplayable while those with smaller variance are classified as playable. How can the decision being made at node B be explained?

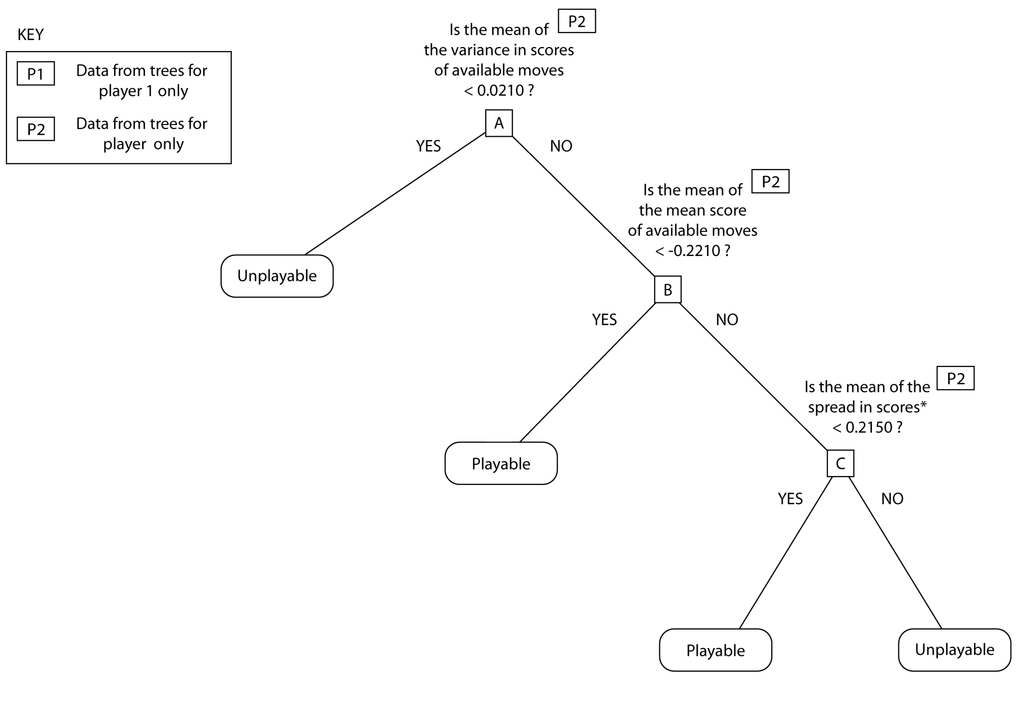
Imagine an unplayable game which is biased and quickly leads to a win for one player. Since the best score for the player with the advantage will be positive and increase to 1, while the best score for the opponent will be negative and decrease to -1, this results in a high variance when considering the UCT trees of both players. So this decision node seems to picks up on games which are too short and biased.

Node B also seems to pick up on several drawish games, though. Drawish games lack the quality of decisiveness, and so a player with a stronger position has difficulty winning the game. For this reason, drawish games may remain in a state where one player has the upper hand for a long time, without being able to end the game. During this prolonged period the best score for the player who is in a stronger/weaker position will be positive/negative. This results in a game which exhibits a high variance in best move score.

Node C also picks out two broken drawish games from a large group of playable games, because they have low variance in the spread in scores. This is typical of a drawish game, because most plausible moves – moves that are not obviously poor – will have scores close to zero throughout. Since this applies to both players, the variance in this statistic is very low for a drawish game.

The decision tree for 10,000-iteration UCT data in figure 5.2 immediately splits out 7 unplayable games using a different measure: the variance in the score of the worst available move (node A). This may again be linked to the fact that in a drawish game, both players will be in similar situations throughout. There will never come a time when one player is ahead (in which case his worst move might have a positive score) and one player will be behind (his worst score would be likely to be near -1). This lack of variation from one player to the other in terms of their game situation leads to a lack of variance in worst move score.

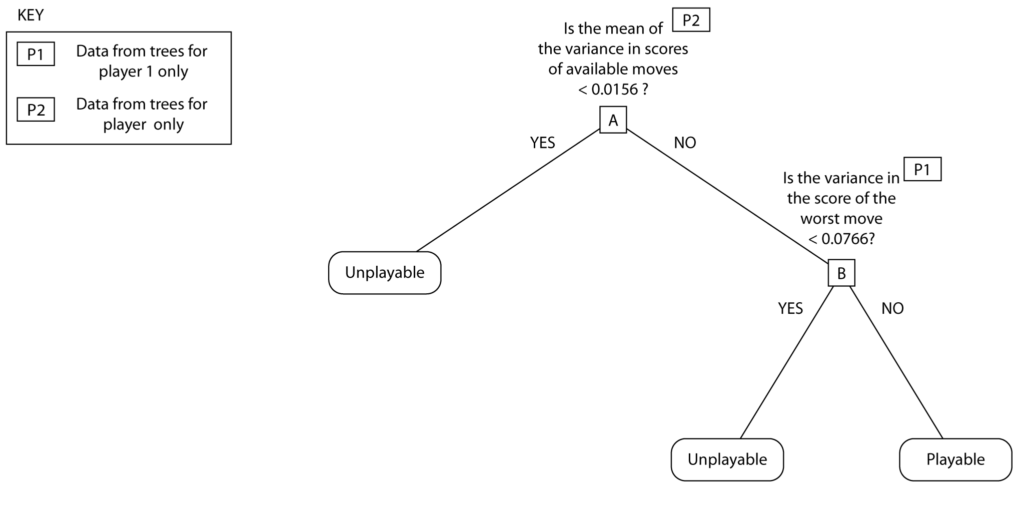
The decision tree for 100,000-iteration UCT data in figure 5.3 uses similar measures to those discussed above. Its two nodes focus on the variance in the spread in scores and the variance in score of the best available move. What of the decision trees that trained on data split between moves for player 1 and moves for player 2? These are shown below.


KEY
P1 Data from trees for player 1 only
P2 Data from trees for player only
Is the mean of the variance in scores of available moves < 0.0210 ? P2
A
YES
NO
Unplayable
Is the mean of the mean score of available moves < -0.2210 ? P2
B
YES
NO
Playable
Is the mean of the spread in scores* < 0.2150 ? P2
C
YES
NO
Playable
Unplayable

Games classified to node					
	Unplayable	6	0	2	3
	Playable	0	10	7	0

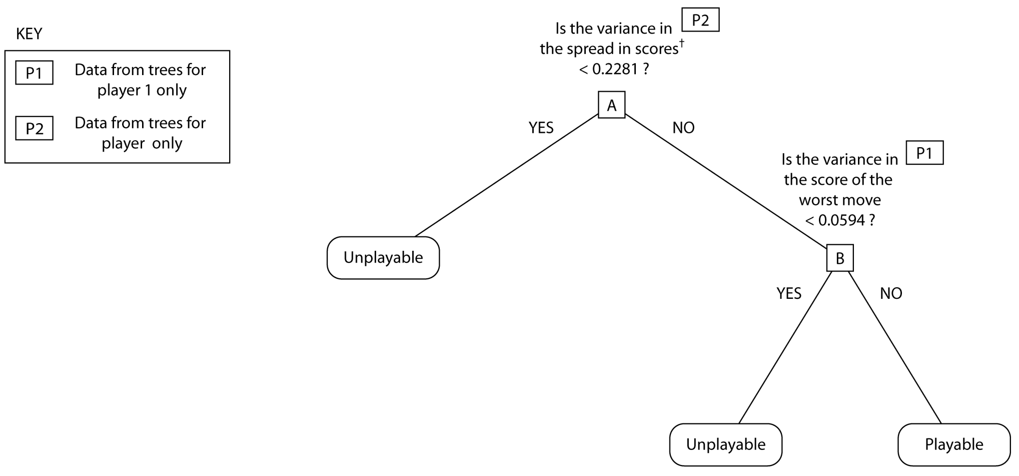
* between the 1st and the $\frac{n}{3}$th move, when moves have been ordered by score (where n is the number of legal moves available)

Figure 5.4: The decision tree learnt from the 1,000-iteration UCT data, split by player number before aggregation. The number of games classified to each node is shown in the table at the bottom of the figure.


KEY
P1 Data from trees for player 1 only
P2 Data from trees for player only
Is the mean of the variance in scores of available moves < 0.0156 ? P2
A
YES
NO
Unplayable
Is the variance in the score of the worst move < 0.0766? P1
B
YES
NO
Unplayable
Playable

Games classified to node				
	Unplayable	8	3	0
	Playable	1	2	14

Figure 5.5: The decision tree learnt from the 10,000-iteration UCT data, split by player number before aggregation. The number of games classified to each node is shown in the table at the bottom of the figure.


KEY
P1 Data from trees for player 1 only
P2 Data from trees for player only
Is the variance in the spread in scores† < 0.2281 ? P2
A
YES
NO
Unplayable
Is the variance in the score of the worst move < 0.0594 ? P1
B
YES
NO
Unplayable
Playable

Games classified to node				
	Unplayable	8	3	0
	Playable	1	1	15

† between the best and worst available moves

Figure 5.6: The decision tree learnt from the 100,000-iteration UCT data, split by player number before aggregation. The number of games classified to each node is shown in the table at the bottom of the figure.

The decision tree trained on 1,000-iteration UCT data in figure 5.4 begins with a node which inspects the mean of the variance in scores of available moves (node A). This picks out several drawish games, some because they are biased and short and some because they are long and drawish. This makes a lot of sense: in the short, biased games player 2's moves are likely to be either all good, or all bad; in the long, drawish games, players 2's moves are likely to all have scores close to 0. In both cases, there will be less variance than in a playable game where board positions are more balanced, but the potential to win or lose the game is ever-present. So the mean of the variance in scores for available moves will be low for these unplayable games. The very same measure is used at the top of the 10,000-iteration UCT data decision tree in figure 5.5.

The most successful decision tree, trained on 100,000-iteration UCT data split by player number uses similar measures to those already discussed. First it isolates 8 unplayable games using a lack in variance in the spread of scores of available moves for player 2. This lack in variance isolates games which are drawish (all scores often close to zero, hence low variance in spread in scores) and also games which are short and biased (scores bunched together close to 1 or -1). Node B separates the three remaining unplayable games using the variance in score of the worst available move for player 1. This node is picking out drawish games, because in a drawish game even the worst move available may not have a score far below zero. Especially with strong play from the 100,000-iteration UCT algorithm, which means that it avoids weak positions, the worst available move will tend to have a score which is only slightly negative. Compare this to a good game of high tension, where one slip up could mean a decisive disadvantage, meaning that the worst available move is likely to have a much more negative score associated with it.

In order to get more of an idea of which UCT tree features appeared most often in the decision trees used during cross-validation, each decision tree was inspected and the feature used at each node was logged. A chart summarising this data is shown in 5.7.

A handful of the statistics were used many times more than others, and as we have seen, very similar nodes appear in several of the decision trees. Investigating the

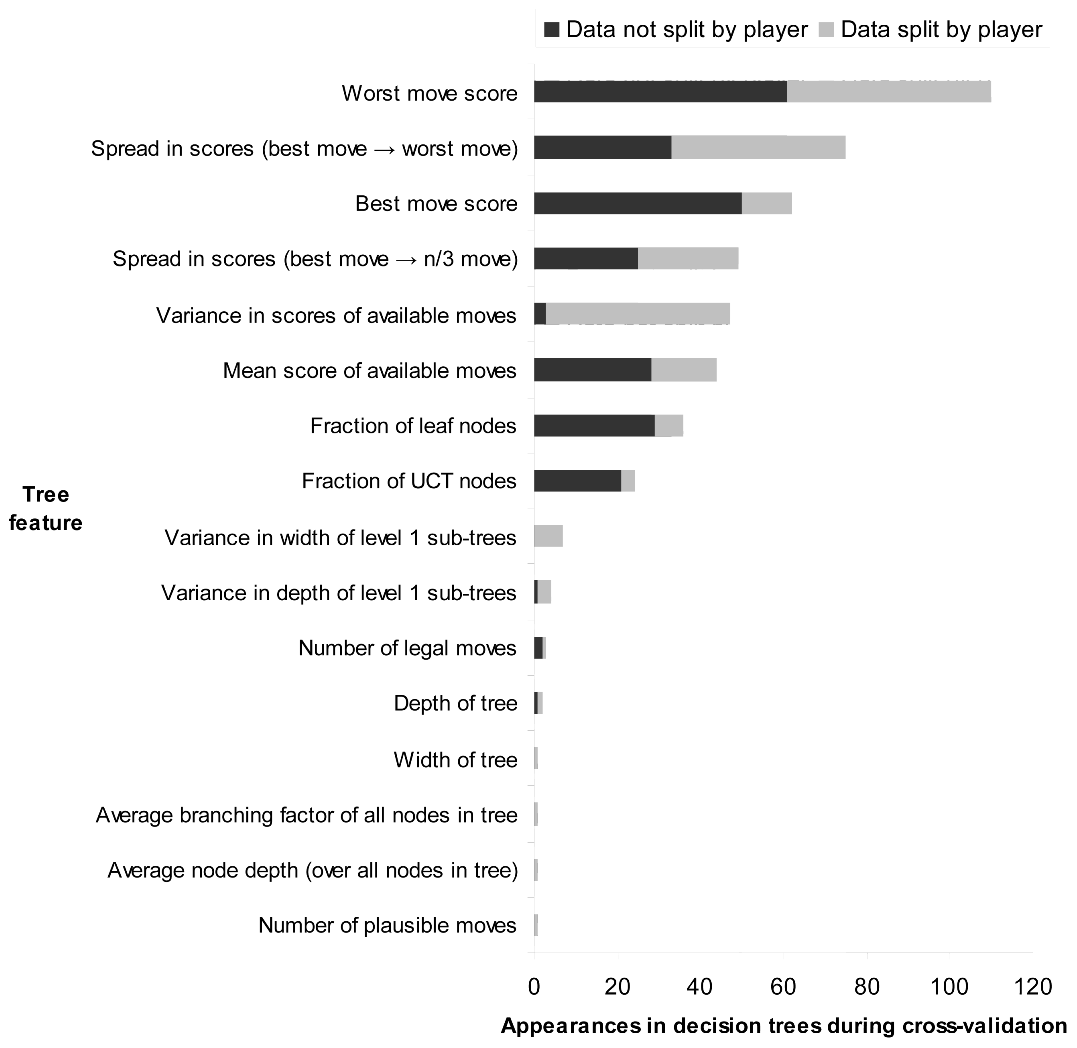


Figure 5.7: Chart showing how often each statistic describing the UCT trees was used in one of the decision trees during the cross-validation process. The data in the chart has been totalled over the cross-validation processes for 1,000-, 10,000- and 100,000-iteration UCT data.

importance and reliability of these decision nodes in classifying games represents an interesting area of future research; this is discussed in the next chapter.

The following section investigates the relationship between recall rates for playable and unplayable games of classifiers, and which is more important when using machine learning to identify playable automatically-generated board games.

5.3 Preferred method for predicting game playability

The headline finding of this research is that it has identified a machine learning system which performed significantly better than the naive classifier in identifying playable board games from UCT tree data: the decision tree learner trained on 100,000 iteration UCT data, split by player number. It has a high recall rate for playable games, and 100% recall for unplayable games, making it by far the preferred method of prediction for the purposes of classifying new games.

Part of the purpose of this research was to find a machine learner which would help identify interesting automatically-generated games. It is worth spending a moment calculating the mathematical advantage brought about by using this best-performing classifier upon new test cases. Doing so will also highlight how recall rates for playable and unplayable games interact to determine the predictive qualities of a classifier, and how the demographic of the test set from which the new cases are drawn can strongly influence whether one classifier outperforms another.

One important assumption will underly this analysis: automatically-generated games are likely, in general, to be unplayable[1]. A machine learner would be employed to vet a large set of games – to inspect them far quicker than a human would be able to, and to provide an initial estimation of their playability. Games classified as playable could then be inspected by human players for confimation.

As discussed in section 2.5.3, in completing this task – removing most of the bad games and picking out playable, potentially good games – it is desirable that no good games are "thrown out" as a result of mistakes made by the machine learner. A higher recall rate for playable games means that fewer potentially good games will be missed by the vetting process, so a high recall rate for playable games seems to be a good measure of a machine learner's suitability.

However, note that a naive classifier which classifies *every* game it encounters as playable will achieve a high recall rate – 100% in fact. This classifier would be completely useless, though, in helping to extract playable games from unplayable, since it would not remove any unplayable games from the test set. It would not reduce the amount of work to be done at the next stage in the process, where human players would check these potentially playable games. So it is also important that our ML system has good recall for *unplayable* games, so that it accurately removes these unplayable games from the test set.

In analysing which machine learning system is most appropriate, it is important to consider not only the proportion of good games identified correctly by the machine learner, but also its effectiveness in terms of the reduction in checking time it offers. This is dependent on both the recall (for playable and unplayable games) of the ML system and the proportion of genuinely playable games in the testing set. This proportion is an unknown, so for the purposes of analysis assumptions must be made about its true value.

If the proportion of genuinely playable games in the set of automatically-generated games is assumed to be 1%, then the recall and precision rates shown in the appendices can be used to calculate the expected number of playable games identified by each machine learning system. The results are shown in figure 5.8 and 5.9.

The best performing classifier is the 100,000-iteration decision tree with data split by player number. Even though its predictive accuracy is 82%, as compared with 52% for the naive classifier, its 100% recall for unplayable games means that it perfectly eliminates all unplayable games. This makes it a perfectly effective classifier (100% effectiveness). This is compared with effectiveness of below 2% for all the other ML techniques. Its above average recall for playable games means that it correctly identifies 7.1 playable games, which is a higher number than many of the other methods.

It is interesting to contrast the effectiveness of some of the other classifiers. Of the data not split by player number, the 10,000-iteration decision tree is the most effective. Despite having a lower predictive accuracy than the decision tree learners trained on 1,000- and 100,000-iteration UCT data, it is a more effective initial test in terms of reduction in checking time. Even though it only identifies 4.7 of the playable games on average, its high recall for unplayable games means that it eliminates many unplayable games correctly, giving it a high effectiveness.

[1]This is backed up by research by Browne, who synthesised 1,389 games, of which 19 were deemed playable[6]

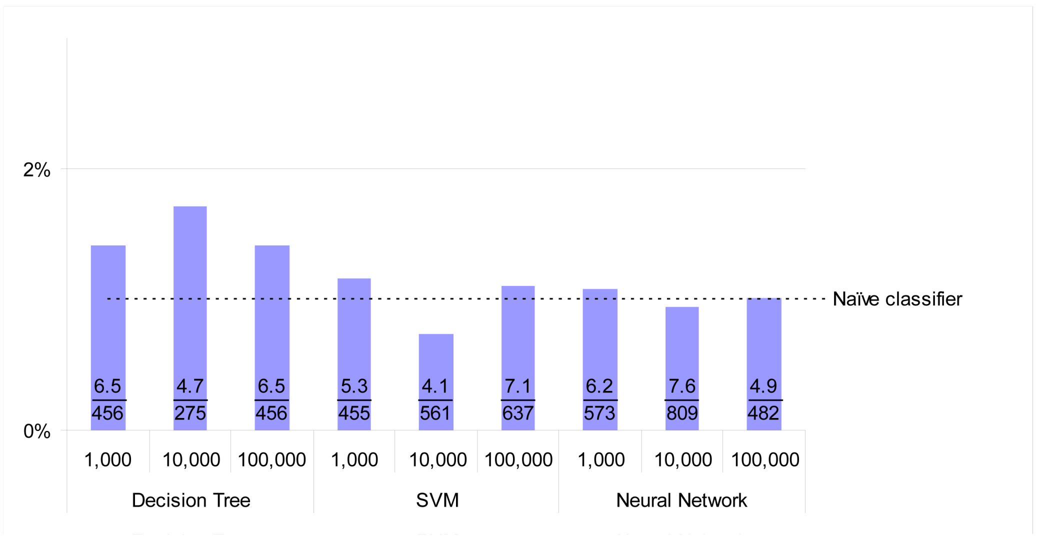


Figure 5.8: Chart showing the effectivenes of each machine learning method (expected proportion of predicted playable games that will actually be playable), assuming that a sample of 1000 automatically-generated games have been analysed, of which 1% are genuinely playable. The fraction used to calculate the effectiveness is shown as a datalabel – notice that some methods accurately identify several playable games (numerator), but classify many unplayable games as playable too, lowering their effectiveness.

By far the best classifier to use for this process is the 100,000-iteration decision tree with data split by player number, with a huge lead in effectiveness over the other methods by virtue of its perfect recall for unplayable games and good recall for playable games. Any classifier with less than 1% effectiveness is not worth considering, since it would be more effective to simply pick out games by using the naive classifier. Simply choosing $\frac{17}{28}$ of the games at random and checking these would identify playable games more quickly.

5.4 Areas for improvement

The decision tree machine learner trained on 100,000-iteration UCT data split by player number showed very strong predictive accuracy results. Statistical analysis showed very strong evidence that this technique is a better predictor of game playability than the naive classifier. This is a welcome result. However, it should be noted that none of the other techniques tested were better than the naive classifier, from a statistical standpoint. Of the p-values in table 5.2, only one is below the 10% mark. So there were several negative results for this decision problem. Further, the performance of the neural networks in predicting game quality was very poor.

What factors have contributed to these negative results, and can they be addressed to achieve better results in the future? This question is the focus of the next two sections. Potential approaches to addressing these issues are discussed in chapter 6.

5.4.1 Sample sizes

From a purely statistical standpoint, it would be valuable to confirm the findings of this research with larger sample sizes. When tackling the game quality prediction problem, a sample of 17 games were presented to the neural networks. These games formed a good cross section of abstract strategy board games, and included a variety of gameplay

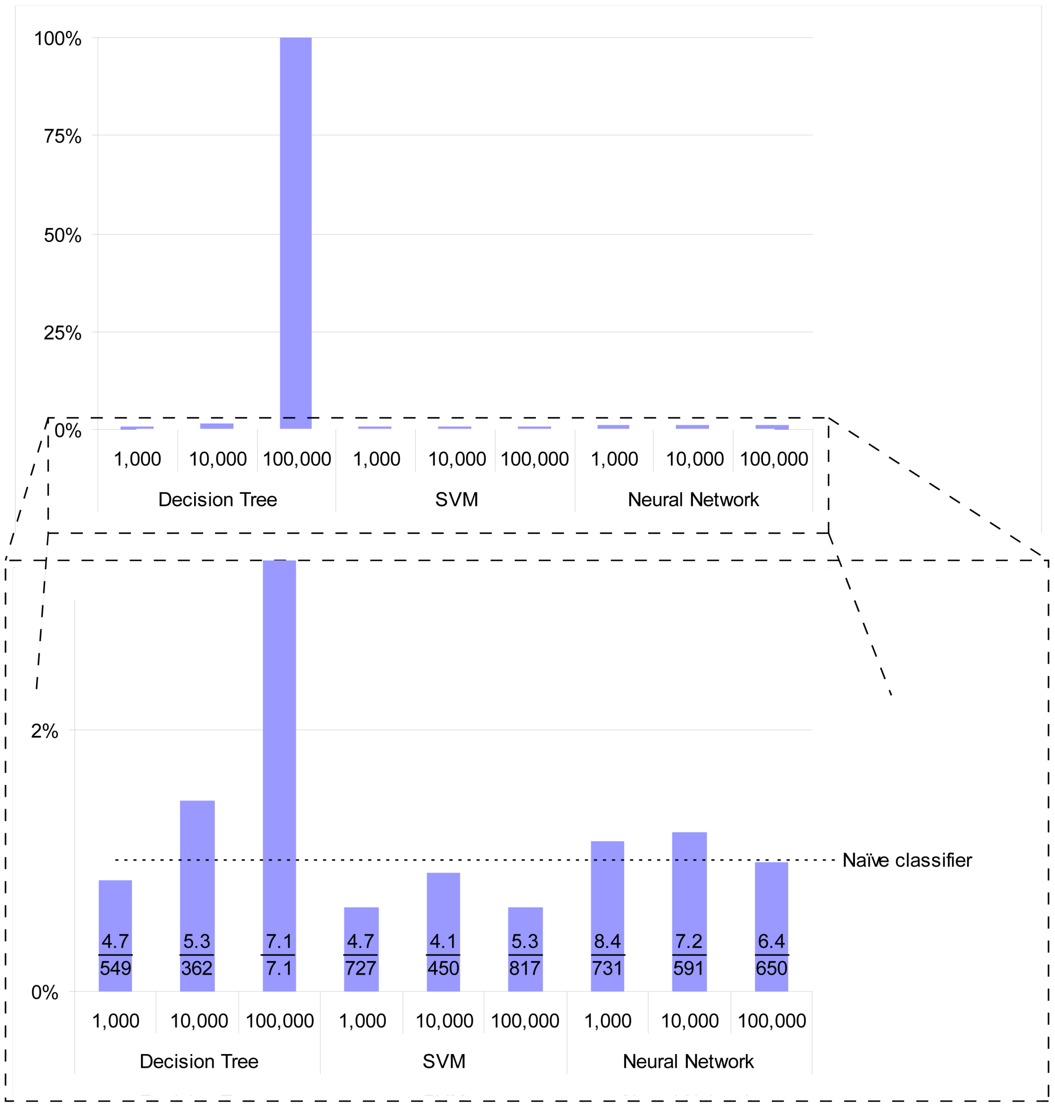


Figure 5.9: Chart showing the effectiveness of each machine learning method (expected proportion of predicted playable games that will actually be playable) when data is split by player number before aggregation. The 100% recall for unplayable games of the 100,000-iteration decision tree makes it a superbly effective classifier, because it accurately eliminates all unplayable games.

models and mechanics. However, for the purposes of statistical analysis and machine learning, 17 games is a very small sample. Programming a larger number of games would have given the neural networks a greater chance of finding a relationship.

For the game playability prediction problem, the group of 17 playable games was extended by adding 11 unplayable games. This gave a slightly larger sample size, but even 28 games still represents a small dataset for a machine learner. All machine learners benefit from the availabilty of more training data; it is highly likely that a larger sample of games would have improved the performance of all the techniques on test. Ideally, around 50 playable and 50 unplayable games would represent a good number. Unfortunately, given the time constraints, this was simply not possible in the course of this research.

5.4.2 Failings of UCT

Another major factor in the lack of positive machine learning results may stem from the UCT algorithm itself. UCT does adopt a rather "human" approach when applying its selection algorithm to sample possible moves. It is very strong in games such as Connect4, where its search tree extends many moves ahead into the possible game continuations. It plays these games in what seems to be a very intelligent manner, and in a style not easily discernable from a strong human player.

However, UCT is not a human player, it is an algorithm, and there are situations where this algorithm falls down. For games with large branching factors, the UCT tree boundary may lie no more than 2 moves ahead, even when 10,000 playouts are used. Compare the UCT search tree for Connect4 (figure 5.10) with that of Breakthrough (figure 5.11). When UCT's search tree is as shallow as that shown in figure 5.11, it is not seeing deeply into the current game position, and it cannot make particularly intelligent moves. If the algorithm is not playing the game intelligently, how can we hope for its tree shape to give any indication of a game's quality or playability?

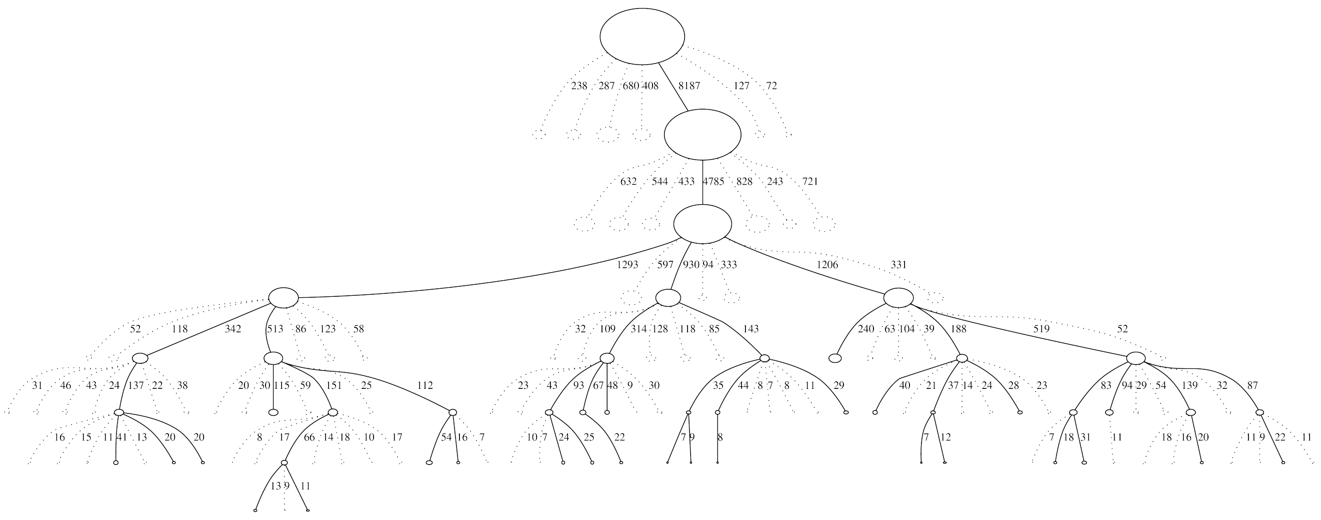


Figure 5.10: An example of a search tree for the game of Connect4 – which has a branching factor of around 7 – for a 10,000-iteration UCT player. Only nodes which have been visited often enough for the node selection algorithm (expression **??**) to be used are shown. Moves currently considered weak are shown with dotted lines, and sub-trees of these weak moves are not shown.

Indeed, in some games even a UCT player with several thousand iterations is a very weak opponent for a human player. Game positions with very obvious conclusions such as the Dots & Boxes situation shown in figure 5.12 can present UCT with difficulties.

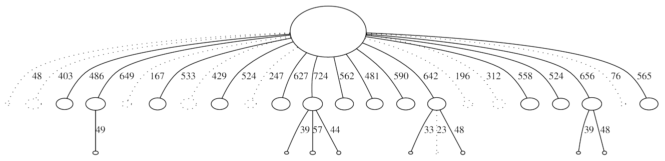
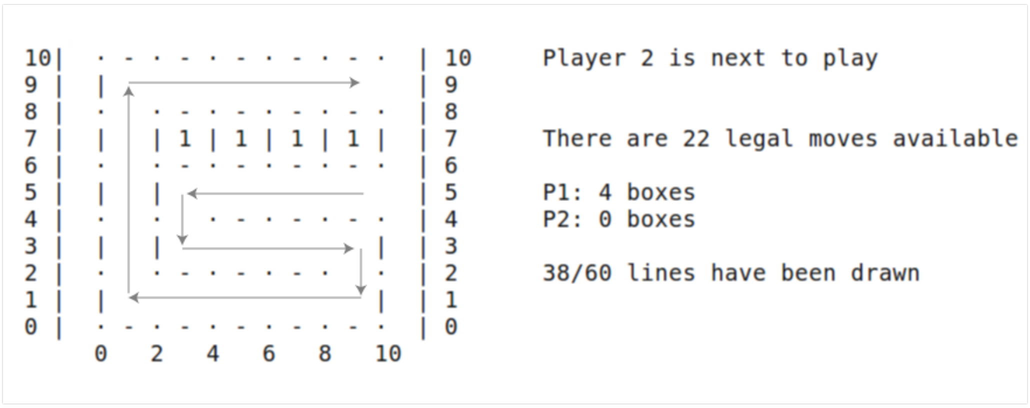


Figure 5.11: An example of a search tree for the game of Breakthrough – which has a branching factor of around 20 – for a 10,000-iteration UCT player. The high branching factor means that UCT cannot see deeply into the game situation.

The game is played as follows: players take turns to draw lines between neighbouring dots on an orthogonal grid. When a square box (of side one unit) is completed by drawing a line, that box is assigned to the player who created it; and the winner is the player with the most boxes at the end of the game. Crucially, when a player completes a box, they are allowed to play again; this creates the possibility of chain moves which capture several boxes in one turn.



```
10|  · - · - · - · - · - ·  | 10     Player 2 is next to play
9 |  |                      | 9
8 |  ·    · - · - · - · - · | 8
7 |  |    | 1 | 1 | 1 | 1 | | 7      There are 22 legal moves available
6 |  ·    · - · - · - · - · | 6
5 |  |    |                 | 5      P1: 4 boxes
4 |  ·    ·    · - · - · - · | 4      P2: 0 boxes
3 |  |    |              |  | 3
2 |  ·    · - · - · - ·    · | 2      38/60 lines have been drawn
1 |  |                   |  | 1
0 |  · - · - · - · - · - ·  | 0
     0    2    4    6    8   10
```

Figure 5.12: An example of an intuitively simple position where UCT fails to identify the correct course of action. Any human player with basic knowledge of the game understands that player 2 must fill in lines along the corridor that exists on the game board to win (indicated by grey arrows). UCT rarely finds this continuation (see figure 5.15).

In the position shown, player 2 has a clear win by filling in lines along the one remaining winding "corridor", as indicated by the white arrows. Each line drawn will create a new box and allow player 2 to continue in one long chain move until the game is won.

However, even a simple win for player 2 in this position is not at all obvious to UCT. The reason UCT has difficulty in deciphering this sort of position is that it is considering so many possible combinations of moves. Since there are 22 moves available at level 1 of the search tree, 21 moves available at level 2 of the search tree, and so on, the number of possible continuations grows factorially. UCT cannot hope to evaluate all the possible continuations, and is likely to make the wrong move in this intuitively simple position.

How likely is it to make an incorrect move? This was investigated by creating a simplified version of the Dots & Boxes game, played on a straight corridor-shaped board which mimics the game situation shown in figure 5.12. The board used is shown in figure 5.13. The probability that a variety of different-strentgh UCT players play the correct move for each board size is shown in figure 5.14. The probability that each of these UCT players performs a complete correct playout for each board size is charted in figure 5.15.

```
2 |  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .  | 2
1 |  |                                                                   | 1
0 |  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .  | 0
     0   2   4   6   8   10  12  14  16  18  20  22  24  26  28  30  32
```

Figure 5.13: The board used for testing UCT's ability in the Dots & Boxes game. Note that the simple corridor arrangement shown here is identical to the game situation shown in figure 5.12. The winning strategy is intuitively obvious to a human player, but difficult for UCT to find due to the huge number of potential continuations.

Even a very strong UCT algorithm often fails for boards of length greater than 20. If these corridor situations occur not at the root node of the search tree (as was the case in this test) but further down, UCT wil have far fewer than its maximum number of iterations remaining with which to perform analysis. Having fewer iterations available increases the chance of a mistake.

Although a 100,000-iteration UCT algorithm has about a 50% chance of playing the correct move on a board of length 24, note that to correctly win all the boxes in succession, the single correct move must be played 24 times consecutively. Playing the correct first move on a board of length 24 leaves a board of length 23; then playing the correct first move leaves a board of length 22, and so on. So the probability of UCT playing out this position correctly is the product of the probability that it plays the correct first move on each board size.

So UCT often fails to calculate how to play out these corridor situations correctly. At a "higher" level of play, it will therefore fail to avoid offering corridor situations to its opponent, and fail to force the game into continuations where it benefits from them. However, the previous sentence perfectly sums up the innate challenge and interest of the Dots & Boxes game. These corridor situations always crop up in some way throughout the game – it is the challenge of predicting them, calculating their effects on the score and if necessary, altering the gameplay to avoid them that makes the game fun for human players.

If UCT is failing to pick up on the key gameplay dynamic that defines Dots & Boxes, how can its search trees be a fair reflection of the game that human players enjoy? If these search trees are not a fair reflection of the game, there seems to be no chance that a machine learner using only tree data can accurately predict game quality or playability. This issue may well have influenced some of the negative machine learning results.

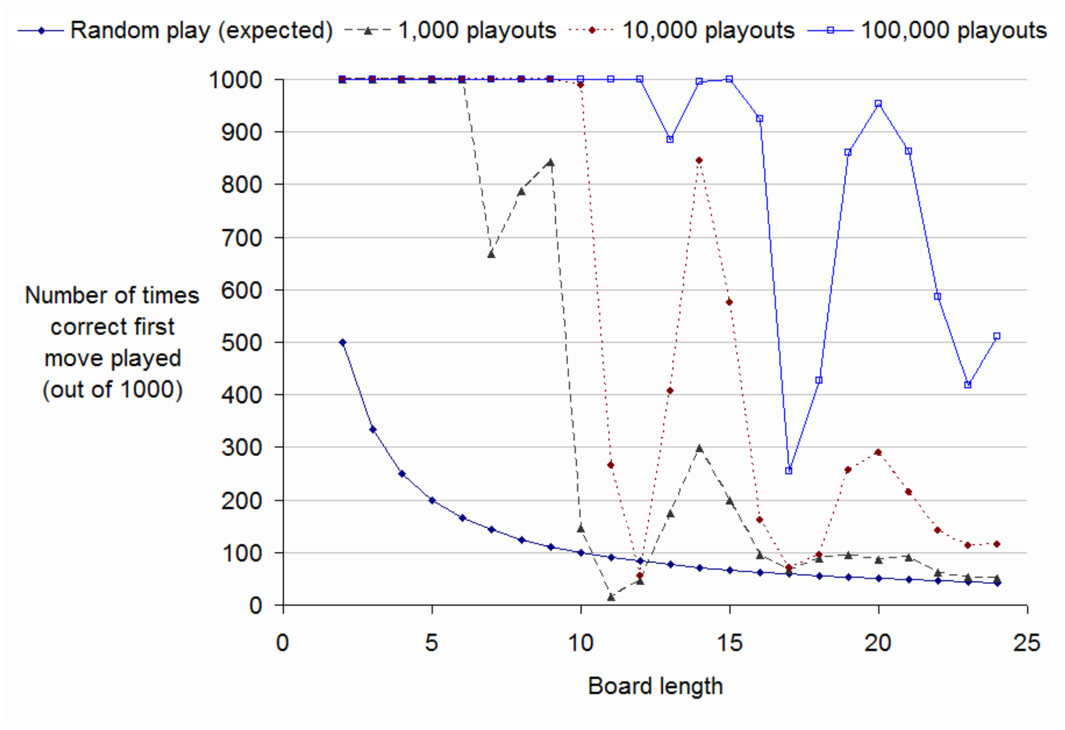


Figure 5.14: Chart showing the probability that the UCT algorithm plays the correct move when faced with the simplified version of the Dots & Boxes game shown in figure 5.13. For boards of length 20 or more, the high branching factor means that even a 100,000-iteration UCT algorithm does not find the right move every time.

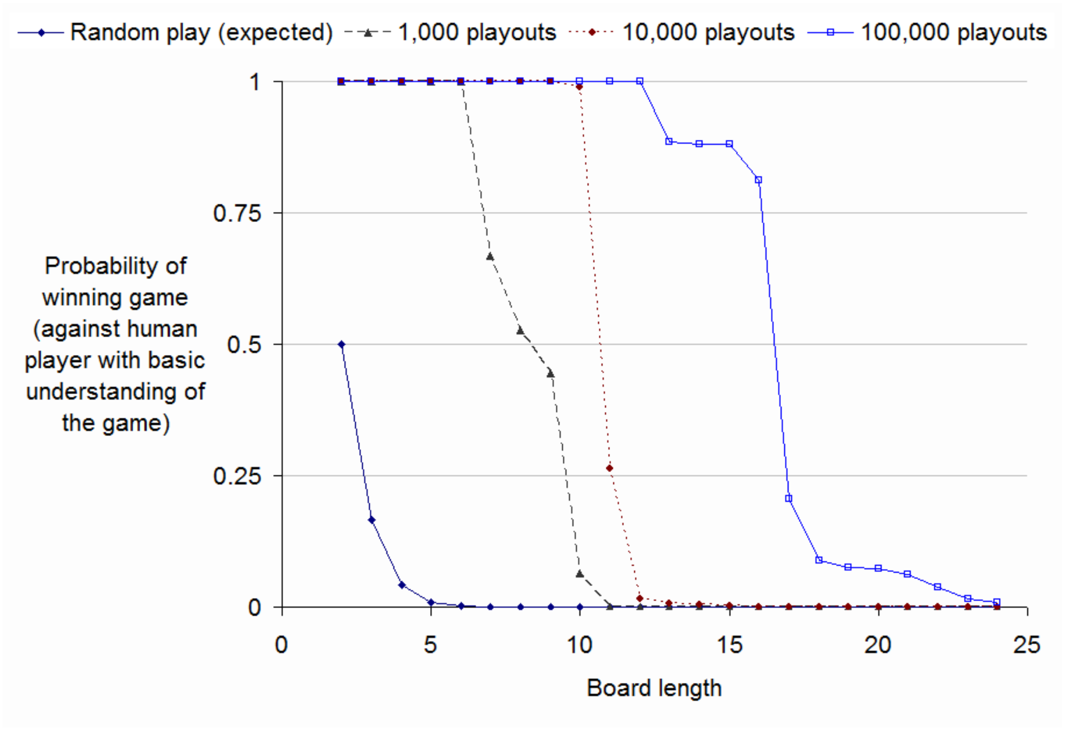


Figure 5.15: Chart showing the probability that the UCT captures all remaining boxes in the testing position shown in figure 5.13. For long boards, the cumulative effect of having to make several consecutive correct moves means that even a 100,000-iteration UCT algorithm is very unlikely to play out this position perfectly.

Chapter 6

Further work

This research has found one machine learning system which was able to identify playable games better than the naive classifier. However, other techniques proved poor. There was no machine learnable link between tree shape and game quality, and even predicting game playability based on tree shape proved difficult. Nevertheless, if the major issues discussed in the evaluation section can be addressed, a link between UCT tree shape and game quality may yet be proved.

6.1 Confirmation of results

The sample size remains an issue with this research. In general, machine learning systems require large datasets to effectively learn a target function. If a larger sample of games of various qualities were to be coded up, the extra data be enough to allow neural networks to find a reliable link between UCT tree shape and game quality. Just as a positive result with a small sample size cannot be considered definitive, the negative results described in this paper do not put an end to this line of research.

A larger sample size may also improve the machine learning systems' ability to predict game playability. Although this research has identified only one [data collection/aggregation, machine learning system] combination that is better than the naive classifier, other techniques and data collection methods may prove viable if larger training sets of games are used. It would be valuable to confirm the results of this paper and to test the performance of other machine learning systems (such as Inductive Logic Programming, Bayesian classifiers and Case-Based Reasoning) with more data.

6.2 Understanding the most used UCT tree features

Figure 5.7 shows the UCT tree features that were most commonly used by the decision tree learners in this research. This chart would seem to imply that some tree features are much more closely related to game quality/playability than others. Again, it would be useful to confirm these findings using a larger sample size.

If UCT trees really can be used to predict game quality or playability, another interesting area of research would be to analyse which tree features are most closely related to game quality, and why these tree features are so revealing. Section 5.2 puts forward some ideas in this respect, but a more focused analysis may bring about advances in understanding. If definitive relationships between certain tree features and game quality can be established, this would be very useful. Automatic generation of board games would become a great deal easier if these features could be tracked during the generation process.

It would also be of interest to reevaluate various machine learning systems when they presented with attributes based on only relevant UCT tree features. The chart in figure 5.7 shows which UCT tree features were most commonly used by the decision tree machine learners. This chart suggests that features such as the number of plausible moves are not relevant to game playability. If this is really the case, techniques such as neural networks may perform far better if trained on only a select set of relevant attributes (based on those features at the top of the chart), since there will not be as much "noise" from irrelevant features to contend with.

6.3 A stronger, more sensible UCT algorithm

As mentioned in chapter 5, UCT sometimes fails to play games in the way it should – the algorithm still falls down in certain situations. At times these are situations in which the right course of action would be very clear to even an inexperienced human player. Addressing this issue (creating a more intelligent version of the algorithm that can play these types of positions well) is likely to enhance any relationship between tree shape and game playability. If the algorithm could play these positions in a more sensible, human manner, its search trees would surely be more closely related to the game's true nature, and hence its playability.

There are three contributing factors to UCT playing these positions poorly.

- Problems coping with large branching factors

  As discussed in the previous chapter, part of the issue when UCT plays positions poorly is to do with large branching factors. Positions with large branching factors can result in a huge number of possible continuations, and although UCT can run very quickly, it cannot possibly analyse all possibilities accurately. In situations like this UCT creates very wide, shallow search trees, meaning that it does not see very far ahead into the position.

- Ignorance of equivalences and symmetries

  UCT ignores considers the same board position brought about by different move orders to be entirely separate. This means that it will analyse two or more identical board positions separately if they lie at separate nodes in its search tree (see figure 6.1). This means that the UCT tree will contain several nodes with low visit counts for the same board position, each of which may have a poor value estimate as a result.

  UCT also has no concept of symmetries in game positions. For example, there are only really three possible game situations after 1 ply in Noughts & Crosses: player 1 may move in the centre, or in the middle of an edge, or in a corner. However, UCT considers there to be 9 possible game situations, because it has no concept of the fact that any move in a corner is equivalent to any other for the by symmetry. This compounds the branching factor problem mentioned above.

- Lack of intuition

  UCT has no intuition, and no concept of the simple strategies and rules of thumb that humans can call upon when they learn to play a game. A position such as that shown in figure 5.12 can easily be won by a human player by simply "filling in boxes along the remaining corridor". UCT will never have an understanding of concepts like this. Although finding the win comes easily to a human player, there is only one winning combination out of several billion possible game continuations. The ease with which this winning combination occurs to humans comes from our ability to pattern match and inductively solve large problems from small ones.

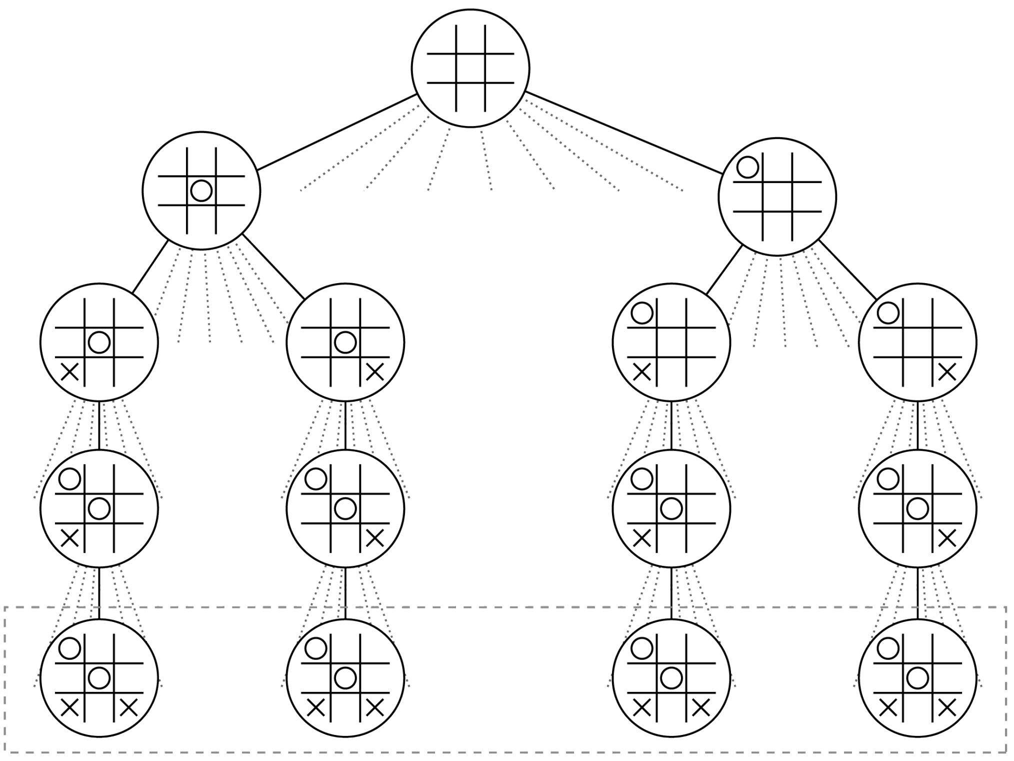


Figure 6.1: An example of UCT's ignorance of symmetry. The Noughts & Crosses board position indicated by the grey dotted lines occupies several nodes in the UCT search tree. Playouts from one of these nodes will never be reflected in the scores for the other "twin" nodes.

So UCT struggles in board positions with large branching factors, fails to identify symmetries in positions, and is unaware of basic strategies and heuristics that human players employ when playing games. The latter is a fundamental part of the way the algorithm works: UCT is a completely general game-playing algorithm that does not use any heuristics or prior knowledge. However, the former two issues might be addressed without stepping outside the bounds of UCT's Monte Carlo Tree Search roots. A possible area of further work presents itself which may address these issues and result in a stronger, more sensible UCT player.

UCT currently considers identical board positions reached through different move orders as entirely different. These board positions would be located at different nodes in its search tree, and have separate visit counts and scores (depending on the precise simulations performed by UCT during its iterations). See figure 6.1 for an example of this. Despite their identical board positions, these "twin" nodes are entirely separate; playouts made through one of these nodes have no influence whatsoever upon the others.

However, there is a strong case for updating *all* of these twin nodes with the score from any playout game through one of them. After all, if this playout gives information about the game state for one twin node – enhancing UCT's understanding of its strength or weakness – then the information is surely relevant to all twin nodes, by virtue of their identical board positions?

If this change to UCT were implemented, there are two possible approaches to propagating playout game scores back up the search tree. One is to update not only the twin nodes, but to update all paths leading back to the root node, as shown in figure 6.2. The other is to update all nodes on the original path of descent and their twins, without tracing multiple paths, as shown in figure 6.3.

The former approach seems valid, since all nodes on the paths being updated *could* have led to the playout game. However, this updating process could prove complicated when multiple paths converge, as shown at the top of figure 6.2. The latter "single return path" approach seems more elegant, and still ensures that the information from the random playout is propagated appropriately to twin nodes more fully than in classic UCT.

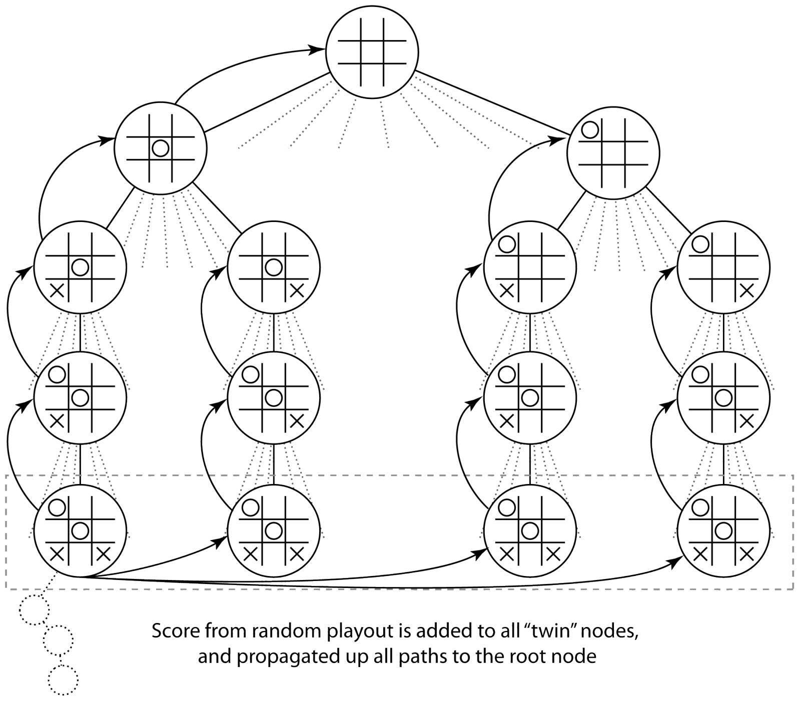

Score from random playout is added to all "twin" nodes,
and propagated up all paths to the root node

Figure 6.2: A suggested alternative approach to positions with identical board positions. Twin nodes are identified each time a new node is added to the tree, and pointers are used to maintain linked lists of twin nodes. When a playout is performed, the playout game score is used to update not only nodes on the original path from the root node, but all nodes on paths back from twin nodes to the root node.

This approach may also help to reduce branching factor. Figures 6.2 and 6.3 show nodes on different paths within the search tree propagating information to their "twin" nodes. This would be possible by maintaining pointers from a node to each of its twins[1].

Yet note that all twins will have exactly the same score and visit counts under this new setup, and, by definition, identical game states. Thus these nodes are simply copies of exactly the same information; they differ only in their parent node. This means that there is no need to keep them separate: they can be represented by just one node. The only proviso is that this node will now potentially have more than one parent. The topology of the search space can now be thought of not just as a tree, but as a web (see figure 6.4).

This setup simultaneously reduces branching factor and improves value estmation. Imagine that each of the nodes with identical board positions in figure 6.1 has been visited 15 times. Instead of having four separate nodes, each with 15 visits, the proposed "search web" shown in figure 6.4 would have just one node, with 60 visits, and hence a much higher understanding of that state's true value. This better understanding would result in more accurate move choices at each of the parent nodes during future iterations, improving the accuracy of their value estimations.

[1]or twin nodes could be kept in linked lists, requiring only one "next twin" pointer per node

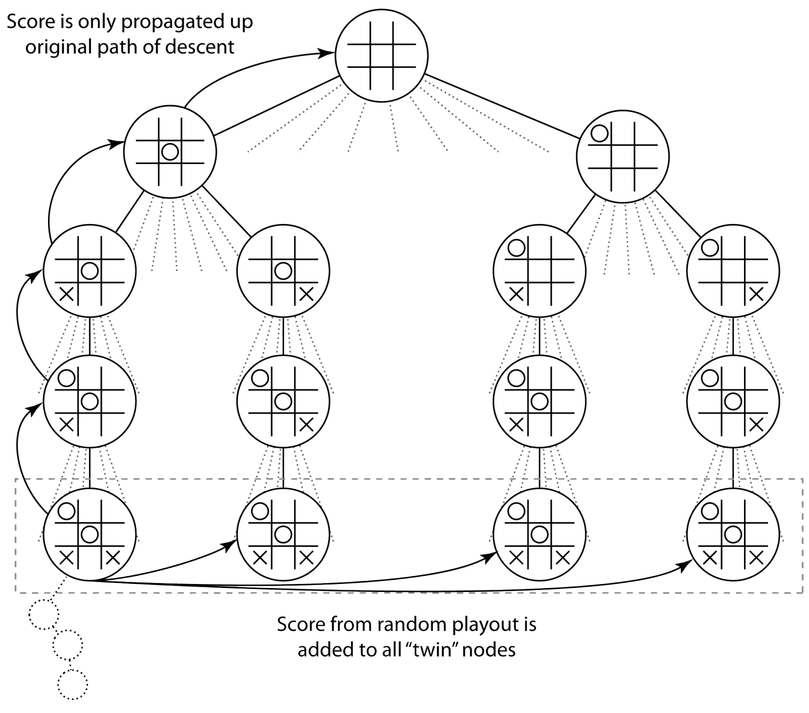

Score is only propagated up original path of descent
Score from random playout is added to all "twin" nodes

Figure 6.3: Another alternative approach to positions with identical board positions. When a playout is performed, the playout game score is used to update the scores of twin nodes and any nodes along the original path from the root node. The score is not propagated up paths from twin nodes to the root node.

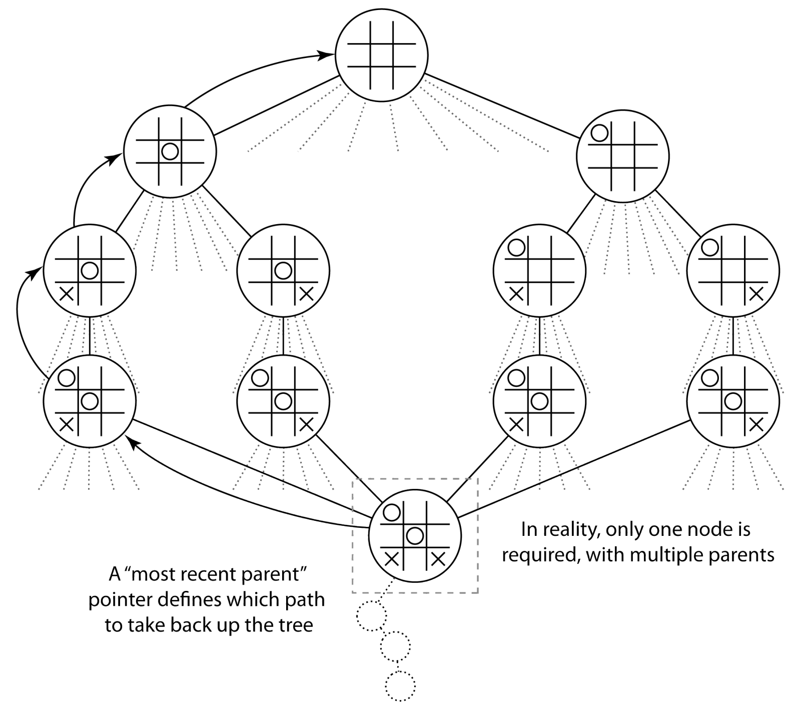

In reality, only one node is required, with multiple parents
A "most recent parent" pointer defines which path to take back up the tree

Figure 6.4: Diagram showing the true nature of the suggested new UCT algorithm – all twin nodes are merged into one and each node can have multiple parents, creating a search web instead of a search tree. Each node can have a "most recent parent" pointer which can be used to trace the correct path back up the web when propagating playout game scores back to the root node.

6.3.1 Reduction in branching factor

By how much does this new algorithm reduce the branching factor of a tree? Let us consider the simple corridor-based game situation of figure 5.13 as an example, with a board of length 24.

Let us label the number of nodes at level n in the tree N_n. In the classic UCT algorithm there are 24 possible board positions at ply 1, 24 × 23 possible positions at ply 2, and so on. So $N_1 = 24$, $N_2 = 24 \times 23$, and

$$N_n = \frac{24!}{(24-n)!} = n!\binom{24}{n}$$

In the new algorithm, the number of positions at level n is dependent on the number of possible combinations of lines drawn L_n. If we label each combination of lines l_i for $i = 1...L_n$

$$N_n^{\text{new}} = \sum_{i=1}^{L_n} b(l_i)$$

where $b(l_i)$ is the number of possible combinations of boxes with line combination l_i.

An upper bound can be placed on the $b(l_i)$. Since each l_i is a combination of n lines, it can create at most n boxes on the board. So $a(l_i)$, the number of boxes created by a line combination l_i, satisfies

$$a(l_i) \leq n$$

Since every box created is won by either player 1 or player 2, $b(l_i) = 2^{a(l_i)}$. Thus the number of possible combinations of boxes $b(l_i)$ satisfies

$$b(l_i) \leq 2^n$$

And since $L_n = \binom{24}{n}$, we can say that

$$\begin{aligned} N_n^{\text{new}} &= \sum_{i=1}^{L_n} b(l_i) \\ &\leq \sum_{i=1}^{L_n} 2^n \\ &= 2^n\binom{24}{n} \end{aligned}$$

Currently UCT wastes valuable iterations through unnecessarily analysing the same position multiple times (on different branches of its search tree). The new implementation of the algorithm is reducing the branching factor by a factor of $\frac{n!}{2^n}$. Mathematically speaking, this is a huge reduction – it grows exponentially with n.

There are many more complicated board games than Dots & Boxes, but the fact that classical UCT creates a different node for each different move order leading to a position means that in general, the new algorithm will reduce branching factor. This reduction should improve UCT's performance by allowing it to create deeper, more focused trees.

6.3.2 Trade-offs

There are, of course, trade-offs to the advantages of this “search web” approach.

One disadvantage arises in identifying the “twin” relationship. When adding a node to the search web, its board position must be compared to that of all the current nodes in the web (to see if the new node needs to be merged with a current one). This will clearly introduce a processing overhead to the algorithm. So although this altered algorithm would be stronger for a given number of iterations, the iterations would be slower to execute.

However, quick methods for board-matching may be possible by using techniques such as hashtables and game-specific methods (for example, in games where one counter is placed on each turn, new nodes only need to be compared with nodes at the same level in the search tree). The winning conditions of some games require some form of board-matching – for example, one of the rules of chess is that if any position is repeated more than 3 times, the game is a draw. Developing quick board-matching algorithms for these games will therefore be an integral part of programming the UCT algorithm in the first place. The extra processing required to search for board positions matching a new node might not represent a huge loss. Also note that only one matching node need be found for the merge to take place – in some cases this node might be found early in the board-matching process.

The board-matching stage of this process may also allow UCT to improve its understanding of symmetry. The function used to compare board positions could also be set up to match board positions which are identical by rotational/reflective symmetry. This would have the effect of bringing together continuations which the classic UCT algorithm considers completely separate. Again, this reduces branching factor, allowing the new UCT algorithm to see deeper into the position, and should produce stronger play for a given number of iterations.

Another disadvantage with the altered algorithm is that each node can now have multiple parents. This means that care must taken to trace the correct path back when ascending back to the root node after making a playout. However, if using the single return path approach shown in figure 6.3, a node does not need to have memory of *all* of its parents – it only needs to remember which parent the current playout descended through. If each node holds a “most recent parent” pointer, it only has to follow this pointer when propagating scores back up the path of decent.

The processing overheads of this approach will result in fewer iterations being completed per unit time, but there are undoubtedly advantages in reducing branching factor and improving value estimation by adopting this search web approach. A rigorous analysis of whether or not it might create a stronger UCT algorithm, and whether inspecting the shapes of its search webs might lead to better game playability prediction than inspecting classic UCT’s search trees is suggested as future work.

Chapter 7

Conclusion

Although game-playing is a recreational activity, it is often a testbed for innovative AI research techniques. Work on game-playing dovetails with many others areas of AI including optimisation and planning. UCT is a very new game-playing algorithm – first documented just four years ago – which incorporates two distinguishing features: it uses Monte-Carlo simulations to estimate the value of nodes in its search tree, and it samples actions selectively according to its current estimation of the values of those actions.

These two features make UCT a rather unique approach to game playing, but UCT's popularity and impact in academic circles have been the result of its simple implementation, extensibility to a variety of games, and its strength as a game playing algorithm. The game of Go, which has now taken over from chess as the frontier of AI game-playing, has always been very difficult to play for computers due to its huge number of potential continuations. However, in just four years, UCT has brought about advances in computer Go that were previously thought to remain many decades away, beating strong amateur Go players even on championship-sized boards.

The Monte-Carlo simulations used by UCT have led to a whole new area of research, Monte-Carlo Tree Search. The fact that random playouts are used to estimate node values means that UCT can play any deterministic turn-based game, provided it can access the rules and dynamics of that game through various functions, without calling on any prior knowledge or heuristics. Its selective sampling method is in many ways an even more interesting feature: it adopts a very human approach, sampling moves which currently look good more often than those which look bad. This gives each UCT tree its own shape which is uniquely related to the game and the exact position it is currently playing.

UCT's unique approach to tree search means that it was the perfect candidate for being applied to a new area of research: automatically assessing board game quality. This research has focused on finding out whether or not this was possible. Browne has already shown that board game quality can be assessed by using statistics garnered from computer self-play[6]. Given the relationship between UCT tree shape and game position, there was a chance that the tree shape might also be related to the quality of the game being played.

So the hypothesis of this research was that a machine learning system, trained on UCT trees for a selection of games, could accurately predict the quality of a new game it was presented with. If they hypothesis was shown to hold, then this system might well be used to predict the quality of computer-generated board games in the future. The system would be able to inspect a large set of automatically-generated games and give an initial prediction of which games were good, and which were bad.

Given that most automatically-generated games are likely to be utterly unplayable, it would also be of interest if a machine learner could be found which could distinguish playable games from unplayable ones. A human could then check those games deemed playable to see if they were indeed playable, and if so, how good they were. Even predictive accuracy in this binary classification task would be of huge benefit in helping to reduce the work required to extract good games from the intial set.

The research contained in this paper has involved first creating a game-playing system around the UCT algorithm, then coding up several games for UCT vs. UCT play. Some of these games were "broken", in order to provide the machine learning systems with examples of unplayable games. Three different-strength UCT algorithms were played against each other, performing 100 instances of each game. Statistics from all the UCT trees created during these games were extracted, recorded in CSV files, and finally aggregated into a form amenable to machine learning.

Machine learnering systems were then trained on these attributes. One machine learning system, Neural Networks, was tested for its predictive accuracy with respect to game quality. Three ML techniques were tested for their predictive accuracy on game playabilty: Neural Networks, Decision Trees and Support Vector Machines. The techniques were assessed using a cross-validation procedure that allows fair comparison of their expected predictive accuracy on new data, not just how well they were able to classify the training data.

The results for game quality prediction did not support this paper's first hypothesis. Even the best-performing network shape and training time setups for ANN learning showed a very poor predictive performance. However, in the task of predicting game playability, there was more success.

The headline result of this research is that one machine learning system, the decision tree trained on 100,000-iteration UCT data split by player number, showed far better predictive accuracy than the results which would be expected from a naive classifier. This result was strongly statistically significant, offering very strong evidence in favour of the second hypothesis.

This classifier was able to predict game playability with predictive accuracy of 82%, meaning that 82% of its predictions should be correct if it is extended to further data. This is a very positive result, and furthermore, given its 100% recall for unplayable games, this classifier would be superbly effective if it were used for the task of vetting a selection of automatically-generated board games. Extrapolating these results implies that every game that it deems playable should be playable, so despite missing some playable games it would leave no work to be done by the human checker at the next stage of the process.

It is very unlikely that this machine learning system would display 100% recall for unplayable games over a larger test set. As discussed in chapter 6, the small sample sizes means that a certain amount of caution must be taken in interpreting the results, and this research would benefit from confirmation on a larger scale. However, the predictive accuracy achieved by this classifier does suggest that further work in this area could bear fruit, since the availability of more data would not only provide a more definitive evaluation of the hypothesis, but would undoubtedly increase the chance of the machine learning systems finding a link between UCT tree shape and playability.

Computers are now some of the best board game players in the world, outstripping human performance in many popular games such as Chess and Checkers. The automatic-generation of board games and assessment of their quality is an intruiging area that has, until now, been very little explored. This research has demonstrated a machine learning system which has the potential to provide a very effective first line of anlaysis in determining whether automatically-generated board games in the future. Given the contributions made by computers in so many other areas of society, and their position at the forefront of board game playing, it would be fitting if they could one day take an active role in board game creation, allowing automatically-generated board games to hold a place alongside the other classic games which permeate our culture.

Appendix A

List of games coded

Achi

(quality rating: 2)

Achi is similar to Noughts & Crosses, and is played on the same board with the object of getting three-in-a-row. However, once three pieces each have been placed, play continues in a style similar to Nine Men's Morris, moving a piece into an empty neighbouring square on the board.

See `http://en.wikipedia.org/wiki/Achi_(game)` for more details

Alak

(quality rating: 3)

Alak is a one-dimensional version of the game Go. Players try to take territory by placing stones in such a way that they trap their opponent's stones. The game is won when either player cannot make a move, when the winner is the player with the most stones.

See `http://en.wikipedia.org/wiki/Alak_(board_game)` for more details

Dots & Boxes

(quality rating: 3)

Dots & Boxes is played by drawing lines on an orthogonal grid. When a player creates a box by drawing a line, they capture this box and they get to play again. The winner at the end of the game is the player with the most boxes captured.

See `http://en.wikipedia.org/wiki/Dots_and_boxes` for more details

Breakthrough

(quality rating: 5)

Breakthrough is played on a chessboard with two armies of 16 pawns. These pawns move exactly like chess pawns, with the exception that they are allowed to move one square directly forward if the square being moved to is empty. The object is to get a pawn to the opponent's side of the board. Breakthrough cannot end in a draw.

See `http://en.wikipedia.org/wiki/Breakthrough_(board_game)` for more details

Checkers

(quality rating: 3)

The classic game of moving pieces over the dark squares of a chess board. The object is to capture all of your opponent's pieces. Capturing is compulsory if possible, and a capturing piece can make multiple captures in one turn (and must do so if further captures exist). Pieces become kings, which are free to move in any direction, if they reach the opponent's side of the board. If the game has gone 50 moves without a capture and 50 moves without a piece moving towards crowning, it is a draw

See `http://en.wikipedia.org/wiki/Checkers` for more details

Connect4

(quality rating: 3)

The well-known game where players take turns placing pieces into a vertical board aiming to create 4 pieces in a row.

See `http://en.wikipedia.org/wiki/Connect4` for more details

Dipole

(quality rating: 4)

Dipole is played with checkers pieces on a checkers board, but with very different game mechanics. Players move their forces from a single starting square, aiming to capture all their opponent's pieces. Pieces can only move forwards; if a piece is on the forward edge of the board, it may only move off the board and is considered captured. When a player captures all his opponent's pieces, leave them with no legal moves, or forces them to move their last piece off the board, they win.

See `http://www.marksteeregames.com/Dipole_rules.pdf` for more details

The Indonesian Finger Game

(quality rating: 2)

The Indonesian Finger Game is not a board game, but can be modelled as such. It is played with each player holding out two hands, holding out a number of fingers on each hand. On their turn, players can add the fingers from one of their hands to one of their opponent's (by touching their hand to their opponent's). If a player's hand reaches 5 fingers or more, it is reset to none. When a player has no fingers on one hand they can use a turn to split the fingers from their other hand across both hands. If a player has no fingers on either hand, they lose the game.

See `http://www.boardgamenews.com/oldsite/index.php/boardgamenews/comments/valerie_putmanthe_indonesian_finger_game/` for more details

Forms

(quality rating: 3)

Forms is played on a rectangular board (4×6 was chosen for the purposes of this research). The whole board is occupied at the beginning of the game, half for each player. Players take turns in moving a piece to capture an opponent's piece. After this capture any pieces not connected to the remaining group are removed from the board. The last player *with* pieces on the board wins the game.

See `http://www.gamerz.net/pbmserv/forms.html` for more details.

Kalah

(quality rating: 5)

Kalah is a Mancala variant, played on a board where each player owns 6 standard squares on their side of the board, and a store. Pieces are distributed from one of the standard squares around the board in a circular manner, with any pieces put into a player's store staying there for the rest of the game. A player can capture extra pieces for their store with well-judged moves. At the end of the game the player with most pieces in their store wins the game.

See `http://en.wikipedia.org/wiki/Kalah` for more details

Nine Men's Morris

(quality rating: 4)

Nine Men's Morris is played on a square board, with a specific connection structure defining which squares neighbour which others (see 3.9). Players take turns placing pieces: when a player creates a "mill" of three pieces in a row, they may remove an opponent's piece. Once all pieces are placed, play continues by moving pieces to a neighbouring square. The first player with less than 3 pieces remaining loses.

See `http://en.wikipedia.org/wiki/Nine_Men's_Morris` for more details

Noughts & Crosses

(quality rating: 2)

The classic game of three-in-a-row

See `http://en.wikipedia.org/wiki/Noughts_and_crosses` for more details

Noughts & Crosses (alternative rules)

(quality rating: 2)

A variant of Noughts & Crosses where each player may place a nought or a cross at any stage, and the object is for a player to create three-in-a-row of either noughts or crosses.

Othello

(quality rating: 5)

Othello is played by placing pieces on a square 8×8 board. There are four pieces (2 black, 2 white) in the centre of the board at the beginning of the game. Players take turns laying pieces; a player may only lay a piece in such a way that it captures his opponent's pieces (in a style similar to Alak, but in any direction on the board). Captured pieces are flipped to the capturing player's colour.

See `http://en.wikipedia.org/wiki/Reversi` for more details.

Oware

(quality rating: 5)

Oware is another Mancala variant. Players do not have a store, but may capture any pieces when their final distributed piece lands in on an opponent's square with exactly 2 or 3 pieces in it (including the piece being added). The player with the most pieces captured at the end of the game wins.

See `http://en.wikipedia.org/wiki/Oware` for more details

Qirkat

(quality rating: 3)

Qirkat is played like checkers, except that it is played on a 5×5 board with vertical, horizontal and diagonal moves possible. Captures occur like checkers and are compulsory, and multiple captures for the capturing piece are possible and compulsory just like checkers. The player who loses all their pieces loses.

See `http://en.wikipedia.org/wiki/Alquerque` for more details

Qubic

(quality rating: 3)

Qubic is a 3-D version of Noughts & Crosses, played on a $4 \times 4 \times 4$ board. The object is to get 4 pieces in a row in any horizontal, vertical or diagonal direction.

See `http://en.wikipedia.org/wiki/Qubic` for more details.

The "broken" (unplayable) games created from initial set are described in section 3.3.

Appendix B

Features used to describe UCT trees

Number of legal moves available (branching factor)

Tree depth

The score:visits ratio of the best available move
To be more precise, the score:visits ratio of the node pointed to by the best available move

The score:visits ratio of the worst available move

The spread in score:visits ratio between the best and worst available moves

The spread in score:visits ratio between the best and $\frac{n}{3}$ available moves This measure is the spread in score between the 1st move and the $\frac{n}{3}$th move, when moves have been ordered by average score (score:visits ratio). When judging a position, a human player is likely to dismiss truly poor moves very quickly. This measure was supposed to give a sense of the spread in score among a more plausible group of moves.

Mean of level-1 score:visit ratios
The mean of the score:visits ratios over all nodes at level 1 in the UCT search tree.

Variance of level-1 score:visit ratios
The variance of the score:visits ratios over all nodes at level 1 in the UCT search tree.

Fraction of leaf nodes
The fraction of leaf nodes in the UCT tree: nodes which have been visited fewer times than they have legal moves available.

Fraction of UCT nodes
The fraction of nodes in the UCT tree which have been visited more times than they have legal moves available. Hence during UCT iterations these nodes will be applying the UCB formula to determine which action to sample.

Effective branching factor

A measure of the number of plausible moves in the position. Defined to be the number of moves with score within one standard deviation of the best move's score (standard deviation of all moves).

Average node depth

The average depth of all nodes in the tree: the sum of depths of all nodes in the tree, divided by the number of nodes.

Average branching factor

In a sense, the "average width" of the tree. The sum of the branching factors of each node in the tree, divided by the number of nodes.

Variance in maximum depth of level-1 subtrees

The maximum depth of each level-1 subtree (subtrees with level-1 nodes as their root node) was recorded, and the variance of these maximum depths was calculated.

Variance in average width of level-1 subtrees

The average width of each level-1 subtree (subtrees with level-1 nodes as their root node) was recorded, and the variance of these average widths was calculated.

Sum of score differences as UCT tree grows

The average score of each move was recorded after 200, 500, 1,000, 5,000, 10,000, 50,000 and 100,000 iterations of the UCT algorithm. The sum of the absolute value of the differences between each iteration value was calculated, to give a measure of how much the value estimations had changed as the algorithm went on.

Appendix C

Cross-validation results

Full tables of the cross-validation results and precision and recall rates for each of the learning systems are shown over the next few pages.

Decision tree learner
1,000 iterations

		Predicted: Unplayable	Predicted: Playable	
True	Unplayable	6	5	11
True	Playable	6	11	17
		12	16	28

	Unplayable	Playable
Precision	50%	69%
Recall	55%	65%

Decision tree learner
10,000 iterations

		Predicted: Unplayable	Predicted: Playable	
True	Unplayable	8	3	11
True	Playable	9	8	17
		17	11	28

	Unplayable	Playable
Precision	47%	73%
Recall	73%	47%

Decision tree learner
100,000 iterations

		Predicted: Unplayable	Predicted: Playable	
True	Unplayable	6	5	11
True	Playable	6	11	17
		12	16	28

	Unplayable	Playable
Precision	43%	64%
Recall	55%	53%

Table C.1: Cross-validation results and precision/recall rates for decision tree learners

SVM learner
1,000 iterations

		Predicted		
		Unplayable	Playable	
True	Unplayable	6	5	11
	Playable	8	9	17
		14	14	28

	Unplayable	Playable
Precision	17%	44%
Recall	18%	41%

SVM tree learner
10,000 iterations

		Predicted		
		Unplayable	Playable	
True	Unplayable	2	9	11
	Playable	10	7	17
		12	16	28

	Unplayable	Playable
Precision	17%	44%
Recall	18%	41%

SVM tree learner
100,000 iterations

		Predicted		
		Unplayable	Playable	
True	Unplayable	4	7	11
	Playable	5	12	17
		9	19	28

	Unplayable	Playable
Precision	44%	63%
Recall	36%	71%

Table C.2: Cross-validation results and precision/recall rates for SVM learners

NN learner
(best setup)
1,000 iterations

		Predicted		
		Unplayable	Playable	
True	Unplayable	4.7	6.3	11
	Playable	6.5	10.5	17
		11.2	16.8	28

	Unplayable	Playable
Precision	42%	63%
Recall	43%	62%

NN learner
(best setup)
10,000 iterations

		Predicted		
		Unplayable	Playable	
True	Unplayable	2.1	8.9	11
	Playable	4.0	13.0	17
		6.1	21.9	28

	Unplayable	Playable
Precision	34%	59%
Recall	19%	76%

NN learner
(best setup)
100,000 iterations

		Predicted		
		Unplayable	Playable	
True	Unplayable	5.7	5.3	11
	Playable	8.7	8.3	17
		14.4	13.6	28

	Unplayable	Playable
Precision	40%	61%
Recall	52%	49%

Table C.3: Cross-validation results and precision/recall rates for Neural Network learners. Tables shown are from the best network setup (after testing several combinations of network shapes and training times) measured in terms of an average correlation coefficient over 10 repetitions of the cross-validation process.

Each techniques was then re-tested, but with the dataset split by player: trees for moves with player 1 to play were split from those for moves with player 2 to play before aggregation. The results for this data are shown below.

Decision tree learner
1,000 iterations

(Data split
by player
number)

		Predicted: Unplayable	Predicted: Playable	
True	Unplayable	5	6	11
	Playable	9	8	17
		14	14	28

	Unplayable	Playable
Precision	36%	57%
Recall	45%	47%

Decision tree learner
10,000 iterations

(Data split
by player
number)

		Predicted: Unplayable	Predicted: Playable	
True	Unplayable	7	4	11
	Playable	8	9	17
		15	13	28

	Unplayable	Playable
Precision	47%	69%
Recall	64%	53%

Decision tree learner
100,000 iterations

(Data split
by player
number)

		Predicted: Unplayable	Predicted: Playable	
True	Unplayable	11	0	11
	Playable	5	12	17
		16	12	28

	Unplayable	Playable
Precision	69%	100%
Recall	100%	71%

Table C.4: Cross-validation results and precision/recall rates for Decision tree learners, when using data split between trees for player 1 and trees for player 2

SVM learner
1,000 iterations

(Data split by player number)

		Predicted		
		Unplayable	Playable	
True	Unplayable	3	8	11
	Playable	9	8	17
		12	16	28

	Unplayable	Playable
Precision	25%	50%
Recall	27%	47%

SVM tree learner
10,000 iterations

(Data split by player number)

		Predicted		
		Unplayable	Playable	
True	Unplayable	6	5	11
	Playable	10	7	17
		16	12	28

	Unplayable	Playable
Precision	38%	58%
Recall	55%	41%

SVM tree learner
100,000 iterations

(Data split by player number)

		Predicted		
		Unplayable	Playable	
True	Unplayable	2	9	11
	Playable	8	9	17
		10	18	28

	Unplayable	Playable
Precision	20%	50%
Recall	18%	53%

Table C.5: Cross-validation results and precision/recall rates for SVM learners, when using data split between trees for player 1 and trees for player 2

NN learner
(best setup)
1,000 iterations

(Data split
by player
number)

		Predicted		
		Unplayable	Playable	
True	Unplayable	3	8	11
	Playable	2.8	14.2	17
		5.8	22.2	28

	Unplayable	Playable
Precision	52%	64%
Recall	27%	84%

NN learner
(best setup)
1,000 iterations

(Data split
by player
number)

		Predicted		
		Unplayable	Playable	
True	Unplayable	4.5	6.5	11
	Playable	4.7	12.3	17
		9.2	18.8	28

	Unplayable	Playable
Precision	49%	65%
Recall	41%	72%

NN learner
(best setup)
1,000 iterations

(Data split
by player
number)

		Predicted		
		Unplayable	Playable	
True	Unplayable	3.9	7.1	11
	Playable	6.2	10.8	17
		10.1	17.9	28

	Unplayable	Playable
Precision	39%	60%
Recall	35%	64%

Table C.6: Cross-validation results and precision/recall rates for Neural Network learners, when presented with data split between trees for player 1 and trees for player 2. Tables shown are from the best network setup (after testing several combinations of network shapes and training times) measured in terms of an average correlation coefficient over 10 repetitions of the cross-validation process.

Bibliography

[1] D. Parlett. *The Oxford history of board games.* Oxford University Press, 1999.

[2]

[3] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. *15th European Conference on Machine Learning*, pages 282–293, 2006.

[4] M. Enzenberger and M. Müller. Fuego–an open-source framework for board games and go engine based on monte-carlo tree search. 2009.

[5] Y. Björnsson and H. Finnsson. CadiaPlayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[6] C. Browne. *Automatic Generation and Evaluation of Recombination Games.* PhD thesis, Queensland University of Technology, 2008.

[7] A. Y. Ng M. Kearns, Y. Mansour. A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes. *Machine Learning*, 2002.

[8] P. Fischer P. Auer, N. Cesa-Bianchi. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 2002.

[9] J. M. Thompson. Defining the abstract. *The Games Journal*, 2000.

[10] W. Kramer. What makes a game good? *The Games Journal*, 2000.

[11] G.F. Cooper, C.F. Aliferis, R. Ambrosino, J. Aronis, B.G. Buchanan, R. Caruana, M.J. Fine, C. Glymour, G. Gordon, B.H. Hanusa, et al. An evaluation of machine-learning methods for predicting pneumonia mortality. *Artificial Intelligence in Medicine*, 9(2):107–138, 1997.

[12] D.A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network, 1989.

[13] K.F. Lee. *Automatic speech recognition: the development of the SPHINX system.* Kluwer Academic Pub, 1989.

[14] Wikipedia. Nine men's morris board with coordinates, 2006. [Online; accessed 15 September 2010].