WITS
UNIVERSITY

The Effect of Simulation Bias on Action Selection in Monte Carlo Tree Search

Steven James

475531

Supervised by: Pravesh Ranchod, George Konidaris & Benjamin Rosman

August 2016

A dissertation submitted to the Faculty of Science, University of the Witwatersrand,
in fulfilment of the requirements for the degree of Master of Science.

Abstract

Monte Carlo Tree Search (MCTS) is a family of directed search algorithms that has gained widespread attention in recent years. It combines a traditional tree-search approach with Monte Carlo simulations, using the outcome of these simulations (also known as playouts or rollouts) to evaluate states in a look-ahead tree. That MCTS does not require an evaluation function makes it particularly well-suited to the game of Go — seen by many to be chess's successor as a grand challenge of artificial intelligence — with MCTS-based agents recently able to achieve expert-level play on 19×19 boards. Furthermore, its domain-independent nature also makes it a focus in a variety of other fields, such as Bayesian reinforcement learning and general game-playing.

Despite the vast amount of research into MCTS, the dynamics of the algorithm are still not yet fully understood. In particular, the effect of using knowledge-heavy or biased simulations in MCTS still remains unknown, with interesting results indicating that better-informed rollouts do not necessarily result in stronger agents. This research provides support for the notion that MCTS is well-suited to a class of domain possessing a smoothness property. In these domains, biased rollouts are more likely to produce strong agents. Conversely, any error due to incorrect bias is compounded in non-smooth domains, and in particular for low-variance simulations. This is demonstrated empirically in a number of single-agent domains.

Declaration

I, Steven James, hereby declare the contents of this research proposal to be my own work unless otherwise explicitly referenced. This research proposal is submitted for the degree of Master of Science (Dissertation) at the University of the Witwatersrand, Johannesburg. This work has not been submitted to any other university, nor for any other degree.

Signature

15/08/2016

Date

Acknowledgements

I'd like to take this opportunity to thank all those who have helped me along the way. There is a problem, however, in that I am no James Joyce, and thus my words do absolutely no justice to the depth of my gratitude. I shall nonetheless proceed with the knowledge that the following is wholly inadequate.

To begin with, I am indebted to my supervisors Benji, George and Pravesh,[1] whose constant advice and (in particular) encouragement were unwavering throughout this process. Thank you to Benji and George for setting aside time every week without fail, despite your busy schedules and the accursed daylight savings time!

To Shun Pillay, Mohsin Desai and Brian Maistry from the university's MSS department: thank you for always being ready and willing to help out with any requests I had or fix any problems that I inadvertently caused (*I swear — that fork bomb was a complete accident!*).

To everyone I've come across during this time (*if you're reading this, then it's probably you*): thank you for the camaraderie, the discussions, and for making this experience a pleasure. Dean and Jeremy in particular.

Last, but certainly not least, to my parents for their love, support and patience. Thank you for giving me the opportunity to do everything I've ever wanted. A final thank you to my mom for going above and beyond the call of duty in proofreading this thesis (*but please leave my Oxford commas be!*). Any mistakes are mine, and mine alone.

[1] Names randomly permuted using a PRNG.

Go is to Western chess what philosophy is to double-entry accounting.

– Trevanian, *Shibumi*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Classic two-player games have always served as important proving grounds in the field of artificial intelligence. For many decades, the king of these was arguably the game of chess. However, with DEEP BLUE's victory over then world champion Garry Kasparov in 1996 and the subsequent dominance of chess programs over humans, focus has shifted elsewhere. Recently the ancient game of Go has received particular attention. The game's state-space complexity, far greater than even that of chess, renders it impervious to the brute-force techniques that were so successful in many other games, such as checkers and Othello. It is thus unsurprising that Go has supplanted chess as a grand challenge for artificial intelligence [Gelly *et al.* 2012].

For many years, research into Go produced programs of only amateur-level strength. A major reason for this resided in Go's complex strategy that belies its simple rules. Fundamental concepts such as *life and death*, *seki* and *semeai*[1] have all been understood by humans for centuries, but remain difficult to encode precisely. This extends to the evaluation of board positions: whereas games such as chess have readily available heuristics for positional evaluation (a queen is more valuable than a pawn, for instance), Go does not. Every piece (*stone*) has the same value and the effect of a single move may not be readily apparent until dozens of moves later. Furthermore, the large state-space and game-tree complexity[2] of Go effectively preclude the full-width techniques used in other games.

Previously, Go programs made use of expert knowledge in the form of pattern databases, combined with popular search algorithms (such as alpha-beta pruning) and domain-specific enhancements [Cai and Wunsch II 2007]. However, a paradigm shift occurred with the introduction by Coulom [2006] of the Monte Carlo Tree Search (MCTS) algorithm. A variation of MCTS, known as the UCT (Upper Confidence bound applied to Trees) algorithm [Kocsis *et al.* 2006], further improved matters and allowed programs to compete against experts on smaller 9×9 boards. More recently, a combination of MCTS with deep neural networks [Silver *et al.* 2016] was able to defeat

[1] Refer to Appendix A for details.

[2] Allis [1994] estimates the game-tree complexity (that is, the total number of games that can be played) on a 19×19 board at 10^{360}, whilst Tromp [2016] calculates the exact number of legal positions reachable from the start (the state-space complexity), approximated here as $\sim 2.081681994 \times 10^{170}$. For comparison, the game-tree complexity of chess is estimated at 10^{123} while its state-space complexity is 5×10^{52}.

world champion Lee Se-dol on a full-sized 19×19 board.

This thesis focuses on the popular UCT algorithm which, despite its shortcomings [Domshlak and Feldman 2013], is widely used in practice. As with all members of the MCTS family, UCT consists of four distinct phases: selection, expansion, simulation and backpropagation. Amongst its many advantages, including the fact that it is both practically and conceptually simple, Browne *et al.* [2012] cite its three most important characteristics:

- UCT can return a valid result at any stage of its computation, even if interrupted (a so-called anytime algorithm);
- it is an asymmetric directed search, which allows it to focus on more promising lines of play (this is particularly important in high-branching games where investigating all lines, as per a classic minimax search, results in only very shallow depths being reached);
- the basic UCT algorithm requires no heuristic or domain-dependent knowledge whatsoever. This gives it huge significance, not only in the game of Go (where strong heuristics are difficult to encode), but also in a multitude of fields such as general game playing [Björnsson and Finnsson 2009; Méhat and Cazenave 2010], POMDPs [Silver and Veness 2010] and Bayesian reinforcement learning [Guez *et al.* 2013].

A great deal of analysis on MCTS revolves around the selection phase of the algorithm, which provides theoretical convergence guarantees and upper-bounds on the regret [Kocsis and Szepesvári 2006; Coquelin and Munos 2007]. Conversely, very little is known about the simulation phase. MCTS calls for this phase to be performed by randomly selecting actions until a terminal state is reached. The strategy for executing a simulation is known as a *rollout* or *playout policy*. In Go, for instance, MCTS randomly selects and plays moves for both players from a given position until the game is over. The result of the simulated game (win or loss) is then propagated to the root of the game tree. Averaging these results over many iterations provides a fairly accurate measure of the strength of the initial position, despite the fact that the simulation is completely random. As the outcome of the simulations directly affects the entire algorithm, the manner in which they are performed has a major effect on the overall strength of MCTS.

A natural assumption to make is that completely random simulations are not ideal, since they do not map to realistic or rational actions. A different approach is that of so-called *heavy playouts*, where moves are intelligently selected using domain-specific rules or knowledge. These simulations are said to be biased, since their returns depend in part on the user-specified domain knowledge. Counterintuitively, results indicate that using these stronger rollouts can actually result in a decrease in overall playing strength [Gelly and Silver 2007].

This implies that a compromise needs to be found between playing strength and the sampling of a diverse range of moves. In Go, simple handcrafted simulation strategies often outperform more sophisticated rollout policies — Wang and Gelly [2007], for instance, prioritise certain moves in their program MoGo, but play randomly otherwise. This suggests that the ideal simulation policy is one which is not overly deterministic, while simultaneously avoiding truly disastrous moves. Silver and Tesauro [2009] posit that it is not the strength of the playout that is important, but rather that the two players in the simulation are balanced. Thus any errors in play (severe blunders notwithstanding) are cancelled out by the opponent's errors, producing a result that is more reflective of the initial position.

The aim of this research is to provide some insight into the effect of different rollout strategies

for MCTS in a variety of domains. To simplify matters and reduce the conflation that occurs when performing a multi-player simulation, the primary focus is that of single-agent domains. Of particular interest is identifying the domains for which MCTS is a good choice of algorithm, as well as the effect different kinds of simulations can have on the performance of MCTS in these domains. The results of this research provide a clear indication that the algorithm is well-suited to domains possessing a smoothness property. Furthermore, low-variance rollout policies are identified as potentially dangerous choices to use in the simulation phase. These conclusions will assist in applications of MCTS in a variety of fields, providing users some insight into what may or may not be successful without the need for trial-and-error testing.

Having briefly introduced the problem area, the remainder of this thesis is structured as follows: Chapter 2 formalises the Markov decision process (MDP) model that is used to represent the class of domain into which many games fall. Included in this chapter is a discussion of traditional game-playing approaches, as well as the more recent MCTS algorithm. Chapter 3 provides some examples that illustrate the need for more understanding of MCTS, while Chapter 4 focuses on the research methods. The research questions are formally presented in Section 4.2, followed by an outline of the experimental methodology used in order to answer these questions. The results of the experiments undertaken are presented in Chapters 5 and 6, together with a discussion of their implications. Lastly, Chapter 7 summarises the contribution of this research and provides ideas for future work.

Chapter 2

Background and Related Work

The previous chapter mentioned the Monte Carlo Tree Search (MCTS) algorithm responsible for producing significant improvements in Go programs. This chapter provides the background to MCTS, as well as the notation used throughout this thesis. Many domains, and especially two-player games, can be formulated under the mathematical framework of Markov decision processes (MDPs). Section 2.1 provides a precise description of this class of domain, as well as reinforcement learning — an approach to decision making well-suited to such environments. This is followed by a discussion of traditional algorithms used by game-playing programs and a full explanation of MCTS and its workings (Sections 2.2 and 2.3). Lastly, Section 2.4 presents a subset of popular MCTS enhancements applicable to numerous fields.

2.1 Reinforcement Learning

This section provides a brief introduction to reinforcement learning, a paradigm concerned with learning through trial-and-error interaction within an environment. The section discusses some of reinforcement learning's underpinning theory and pertinent methods for producing solutions.

2.1.1 Sequential Decision Problems

A large subset of problems in the field of artificial intelligence, including Go, can be classified as sequential decision problems. In these cases, the decision-making entity — the *agent* — attempts to maximise its utility[1] by making a sequence of decisions. At each time step, the agent receives observations from its environment[2] and performs an action accordingly. As a consequence of its action, the agent receives feedback (*reward*) and finds itself, by way of some transition function, in a new state. Whereas the rewards represent only the immediate outcome, the utility captures

[1] The utility is a measure of the agent's satisfaction with states [Russell and Norvig 2010]. This is characterised by the notion of a state-value function (Section 2.1.2).

[2] Sutton and Barto [1998] define the environment as anything that cannot be arbitrarily changed by the agent.

the long-term consequences of actions. The goal of the agent is simply to maximise the rewards it receives over time.

More formally, let $\mathcal{S}$ be the set of possible environment states and $\mathcal{A}(s)$ the set of possible actions at state s. At time t, the agent receives observation $s_t \in \mathcal{S}$ and selects an action $a \in \mathcal{A}(s_t)$. Action a is selected according to the agent's policy — a mapping[3] $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. The agent then receives a numerical reward $r_{t+1} \in \mathbb{R}$ and transitions to the new state s_{t+1}. The series of observations, actions and rewards constitutes the agent's *experience*. Reinforcement learning uses this experience to modify the agent's policy and maximise the total expected reward, given by

$$r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.1}$$

where $0 \leq \gamma \leq 1$ is used to discount future rewards.

Board games such as Go, chess and checkers can all be formulated as sequential decision problems with a number of attractive properties. Being turn-based games, the problems are temporally discrete. The games' reward functions (a mapping from perceived states to rewards) are simple: a positive reward for positions in which the agent wins and zero for states in which it loses. The environment is fully observable — the entire state of the system is known to the agent — and state transitions are completely deterministic. Furthermore, the action- and state-space are both finite, albeit large.

Owing to the simplifications that can be made as a result of the above characteristics, these types of domains are popular testbeds for new ideas and algorithms. Above all else, however, their primary advantage is that they can be formulated as a finite MDP.

2.1.2 Markov Decision Processes

A class of tasks that is particularly important to reinforcement learning is that of finite MDPs [Sutton and Barto 1998]. A sequential decision problem is an MDP if its transition function is *Markovian* — that is, the probability of reaching state s' from state s is dependent only on s and not on the history of earlier states [Puterman 2009]. Formally,

$$p(s_{t+1} \mid s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t, a_t) = p(s_{t+1} \mid s_t, a_t). \tag{2.2}$$

An finite MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where:

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}(s)$ is a finite set of actions available at state s
- $P(s' \mid s, a)$ is the state-transition model

[3] Mohri *et al.* [2012] observe that π is more precisely a stationary policy, since the choice action is independent of time. This is a simpler formulation than a non-stationary policy, but suffices for our purposes. However, in finite-horizon cases (in which a finite number of actions are allowed), a non-stationary policy may be required.

- $R(s, a, s')$ is reward function
- $\gamma \in [0, 1]$ the discount factor.

The ultimate aim of the agent is to learn the optimal policy π^*. To define what is meant by this, the concept of a state-value function is first required. Under any policy π, the value of state s is given by

$$V_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(s, a) \left[\sum_{s'} P\left(s' \mid s, a\right) \left(R(s, a, s') + \gamma V_\pi(s')\right) \right]. \tag{2.3}$$

For deterministic transition models and policies, the above reduces to a system of $|\mathcal{S}|$ equations in $|\mathcal{S}|$ unknowns, which admits a unique solution and can be solved efficiently using linear algebra techniques [Mohri *et al.* 2012]. A policy π^* is said to be optimal if $\forall s \in \mathcal{S}$, $V_{\pi^*}(s) = \max_\pi V_\pi(s)$. For convenience, the notation $V^*(s)$ is universally adopted in place of $V_{\pi^*}(s)$. The optimal policy values are given by the set of *Bellman equations*:

$$V^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s'} P\left(s' \mid s, a\right) \left[R(s, a, s') + \gamma V^*(s')\right]. \tag{2.4}$$

Again for deterministic transitions, the above are $|\mathcal{S}|$ equations in $|\mathcal{S}|$ unknowns; however, unlike Equation (2.3), they are non-linear. This means that fast linear algebra techniques cannot be applied. Instead, dynamic programming techniques such as value iteration are used [Bellman 1957]. Having solved the Bellman equations for the optimal value function, it is then trivial to derive the optimal policy π^*, which is greedy with respect to the optimal value function.

When a model of the environment is not readily available, it is often easier to learn the optimal policy using the action-value function $Q_\pi(s, a)$, which represents the expected outcome of taking action a in state s and thereafter following policy π. The optimal Q-value function Q^* can be written in terms of V^*:

$$Q^*(s, a) = \sum_{s'} P\left(s' \mid s, a\right) \left[R(s, a, s') + \gamma V^*(s')\right], \tag{2.5}$$

from which the optimal policy immediately follows.

2.1.3 Monte Carlo Methods

The techniques of the previous section require the complete probability distribution over transitions. Monte Carlo methods, on the other hand, only require transitions sampled from the distribution. These methods are commonly used to obtain numerical approximations to mathematical problems with no analytical or closed-form solution. An oft-cited example of this is using random sampling to approximate the value of π, achieved by inscribing a circle in a unit square. Points are then uniformly distributed inside the square at random. After a sufficiently large number of points have been placed, the ratio of points inside the circle to the total number of points approximates $\frac{\pi}{4}$ [Krauth 1998].

Monte Carlo methods are not only applied to purely mathematical problems. Abramson [1987] showed that Monte Carlo sampling could be used to approximate the value of game states, while Brügmann [1993] used Monte Carlo methods in his Go program. It was not until 2006, however, that Monte Carlo Go found widespread popularity. Monte Carlo approaches have also been applied to a variety of games, such as bridge [Ginsberg 2001], Scrabble [Sheppard 2002], backgammon [Tesauro and Galperin 1996] and poker [Billings *et al.* 2002], with great success.

At an intuitive level, the value of a game state s can be approximated by running simulations from s and averaging the returns. In a simulation, moves are randomly selected until the end of the game, at which point the result G of the game is returned. Assume that we receive reward G_i after performing the i-th simulation using policy π. The value of state s after n simulations is then simply

$$V_\pi(s) = \frac{1}{n} \sum_{i=1}^{n} G_i, \tag{2.6}$$

which converges to a fixed number in the limit [Chaslot 2010].

Note also that the value of state-action pairs $Q(s, a)$ can be estimated by Monte Carlo simulation [Gelly and Silver 2011]. Let $N(s, a)$ be the number of times action a was selected in state s and $N(s)$ be the number of times a simulation passed through s. Then:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s)} \mathbb{1}_i(s, a) G_i, \tag{2.7}$$

where $\mathbb{1}_i(s, a) = 1$ if a was selected in s during simulation i, and $\mathbb{1}_i(s, a) = 0$ otherwise.

2.1.4 Temporal Difference Learning

In the above discussion of Monte Carlo methods, one had to wait until the end of the episode for the return to be used. By contrast, temporal difference methods can update state values at the next time-step. These methods are said to *bootstrap*, since they update the estimate of the value of state s_t based on the values of subsequent states, themselves estimates. The simplest form of temporal difference algorithms, TD(0), uses only the value of the successor state s_{t+1}.

Both Monte Carlo and temporal difference methods update state values in the direction of some target value. The former's target is the actual return, while the latter's is the subsequent state's value. Let δ_t be the error between the state and target value. Then state-value updates take the form

$$V(s_t) \leftarrow V(s_t) + \alpha \delta_t, \tag{2.8}$$

where α is a step-size parameter that controls the learning rate.

Equations (2.9) and (2.10) give the errors for Monte Carlo methods and TD(0) respectively. In particular, Equation (2.10), which bears a resemblance to the Bellman equations (2.4), suggests

that the TD(0) algorithm adjusts the value of a state to make it more consistent with the next state's value.

$$\delta_t = G_t - V(s_t) \tag{2.9}$$

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.10}$$

Temporal difference learning need not just update its values from the next state — rather, values many steps into the future can be used. This is known as the TD(λ) algorithm, where $0 \leq \lambda \leq 1$ weights the strength of updates from subsequent states: TD(0) bootstraps from the immediate successor, while TD(1) updates from the final return only. In other words, TD(1) and Monte Carlo methods are equivalent. Naïvely updating the value of a state from rewards n steps into the future can be problematic, since it requires waiting for n rewards to be observed before the update can occur. In practice, *eligibility traces* are used to update the value of states incrementally, without having to wait for future rewards [Sutton and Barto 1998].

2.1.5 Function Approximation

For domains with large state-spaces such as Go, finding a solution to Equation (2.4) becomes intractable; furthermore, simply storing the states in memory is not possible. Unfortunately, this is often the case in real-world problems. To overcome this, the state-space is represented as a parameterised function. Previously the value function was represented as a table, with each entry holding the value of a single state. Instead, the state-space can be approximated by a set of *features* or basis functions $\boldsymbol{\phi}$, the dimension of which is much smaller than the size of the state-space.

As states with similar features are considered similar under this approach, function approximation's real strength is its ability to generalise to states not yet encountered [Russell and Norvig 2010]. The most common method of performing function approximation is to use a weighted linear combination of features to represent the value function, which is then represented in the compressed form

$$V(s) \approx \sum_{i=1}^{N} \theta_i \phi_i(s) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s), \tag{2.11}$$

where $N \ll |\mathcal{S}|$ is the dimension of $\boldsymbol{\phi}$, $\boldsymbol{\theta}$ is a weight vector that is learnt by a reinforcement learning algorithm, and $\boldsymbol{\phi}$ is a mapping $\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^N$.

Features are often binary (i.e. $\phi_i : s \to \{0, 1\}$) in the sense that they represent the existence of a particular component. In Go, a state may be represented by the existence of patterns at various points — Silver [2009], for instance, adopts exactly this approach in his program RLGO.

In addition to using function approximation in lieu of an explicit state-value mapping, it can also be used to construct a stochastic policy. Instead of defining features over the state space, features are defined for state-action pairs — that is, $\boldsymbol{\phi} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^N$. The policy can then be

parameterised by the weight vector $\boldsymbol{\theta}$. A common way of doing so is to construct a *softmax* policy

$$\pi_{\boldsymbol{\theta}}(s,a) = \frac{e^{\boldsymbol{\phi}(s,a)^T\boldsymbol{\theta}}}{\sum_b e^{\boldsymbol{\phi}(s,b)^T\boldsymbol{\theta}}}. \tag{2.12}$$

2.2 Game Tree Search

In this section, we discuss the full-width search methods that were so successful in games such as chess and checkers. These algorithms, such as minimax, proved ultimately to be unsuitable for Go, which led to the development of Monte Carlo-based algorithms and eventually MCTS (Section 2.3).

In his landmark paper in the field of computer chess, Shannon [1950] provides two strategies for playing chess: *type A* and *type B*. *Type B* strategies consist of investigating only a few promising-looking moves and calculating their variations. While this is the approach adopted by human players, chess engines instead make use of *type A* strategies, which are discussed in Section 2.2.2. While the content of his paper revolves around chess, Shannon's ideas are readily transferable to other two-player, turn-based games.

Before these ideas can be introduced, the notion of a game tree must first be discussed. The game tree is a tree in which each of its nodes represents a game state and its edges are moves [Kuhn 1953]. A single edge in the tree is known as a *ply*, representing a move by one side. Unfortunately, the number of successors of each node — the branching factor — of the game tree depends solely on the number of moves available to each side at a given game state. In certain games (such as Tic-tac-toe) there are only a few moves that can be made at any given point, which means that the branching factor of the game tree is low. If d is the maximum number of moves needed to complete a game (9 in the case of Tic-tac-toe) and b the branching factor, then the number of leaf nodes which represent completed games is at most b^d. The branching factor of Go, combined with its duration, means that the full game tree cannot be represented, even on a 9×9 board. Thus, unlike Tic-tac-toe, computing an exact solution is presently infeasible.

A program designed to play turn-based games can often be divided into two distinct components: a search algorithm and an evaluation function. The remainder of this section is devoted to a brief description of these components.

2.2.1 Evaluation Function

Many game-playing programs make use of a heuristic known as an evaluation function. In essence, this function accepts a game position and returns an estimate of the value of that position. It is therefore nothing more than a user-defined value function, often taking the form specified by Equation (2.11).

An advantage of such a function is that in zero-sum[4] games V can be used when considering the value of a state from each player's perspective. If position s is described by the differences in

[4]A zero-sum game is one in which the gains made by one player are, by definition, equal to the losses suffered by the other player.

the players' features, then $V(s)$ represents an evaluation function from the first player's perspective and $-V(s)$ from the second's perspective.

Depending on the game, there are a myriad of features that could be considered when devising an evaluation function — the chess engine DEEP BLUE used more than 8000 [Campbell *et al.* 2002]. All these features are derived from humans' understanding of the game, which is problematic in games such as Go where it is difficult to quantify them.

2.2.2 Minimax Theorem

Shannon's *type A* strategies refer to the application of the *minimax* theorem to games. Consider the two players Black and White. Assume that Black is attempting to maximise the evaluation function, while White attempts to minimise it. The optimal value of a state can then be calculated by its minimax value. Let $\mathcal{A}(s)$ be the set of legal moves at state $s \in \mathcal{S}$, and $\delta(s, a)$ represent the state reached by playing move a at state s. Assuming optimal play by both sides, the minimax value of a state is then given by the recursive function

$$\text{minimax}(s) = \begin{cases} V(s) & \text{if } s \text{ is terminal} \\ \max\limits_{a \in \mathcal{A}(s)} \Big(\text{minimax}\left(\delta(s,a)\right) \Big) & \text{if Black to play} \\ \min\limits_{a \in A(s)} \Big(\text{minimax}\left(\delta(s,a)\right) \Big) & \text{if White to play.} \end{cases} \tag{2.13}$$

The above equation clearly demonstrates the major flaw in minimax: the algorithm can only be used if it is able to reach all terminal states — a terminal state is a leaf node of the tree and corresponds to a state in which the game is completed, the value of which would reflect the outcome of the game. However, since traversing the entire tree is not feasible in many domains, it must be restricted to a certain depth. To illustrate, consider the game tree in Figure 2.1, which has been restricted to a 2-ply depth:

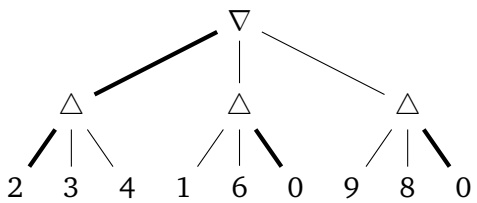


Figure 2.1: Example of the minimax algorithm. Optimal moves for both players are indicated by bold edges.

Assume that Black plays at the ∇ state, while White plays at $\triangle$ states. Black has three moves available, each of which leads to a different state, at which point it is White's turn to play. White's moves lead to other states which are then evaluated. The outcome is calculated using Equation (2.13) and is a simple case of backward induction. In this instance, the value of the root node is calculated to be 2.

2.2.3 Alpha-Beta Pruning

An issue with the minimax algorithm is that it must visit every node in the game tree. Since the number of nodes is exponential in the depth of the tree, only shallow depths can feasibly be searched. However, it is possible to compute the correct minimax value without examining every game state in the tree — that is, certain nodes can be disregarded or pruned from the game tree without the need to evaluate them.

A popular pruning technique is that of alpha-beta pruning. Although the idea cannot be attributed to any one person, Knuth and Moore [1975] refined the algorithm and provided a complete analysis of it. To illustrate the algorithm, consider Figure 2.2 which presents the same game tree as Figure 2.1:

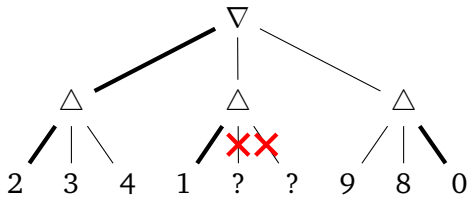


Figure 2.2: Example of alpha-beta pruning. Optimal moves for both players are indicated by bold edges, while edges to pruned nodes are crossed off.

Recall that the minimax algorithm visits nodes in a depth-first search manner. Assume that the leftmost subtree has been investigated. Black has calculated that playing his first move results in a score of 2. Now Black investigates the middle subtree and discovers that White's first response results in a score of 1. Therefore, without even considering the value of the other two leaf nodes in the subtree, Black can conclude that he is better off playing in the left subtree, since White can force a result of 1, which is worse than any return that ensues from the left subtree. Thus two leaf nodes have been pruned from the middle subtree.

Alpha-beta pruning can be applied not only to leaf nodes, but also to interior nodes, meaning entire subtrees can be disregarded. The name itself is derived from the two bounds α and β used to prune the game tree. α is used to store the best (maximum) score attained thus far for Black, given optimal play by White (when the nodes in the above diagram are pruned, $\alpha = 2$). Similarly, β represents the best (minimum) score for White. In the above diagram, only α is used to prune the tree. However, β, too, can be used to prune nodes. Alpha- and beta-cutoffs occur when nodes are pruned on these two bounds respectively.

One final point to note is the order in which leaf nodes are examined by the algorithm. Notice that in the rightmost subtree, White's best move was evaluated last — as a result, no pruning could take place. Conversely, a relatively strong move was considered first in the middle subtree, which produced a cutoff. Should moves be examined in order from worst to best, then no pruning takes place and the algorithm reduces to minimax. However, investigating moves that have been ordered optimally results in an examination of only $O(\sqrt{b^d})$ leaf nodes [Russell and Norvig 2010].

2.2.4 Iterative Deepening

Section 2.2.2 discussed limiting the depth of the game tree. However, this produces another complication — deciding on the depth. Since moves often need to be made within some time limit, it would be ill-advised to select a fixed number. For instance, selecting a depth of 4 may mean that the program takes too long to calculate the best move in certain dense positions. In other positions, however, and especially during the endgame when fewer moves are available, the program may have been able to calculate to a much greater depth almost instantaneously.

A solution to this problem is known as iterative deepening. This strategy simply performs a search to a depth of 1. Having completed this 1-ply search, it performs another search to a depth of 2, and then one of 3, and so on. When the time allocated to the search has expired, the results of the last complete search are returned.

While it may seem that much work is being wasted on shallow depths that are never used, the number of nodes evaluated using the minimax algorithm is in fact $b^d + 2(b^{d-1}) + \ldots + (d-1)b^2 + (d)b = O(b^d)$ [Russell and Norvig 2010]. Thus the number of nodes visited is of the same order of magnitude as one iteration of the minimax algorithm for a search to the same depth. Furthermore, the results of shallower searches can be used to order moves so as to increase the number of pruned nodes in alpha-beta pruning.

2.3 Monte Carlo Tree Search

In the domain of perfect-information games, the techniques discussed in the previous section were, for many decades, almost *de rigueur*. While they found much success in chess and checkers, their correlation to the game tree size, coupled with the difficulty in formulating a competitive evaluation function [Wang and Gelly 2007], make them deeply unsuited to Go. Add to this the long-term influence of moves[5], and it is unsurprising that these traditional Go programs were only able to compete at an amateur level against humans [Gelly *et al.* 2012].

However, a paradigm shift occurred with the emergence of Monte Carlo Tree Search (MCTS) [Coulom 2006]. MCTS makes use of the same game tree as search algorithms discussed in the previous section. Instead of using an evaluation function to calculate the values of nodes, it instead estimates the values of both leaf and interior nodes using Monte Carlo simulation. This is particularly useful to Go, since it negates the need for a strong evaluation function. This means that the algorithm is in fact domain-independent, and is thus applicable to a variety of fields.[6]

Another major advantage of MCTS is its directed nature, which allows it to focus on seemingly more promising regions and avoids wasting time on suboptimal lines of action. This allows the algorithm to build its tree to a great depth, which is advantageous in domains with extremely large state-spaces.

[5] Section A.3 presents a game in which the strength of a move is only revealed many moves into the future.

[6] Browne *et al.* [2012] provide a broad survey of applications.

The game tree used by MCTS differs from a normal game tree in that each node contains at least two additional pieces of information: the current value of the node and the number of times the node has been visited by the algorithm. When MCTS terminates, these values are used to decide on the final move selection. Several criteria, such as selecting the child of the root with the highest visit count, exist for doing so [Chaslot 2010].

The algorithm is fully described by the four phases illustrated in Figure 2.3, each of which is described below. Note that the description of the algorithm and the details that follow are relevant to the "vanilla" MCTS algorithm, as opposed to the numerous variants thereof.

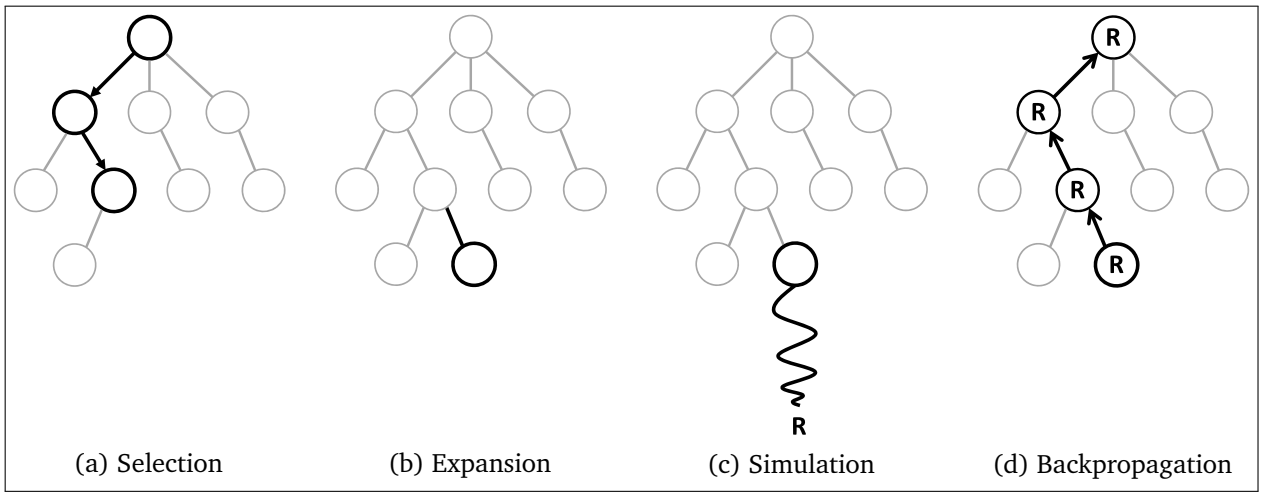


Figure 2.3: Phases of the Monte Carlo tree search algorithm. The game tree, rooted at the current position, is grown by repeating the above four phases.

2.3.1 Selection

The first phase of MCTS is the selection phase, in which the algorithm descends the tree recursively from the root until either a leaf or unexplored node is reached. The way in which this node is selected is known as the *tree policy*. This phase is of great importance, since it is responsible for balancing the trade-off between *exploration and exploitation*.

This trade-off is a problem central to reinforcement learning. Exploitation of actions currently believed to be optimal may result in large short-terms gains, but there may be superior actions that have yet to be discovered. Selecting suboptimal actions in order to learn more about the agent's environment (exploration) will result in smaller immediate rewards, but may result in an improvement to the agent's policy and, ultimately, greater returns.

This dilemma has been thoroughly analysed using a class of sequential decision problems known as *multi-armed bandit* problems [Robbins 1952]. In this problem, a player is required to choose amongst K arms of a multi-armed bandit slot machine in order to maximise his cumulative reward over a number of actions. The underlying reward distributions of each arm are stationary but

unknown, and rewards must therefore be estimated using past observations.[7] Many approaches to the selection phase of MCTS consider it a multi-armed bandit problem (with the child nodes in place of the bandit's arms) and act accordingly. The task is made more difficult by the fact that the reward distribution is non-stationary in the MDP setting.

The effectiveness of a policy for multi-armed bandit problems is usually quantified by its *cumulative regret* — the difference between the optimal and actual rewards received after n simulations. One policy that exhibits a regret growth of $O(\log n)$ is *UCB1* [Auer *et al.* 2002], which advocates playing arm j that maximises

$$\overline{X}_j + \sqrt{\frac{2 \ln n}{n_j}} \,, \tag{2.14}$$

where $\overline{X}_j$ is the average reward of arm j, n_j is the number of times arm j was played and n is the total number of plays.

Though there are many solutions to the multi-armed bandit problem, UCB1 is of great importance to MCTS as it forms the basis for the most well-known MCTS variant, Upper Confidence bounds applied to Trees (UCT) [Kocsis and Szepesvári 2006]. The UCT algorithm uses Bound (2.14) as its tree policy. In particular, in the selection phase a child node j is selected to maximise

$$\overline{X}_j + 2C_p \sqrt{\frac{\ln n}{n_j}} \,, \tag{2.15}$$

where $\overline{X}_j$ is the average return from child j, n is the number of times the current node has been visited, n_j is the number of visits to child j, and C_p is a constant. The right-hand term encourages exploration of less-visited nodes (and can be adjusted using the parameter C_p), while the left-hand term represents the child node's estimated value, which encourages exploitation of high-value nodes. It can be shown that after a a period of time N_0, UCT will follow only the optimal branch, yielding an asymptotic regret of $O(N_0 + \log n)$. However, Coquelin and Munos [2007] show that N_0 may be long, and prove that UCT's worst-case regret for a tree of depth D is $\Omega(\exp(\cdots \exp(1) \cdots))$ — that is, $D - 1$ composed exponential functions.

Perhaps the most attractive aspect of UCT is that is has been shown to be consistent: that is, the probability of selecting the incorrect action converges to zero in the limit [Kocsis *et al.* 2006]. Thus, given enough time and memory, UCT will always return the optimal action. Figure 2.4 demonstrates the performance of UCT in a simple domain, and illustrates the exploration-exploitation dilemma.

[7] The problem can be solved by computing the *Gittins index* [Jones and Gittins 1972]. However, this method does not appear to be computationally tractable, nor generalise to full reinforcement learning problems [Sutton and Barto 1998].

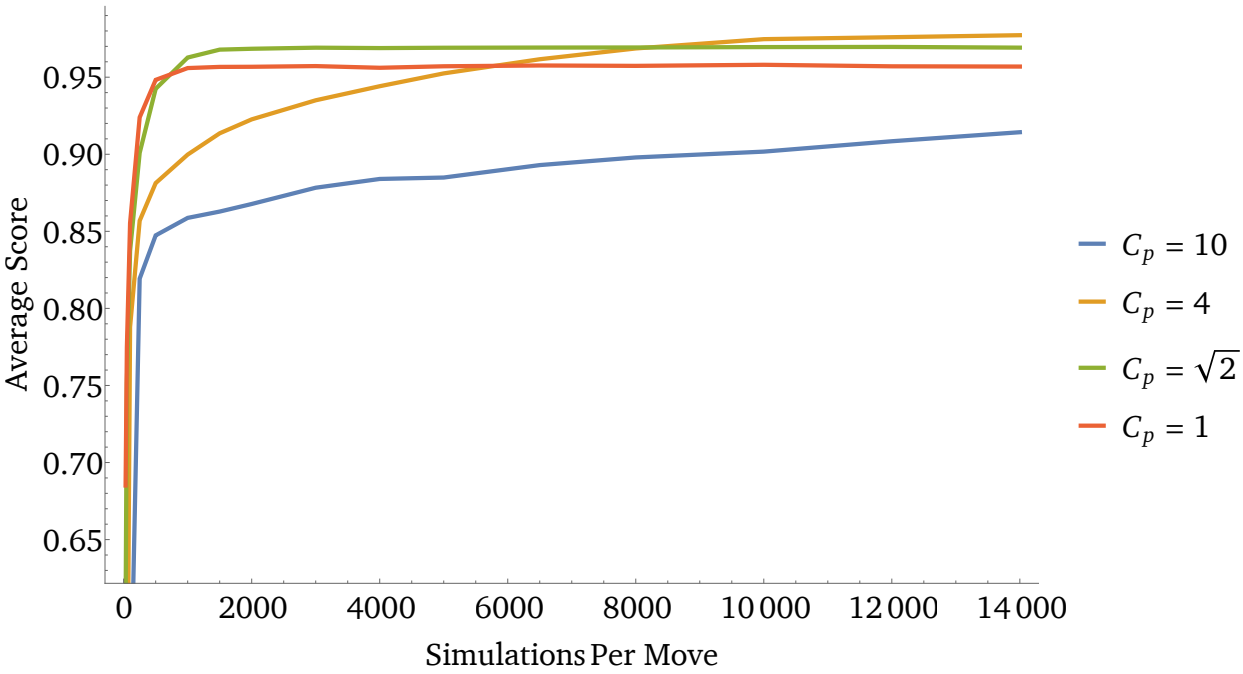


Figure 2.4: An illustration of exploration-exploitation dilemma and the effect of the exploration parameter. The UCT algorithm is applied to a game tree with both a height and branching factor of 5, and rewards in the range $[0, 1]$ randomly distributed at the leaves. Plotted above are the results for different exploration parameters, averaged over 10 000 instances of the domain. The graph illustrates that for small values of C_p, UCT stops exploring early on, achieving a higher score in the short term but ultimately plateauing too early. Larger values of C_p require more simulations, but outperform the more exploitive strategy in the long run. However, too great a value (e.g. $C_p = 10$) means that the nodes' estimates take a long time to converge (both in theory and practice), which can result in suboptimal performance as the agent explores too often. Thus a balance between exploring and exploiting current knowledge is required.

2.3.2 Expansion

Once a frontier node[8] has been selected, a child node not already present in the tree is added. Some implementations expand the full set of children, although the difference between the two approaches appears to be negligible [Browne *et al.* 2012], save for memory constraints and ease of use in different domains. In most cases, expanding a single node is sufficient.

[8] A frontier node is one where the number of child nodes is less than the number of possible children. In other words, the node has not yet been fully expanded.

2.3.3 Simulation

The next phase of the algorithm performs a simulation (also known as a *rollout* or *playout*) from the node expanded in the previous phase until a terminal state is reached. This is done according to the algorithm's *simulation policy*. In the simplest case, the simulation policy selects actions uniformly randomly (*light playouts*). However, other policies can be used in which actions are biased (*heavy playouts*) or even completely deterministic — this often depends on the domain in question. Section 2.5 provides a discussion on methods for learning biased rollout policies, which improve over a purely random strategy.

2.3.4 Backpropagation

Finally, the result of the simulation is propagated back through the tree to the root node, updating the value and visit count of each node along the way. The value propagated from the outcome of the simulation depends on the domain. In Go, it is simply the result of the game: 1 and 0 for a win and loss respectively and $\frac{1}{2}$ for a draw.

2.4 Enhancements to MCTS

Many enhancements applicable to every phase of the basic MCTS algorithm have been proposed. A subset of these, selected for their importance or relevance to this research topic, is presented below.[9]

2.4.1 Rapid Action Value Estimation

A major enhancement to MCTS in computer Go is *Rapid Action Value Estimation* (RAVE). As Domshlak and Feldman [2013] observe, improvements in Go owe more to the introduction of RAVE than it does to UCT.

RAVE itself is based on a history heuristic known as *All-Moves-As-First* (AMAF), which treats actions played during the simulation phase as though they were played during the selection step. This allows for the sharing of information between subtrees. MCTS ordinarily updates only those nodes selected by the tree policy with the playout result, whereas AMAF additionally updates any sibling nodes containing an action that was executed during the playout. The value of action a in state s is thus updated whenever a is encountered during a simulation, even if a was not actually played in position s. More information about the value of actions can be computed in this manner without the need for more iterations. However, AMAF makes the assumption that the order of moves is irrelevant, which is not always the case. Thus AMAF provides only a rough estimation of the value of an action.

[9] For a comprehensive summary of MCTS enhancements, see Browne *et al.* [2012].

The α-AMAF heuristic expands on this by combining AMAF's fast but inaccurate estimation with Monte Carlo's slow but accurate evaluation. The value of a node is a combination of the AMAF score A and Monte Carlo score M:

$$\alpha A + (1 - \alpha)M, \tag{2.16}$$

where α is a fixed constant.

RAVE is similar to α-AMAF in that it combines AMAF with Monte Carlo. However, instead of a fixed parameter α, RAVE uses a decreasing schedule. Given a user-specified constant V, the weighting parameter α for each node is calculated as

$$\alpha(s) = \max\left\{0, \frac{V - v(s)}{V}\right\}, \tag{2.17}$$

where $v(s)$ is the number of visits to node s [Helmbold and Parker-Wood 2009].

Other formulae exist in variants of RAVE[10], but all conform to the same idea: at the beginning of the search, when only few simulations have been performed, $\alpha \to 1$ and the AMAF score is weighted more highly. However, after many simulations, $\alpha \to 0$ and the more accurate Monte Carlo score has a greater effect.

2.4.2 Progressive Bias and Search Seeding

Two common methods of including domain-specific information in MCTS are *progressive bias* and *search seeding*. Progressive bias [Chaslot *et al.* 2008] incorporates a heuristic evaluation function into the selection phase of MCTS. A term of the form

$$f(n_i) = \frac{H_i}{n_i + 1} \tag{2.18}$$

is added to the selection formula of MCTS, where H_i is the heuristic value of node i according to some evaluation function, and n_i is the visit count of node i. The influence of this term decreases with the number of visits to the node. Progressive bias attempts to offset the use of a computationally expensive (in most cases) evaluation function by improving the accuracy of the tree policy.

Similarly, heuristic knowledge can be inserted into MCTS in a procedure known as search seeding. Normally, when a node is added to the tree, its value and visit count are set to 0. Search seeding instead initialises these values according to the heuristic value of the node — Gelly and Silver [2007] use a learned evaluation function to initialise nodes' statistics.

[10] Gelly and Silver [2011] found success using an equivalence parameter k and setting $\alpha(s) = \sqrt{\frac{k}{3v(s) + k}}$.

2.5 Learning Feature Weights

Enhancing the rollout policy has proven to be difficult, since small changes appear to have a major effect on the overall performance [Rimmel *et al.* 2010]. Thus incorporating expertise or domain-specific knowledge into the simulation phase of MCTS is a challenging problem, although successful methods do exist. This is often achieved by specifying a feature-space for the domain, learning the features' corresponding weights and using the result to either seed nodes (as in Section 2.4.2) or construct a softmax simulation policy. Some approaches to learning feature weights are examined below.

2.5.1 Simulation Balancing

Instead of constructing an objectively strong softmax policy using reinforcement learning and then transplanting it into the simulation phase of MCTS, Silver and Tesauro [2009] propose the idea of *simulation balancing*. This concept is applicable to two-player games and posits that the simulation policy need not be objectively strong as long as it is *balanced*. This means that during the simulation, any errors made by one player tend to be cancelled out by his opponent.

To formalise the concept of balance, let $\delta_t^{(k)} = V^*(s_{t+k}) - V^*(s_t)$ be the error incurred by a move played at time t after k future moves, where $V^*(s)$ is the minimax value of state s. Letting $\rho(s)$ be the distribution of states evaluated during a Monte Carlo search, the k-step imbalance is given by

$$B_k(\boldsymbol{\theta}) = \mathbb{E}_\rho \left[(\mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\delta_t^{(k)} \mid s_t = s])^2 \right]. \tag{2.19}$$

Of interest is the 2-step imbalance which suggests that mistakes are allowed if they are immediately cancelled out by the opponent's next move, and the full imbalance ($k = \infty$) which allows for mistakes as long as they are cancelled out by the end of the simulation. Simulation balancing then learns the weights that minimise $B_k(\boldsymbol{\theta})$, and thus a "fair" policy with small imbalance.

Simulation balancing provided promising results on small 5×5 and 6×6 Go boards, illustrating that weak policies do not necessarily imply inferior MCTS agents. Huang *et al.* [2011] were later able to extend the idea to 19×19 boards.

2.5.2 Supervised Learning

A more recent approach for learning feature weights $\boldsymbol{\gamma}$ is that of *supervised learning*, where the feature weights are inferred from labelled training data in an attempt to predict the action an expert would select in a given situation. In the context of Go, samples are collected from expert games, with each sample consisting of a binary set of state-action features $\boldsymbol{\phi}(s, a) \in \{0, 1\}^N$, and a binary label that indicates whether that sample corresponds to the expert's choice.

There are numerous ways of inferring the feature weights from the above-labelled samples. One method that has recently found favour in Go is to train *deep neural networks* to predict an expert's moves [Clark and Storkey 2014; Silver *et al.* 2016]. Another, less resource-intensive technique,

is to make use of the Bradley-Terry (BT) model [Hunter 2004; Coulom 2007]. The BT model predicts the outcome of a competition between two individuals i and j, each of which has assigned strengths γ_i and γ_j. The model specifies the probability of i winning as

$$p(i \text{ wins}) = \frac{\gamma_i}{\gamma_i \gamma_j}. \tag{2.20}$$

This can be further extended to account for teams of players involved in a competition against other teams. For example,

$$p((i,j) \text{ beats } (j,k) \text{ and } (i,l,m)) = \frac{\gamma_i \gamma_j}{\gamma_i \gamma_j + \gamma_j \gamma_k + \gamma_i \gamma_l \gamma_m}.$$

The BT model can be leveraged to learn feature weights by treating each state-action sample as a team whose members are its set of *active*[11] features. The strength of the members corresponds to the weights of the individual features. A state-action sample is considered to have "won" if the action was selected by the expert. The BT model can be used to learn the strengths of the individual members, which simply represent the individual feature weights themselves. One limitation of this model is that it considers the features to be independent, and does not capture the effect of the interaction between features.

There are a number of techniques for estimating feature weights.[12] Coulom [2007] calculates the *maximum a posteriori* probability estimate using the *Maximization-Minorization* formula, while Weng and Lin [2011] represent each weight as a Gaussian distribution, updating them using lightweight approximations when a new result is observed. This latter method is termed *Bayesian Approximation for Online Ranking* (BAR), and approximates the integrals that appear when performing Bayesian inference by applying Woodrofe-Stein's identity. BAR can be used with any model that can be written as the product of factors, leading to the likelihood function

$$\mathcal{L} = \prod_{i=1}^{k} \prod_{q=i+1}^{k} P(\text{outcome between team } i \text{ and team } q),$$

where k is the number of teams.

Presented below is one further method — *Latent Factor Ranking* (LFR) — which addresses the shortcoming of the BT model mentioned above, and which is used in a number of experiments in this research.

Latent Factor Ranking

Latent Factor Ranking [Wistuba and Schmidt-Thieme 2013] makes use of the *Factorization Machine* (FM) model [Rendle 2010] to learn not only the individual feature weights, but also the weights of interactions between all pairs of features. If there are m features, interactions between all pairs

[11] A binary feature ϕ_i is said to be active for the state-action pair (s,a) if the i-th element of $\boldsymbol{\phi}(s,a)$ is 1.
[12] The full details of each method are available in the cited papers.

would require a $\mathbb{R}^{m \times m}$ matrix. Using an FM, LFR is able to factorise this sparse matrix, reducing it to $\mathbb{R}^{k \times m}, k \ll m$.

Let $\phi \in \mathbb{R}^m$ represent the binary features of a state-action pair (s, a), and let $\mathcal{I}(\phi) = \{i \mid \phi_i = 1\}$ be the set of active features. For a given state s_j, with possible actions $a_1, \ldots, a_n$, the training set of action decisions is given by

$$\mathcal{D}_j = \left\{\phi^{(1)} = \phi(s_j, a_1), \ldots, \phi^{(n)} = \phi(s_j, a_n)\right\}. \tag{2.21}$$

Each feature ϕ_i has corresponding weight θ_i, as well as vector $\mathbf{v}_i \in \mathbb{R}^k$ which is used to calculate the weight of interaction between features. For instance, the weight assigned to the interaction of features ϕ_i and ϕ_j is given by $\frac{1}{2}\mathbf{v}_i^T \mathbf{v}_j$.

Each state-action pair in the training data is labelled according to whether it was selected by the expert. That is,

$$y(\phi) = \begin{cases} 1 & \text{if } \phi \text{ contains the chosen action in its corresponding state} \\ 0 & \text{otherwise,} \end{cases} \tag{2.22}$$

while the estimate of this label, according to the FM model, is given by

$$\hat{y}(\phi) = w + \sum_{i \in \mathcal{I}(\phi)} \left(\theta_i + \sum_{i \in \mathcal{I}(\phi), i \neq j} \frac{1}{2} \mathbf{v}_i^T \mathbf{v}_j \right), \tag{2.23}$$

where w is some bias term.

In order to learn the weights, the gradient of $\hat{y}$ is required:

$$\frac{\delta}{\delta \xi} \hat{y}(\phi) = \begin{cases} 1 & \text{if } \xi = w \\ 1 & \text{if } \xi = \theta_i \text{ and } i \in \mathcal{I}(\phi) \\ \sum_{j \in \mathcal{I}(\phi) \setminus \{i\}} \mathbf{v}_{j,f} & \text{if } \xi = \mathbf{v}_{i,f} \text{ and } i \in \mathcal{I}(\phi) \\ 0 & \text{otherwise.} \end{cases} \tag{2.24}$$

LFR uses stochastic gradient descent with L2-regularisation to estimate the weight parameters in an attempt to minimise the error in the ranking of moves. Specifically, assume that $\phi^{(1)}$ represents the expert's selection. The weights corresponding to any other $\phi^{(x)}$ are then updated only when $\hat{y}(\phi^{(x)}) \geq \hat{y}(\phi^{(1)})$. The full algorithm is presented in Algorithm 1.

function LFR($\mathcal{D}$) ▷ $\mathcal{D}$ is the training set consisting of move decisions $\mathcal{D}_0, \ldots \mathcal{D}_t$
  $\theta_i \leftarrow 0, w \leftarrow 0, v_{i,f} \sim \mathcal{N}(0, 0.1)$
  while not converged **do**
    for each $\mathcal{D}_j \in \mathcal{D}$ **do**
      for each $\phi \in \mathcal{D}_j$ **do**
        if $\hat{y}(\phi) \geq \hat{y}(\phi^{(1)})$ **then**
          $\Delta y \leftarrow \hat{y}(\phi) - y(\phi)$
          $w \leftarrow w - \alpha \cdot \Delta y$
          for each $i \in \mathcal{I}(\phi)$ **do**
            $\theta_i \leftarrow \theta_i - \alpha(\Delta y + \lambda_\theta \theta_i)$
            for $f = 1 \rightarrow k$ **do**
              $v_{i,f} \leftarrow v_{i,f} - \alpha \left(\Delta y \frac{\delta}{\delta v_{i,f}} \hat{y}(\phi) + \lambda_v v_{i,f} \right)$
            end for
          end for
        end if
      end for
    end for
  end while
end function

Algorithm 1: Latent Factor Ranking. The algorithm requires regularisation parameters λ_θ and λ_v, learning rate α, and dimension $k \ll m$, all of which are provided by the user.

2.6 Conclusion

The topic of this research is that of understanding the simulation phase of MCTS in the context of MDPs. This chapter therefore attempted to provide background on these core topics: MDPs, MCTS and its simulation phase.

Reinforcement learning and its accompanying notation were presented, providing a formal context which can be applied to various domains. This included Monte Carlo methods, temporal difference learning and functional approximation, all of which are central to MCTS and its improvements.

Traditional approaches to game-playing were discussed in Section 2.2, providing contrast to the newer paradigm of Monte Carlo Tree Search (Section 2.3). Each of the four phases of MCTS was discussed, with particular emphasis placed on the most popular of MCTS variants — UCT. Finally, certain enhancements to MCTS, and especially those concerned with learning biased rollouts, were presented.

Chapter 3

Insufficiency of Current Explanations

As the groundbreaking results of Silver *et al.* [2016] in Go will undoubtedly bring MCTS methods into sharp focus, developing a better understanding of every facet of the algorithm is important. While much theory has been developed regarding the tree policy (albeit mainly in a bandit setting), a rigorous analysis of the simulation phase in general MDPs is conspicuously absent from the literature. This chapter presents some results that are indicative of the difficulty in reasoning about simulation policies, as well as the unexpected results that can ensue, all of which are presented to illustrate the pressing need for a greater understanding of MCTS.

3.1 The Go Domain

To begin, a series of experiments on smaller 9×9 Go[1] boards is performed, the aim of which is to investigate the performance of MCTS under different simulation policies. FUEGO [Enzenberger *et al.* 2010], an open-source collection of C++ libraries that contains a UCT-based Go agent, is used for these experiments, with its rollout policy modified in a number of ways. In order to be consistent in all the experiments conducted throughout this thesis, a vanilla UCT implementation of FUEGO is selected. Thus additional enhancements such as RAVE are disabled, and the exploration constant set to 0.7. The minimax-derived program GNUGO is also used to act as a control agent.

In order to learn weights for FUEGO's built-in binary features (2154 features in total), a set of training data is required. Samples to be used by a supervised learning method are extracted from high-level games downloaded from the KGS Go Server,[2] with move decisions randomly selected from each game to form a training set of 100 000 samples.

With this data in hand, weights are learned using LFR and BAR (refer Section 2.5.2). Let $\boldsymbol{\theta}_L$ and $\boldsymbol{\theta}_B$ be the weights learned by these two methods respectively. Softmax and deterministic policies (with a deterministic tie-breaking rule) parameterised by these weights are then constructed — the

[1] See Appendix A for a brief introduction to the game.
[2] https://www.gokgs.com/

softmax policy, denoted $\pi^{\sigma}_{\boldsymbol{\theta}}$, takes the form of Equation 2.12, while the deterministic policy $\pi^{D}_{\boldsymbol{\theta}}$ is given by

$$\pi^{D}_{\boldsymbol{\theta}}(s,a) = \begin{cases} 1 & \text{if } a = \text{argmax}_{a \in \mathcal{A}(s)}\, \boldsymbol{\phi}(s,a)^T \boldsymbol{\theta} \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

Along with the standard FUEGO and GNUGO agents, five additional UCT agents are constructed, each of which is based on FUEGO and differs only in its rollout policy as listed in Table 3.1.

Agent Name	Rollout Policy
GNUGO	—
FUEGO	Handcrafted policy[3]
RANDOM	Uniformly Random[4]
LFR MAX	$\pi^{D}_{\boldsymbol{\theta}_L}$
LFR SOFTMAX	$\pi^{\sigma}_{\boldsymbol{\theta}_L}$
BAR MAX	$\pi^{D}_{\boldsymbol{\theta}_B}$
BAR SOFTMAX	$\pi^{\sigma}_{\boldsymbol{\theta}_B}$

Table 3.1: A list of the various UCT Go agents used in this experiment.

Testing agents against one another over many games is computationally expensive. In order to run thousands of games in a reasonable amount of time, experiments are conducted in parallel using the University of the Witwatersrand's Hydra cluster, which runs Ubuntu 14.04.2 and consists of:

- 70 nodes
- 4 physical cores per node (Intel Core i7-4790 @ 3.60 GHz)
- 8 GB RAM per node.

3.1.1 Results

The performance of the agents that use the weights learned by BAR gives credence to the notion that strong rollout policies do not necessarily result in strong MCTS agents. First consider selecting an action to play from the rollout policies BAR SOFTMAX, BAR MAX and RANDOM, in order to

[3] FUEGO's rollout policy is similar to that of MOGO (Chapter 1) in that it prioritises certain moves, such as captures and pattern-matching moves, but plays randomly otherwise.

[4] The policy is not quite uniformly random, as one-point eye-filling moves are forbidden to ensure that the simulation finishes in a reasonable time.

measure their objective strength (Table 3.2a). Both BAR SOFTMAX[5] and BAR MAX were far superior to RANDOM, achieving a win rate of over 80%.

As Table 3.2b demonstrates, however, both learned policies are extremely undesirable as simulation policies of UCT, barely winning a game against uniformly random rollouts. This clearly indicates that while the feature weights result in strong stand-alone policies, they are effectively useless as a simulation policy.

Agent	**Opponent**	
	$\pi^{\sigma}_{\boldsymbol{\theta}_B}$	Random
$\pi^{D}_{\boldsymbol{\theta}_B}$	47 ± 5	83 ± 3.8
$\pi^{\sigma}_{\boldsymbol{\theta}_B}$	—	81 ± 3.9

(a) Win percentage of agents, together with one standard error, using BAR weights when actions were sampled directly from the rollout policy, averaged over 100 games.

Agent	**Opponent**	
	BAR SOFTMAX	RANDOM
BAR MAX	36 ± 4.8	1 ± 1.0
BAR SOFTMAX	—	4 ± 2.0

(b) Win percentage of UCT agents (5000 simulations per move), together with one standard error, using BAR weights in their rollout policies, averaged over 100 games.

Table 3.2: Performance of BAR weights in 9×9 Go.

With the BAR weights producing inferior UCT agents, LFR weights become the sole focus. A round-robin tournament — facilitated by the GOGUI-TWOGTP program from the GOGUI[6] suite — is contested by the agents, with each agent playing one another a total of 100 games (50 as white and 50 as black). The number of simulations allowed per move is fixed, and the tournament is repeated for increasing numbers of simulations. Using the outcome of every game, the Elo[7] rating for each player is calculated using BAYESELO [Coulom 2008]. GNUGO's rating is fixed at 1800, as is standard practice. The results are summarised in Figure 3.1 below.

[5]The magnitude of the weights produced by BAR was quite large, meaning that the softmax policy was often very similar to the deterministic one. Scaling the weights through a logistic function did not change the results significantly.

[6]http://gogui.sourceforge.net/

[7]The Elo rating is a system used to calculate the *relative* strength of competitors in adversarial games.

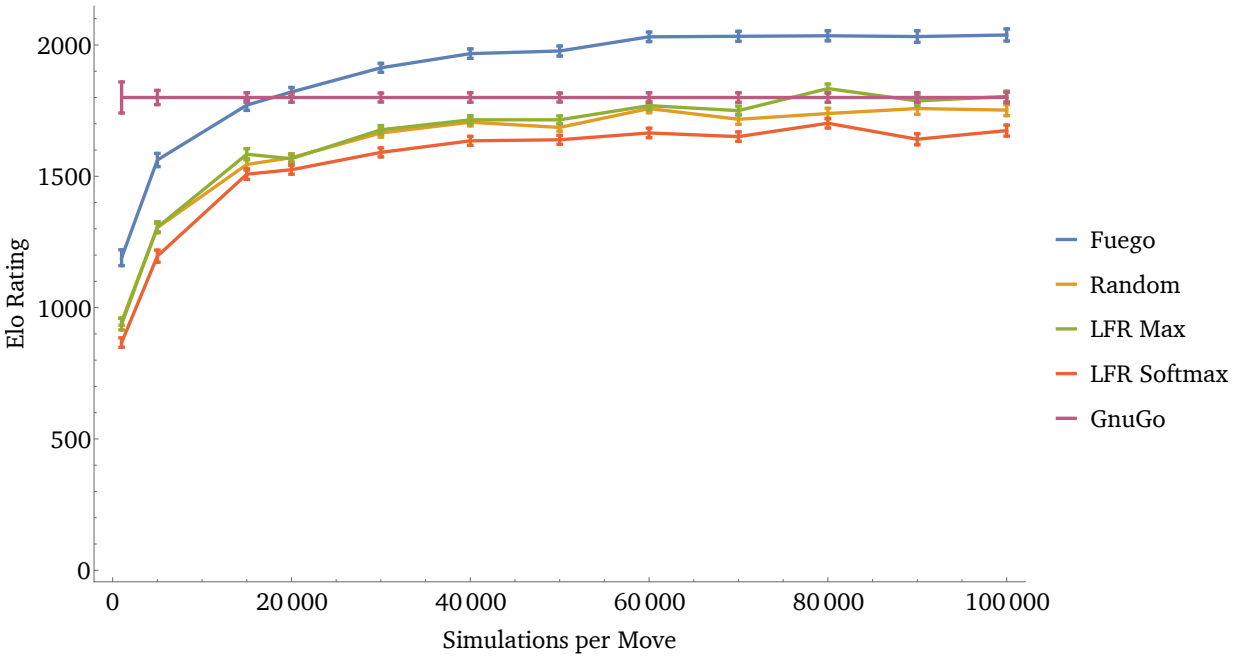


Figure 3.1: Elo Ratings of UCT agents (with standard error bars) in 9×9 Go as a function of the number of simulations allowed per move.

The results provide a very interesting and counterintuitive result — the best-performing agent, aside from FUEGO, was a UCT agent with a *deterministic* rollout policy. Thus the accepted, if overly-simplistic, wisdom of having sufficient stochasticity in the rollout phase does not always hold. There are clearly a number of latent and conflating factors at play in this example, which provides an illustration of the difficulty in reasoning about the simulation phase.

3.2 The INFINITE MARIO Domain

The next domain considered is that of INFINITE MARIO [Karakovskiy and Togelius 2012], an open-source clone of the popular SUPER MARIO BROS. video game (Figure 3.2 provides a snapshot). The aim of the game is for the agent (*Mario*) to reach the end of the level by traversing some required distance to the right without dying — Mario dies when he collides with an enemy character or falls down a hole. Mario has five independent binary actions available to him (`LEFT`, `RIGHT`, `DUCK`, `JUMP`, `FIRE/SPRINT`), any of which can be used simultaneously, resulting in a total of 32 different actions. Furthermore, Mario is also able to exist in three states:

- *Small* — in this state any contact with an enemy results in Mario's death.
- *Large* — any contact with an enemy transforms Mario into the small state.
- *Fire* — Mario is able to shoot fireballs at enemies, which destroy them on contact. As in the large state, contacting an enemy results in a small-state Mario.

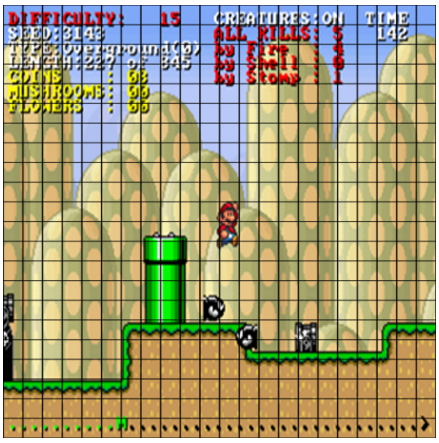


Figure 3.2: A single frame of the INFINITE MARIO domain. Instead of treating the game as a continuous environment, a 22×22 grid is extracted from the game engine at each time step, with Mario positioned at the centre. Each grid square contains either Mario, an enemy, a fireball, the terrain or empty space. Other elements of the game, such as coins and mushrooms, are ignored by the agent.

As in Section 3.1, feature weights are learned and used to bias the rollout policy of a UCT agent. A number of binary features (similar to those used by Curran *et al.* [2015]) are defined, and their weights learned by LFR. Training data for performing LFR is acquired by recording the actions taken by Robin Baumgarten's A* agent, winner of the 2009 Mario AI competition [Togelius *et al.* 2010]. The agent acts as an oracle, with its actions recorded over 70 different levels, resulting in 68 539 samples. Randomly sampling 40 000 action decisions creates the training data that is used to train weights for the features, as defined in Table 3.3.

is Mario small?	is Mario large?	is Mario fire?
obstacle in left 4 squares? (×4)	is hole left?	is Mario in hole?
obstacle in right 4 squares? (×4)	is hole right?	is Mario on ground?
enemy within 1 square? (×8)	can Mario jump?	gap 2-4 squares left?
enemy within 2 – 4 squares? (×8)[8]	gap 2-4 squares right?	expert pressed `LEFT`?
expert pressed `RIGHT`?	expert pressed `DUCK`?	expert pressed `JUMP`?
expert pressed `FIRE/SPRINT`?		

Table 3.3: The set of binary features for the INFINITE MARIO domain.

[8]There are 8 features, one for each direction. If there is at least one enemy in a given direction at a range of 2 – 4 squares, the feature is set to 1.

Aside from being a single-agent environment, one of the fundamental differences in this domain (when compared with Go) is the manner in which the simulations need to occur. Whereas in Go any simulation policy is able to reach a terminal state and return the actual outcome (win/loss), the dynamics of INFINITE MARIO mean that simulations are almost never able to reach a *winning* terminal state — that is, any kind of random walk is unlikely to reach the end of the level (and those that do imply that the agent was already very near the end to begin with).

Thus a different domain-specific reward function needs to be developed to apply UCT to the problem. This is achieved by first truncating the number of sequential actions in the rollout to 6. The agent receives a reward of 1 if it completes the level, and a reward of 0 if Mario dies or is shrunk to a lesser state at any point during the rollout. If neither of these events occurs, then the reward returned is the distance Mario managed to traverse to the right at the end of the rollout, divided by the maximum distance he could have possibly moved. This incentivises the agent to move to the right, as well as bounding all rewards in the range $[0, 1]$. UCT can then be applied to the problem as normal.

3.2.1 Results

Once again, UCT agents are parameterised by greedy, softmax and random rollout policies, and their performances in the domain compared. Each agent was tested on the same 100 levels,[9] with its performance measured by the average distance covered. The results are presented in Figure 3.3 below.

Unlike the results from Go, here a deterministic rollout policy does extremely poorly. Furthermore, biased rollouts confer little, if any, advantage over uniformly random rollouts, despite the fact that executing a random walk in the INFINITE MARIO domain almost always results in death. A point of interest is the decrease in the performance of the deterministic policy when the number of simulations is increased. One explanation is that the greedy policy overestimates the values of all nodes, causing the agent to act recklessly. For fewer simulations, this is somewhat mitigated by the randomness due to the expansion phase. When a sufficient number of simulations is used, however, this no longer holds true and the agent performs poorly.

Additionally, as an approach MCTS is not well suited to the domain. For reference, the expert agent achieves a score of 26870 and is able to do so in real time (24 action decisions per second). There is clearly then some fundamental or latent difference between this domain (where MCTS does poorly) and Go (where MCTS represents state-of-the-art performance). This is further borne out by games such as chess (a two-player, perfect-information, turn-based game like Go) where MCTS is again ill-suited to the problem [Ramanujan *et al.* 2011].

[9] The framework generates levels procedurally based on a random seed and a difficulty setting. For these experiments, an extreme level of difficulty (15) was used in an attempt to provide maximum disambiguation amongst the agents.

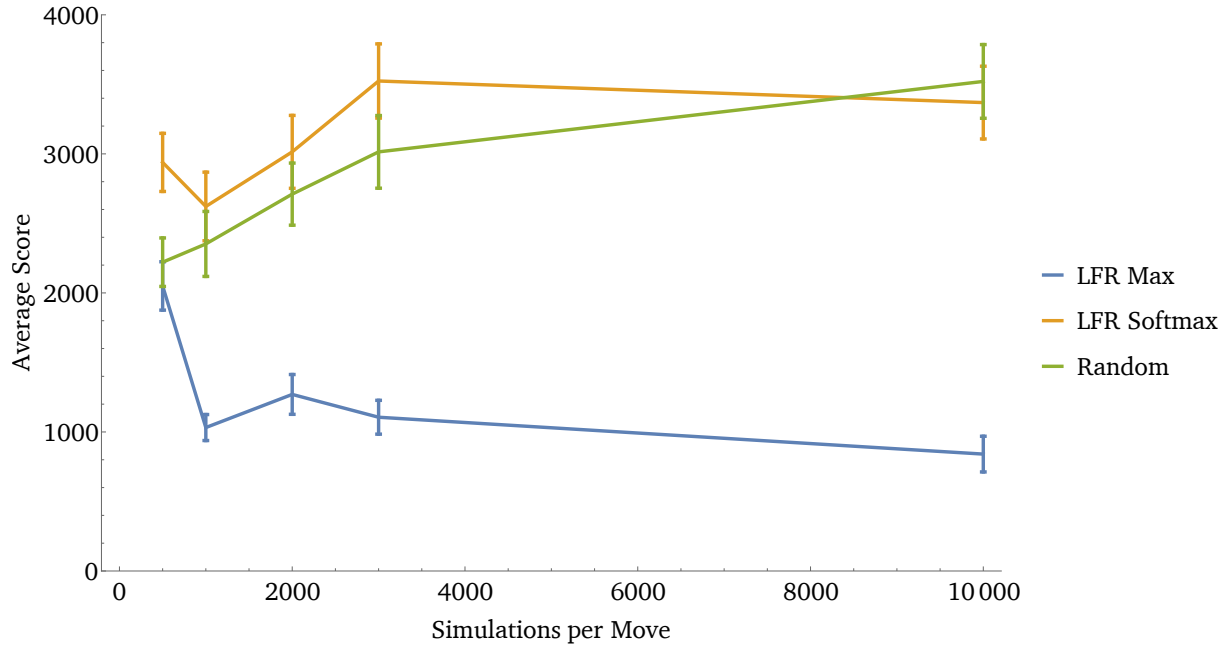


Figure 3.3: Average distance covered for the various UCT agents in INFINITE MARIO, averaged over 100 levels.

3.3 Conclusion

This chapter illustrated some of the difficulties associated with MCTS. Not only is it difficult to ascertain when MCTS is an appropriate choice of algorithm, but determining the effectiveness of simulation policies is also a challenge. Demonstrated, too, was the importance of learning the correct feature weights — both BAR and LFR were trained on the same feature space, but LFR far outperformed BAR. Ultimately, it is clear that there is a desperate need for more insight into the algorithm, its efficacy and the effect of its rollout phase.

Chapter 4

Research Methodology

The previous chapter provided an illustration of the difficulties in working with MCTS. In particular, two lines of enquiry presented themselves — the viability of MCTS as an algorithm and the effect of biased rollouts in a given domain. This chapter expands on these questions and presents a set of hypotheses to be tested experimentally in a number of domains.

4.1 Research Questions

The primary aim of this research is to provide greater understanding of the MCTS algorithm, the effect of its simulation policies and its applicability to arbitrary environments. As briefly mentioned in Chapter 1, work by Gelly and Silver [2007] has indicated that using objectively strong policies in the simulation phase is no guarantee of improving MCTS overall. Furthermore, the previous chapter's results demonstrated that MCTS is not readily applicable to any and all domains. From this, the following research questions are proposed:

Research Question 1: What characteristics or properties of a domain have the greatest bearing on the performance of MCTS?

Research Question 2: Under what conditions do uniformly random rollout policies result in strong MCTS agents?

Research Question 3: Under what conditions are biased or learned rollouts more effective and when do they result in poorer performance?

4.2 Research Hypothesis

There is some evidence to suggest that, in terms of its effect on the performance of MCTS, a key characteristic of a domain is the smoothness of its underlying value function. The phenomenon of game tree *pathology* in minimax searches [Nau 1982], as well as work by Ramanujan *et al.* [2012], both advance the notion of *trap states*, which occur when the value of two sibling nodes differs greatly. Furthermore, in the context of $\mathcal{X}$-armed bandits (where $\mathcal{X}$ is some measurable

space), UCT can be seen as a specific instance of the Hierarchical Optimistic Optimisation (HOO) algorithm. HOO attempts to find the global maximum of the expected payoff function using MCTS. Its selection policy is similar to UCB1, but contains an additional term that depends on the smoothness of the function. For an infinitely smooth function, this term goes to 0 and the algorithm degenerates to UCT [Bubeck *et al.* 2011].

This suggests that MCTS is geared towards smooth domains, where smoothness is defined in terms of the value function. One notion that can be employed is that of Lipschitz continuity, which limits the rate of change of a function. Informally, if a value function is Lipschitz continuous, then states close together should have similar values. More formally:

Definition 1. *A value function V is M-Lipschitz continuous if* $\forall s, t \in \mathcal{S}$*,*

$$|V(s) - V(t)| \leq Md(s, t),$$

where $M \geq 0$ is a constant, $d(s, t) = \|k(s) - k(t)\|$ and k is a mapping from state space to some vector space.

Given that it appears that UCT caters for smooth domains, the following hypotheses arise:

Research Hypothesis 1: The strength of an MCTS algorithm is positively correlated with the smoothness of the domain. That is, MCTS performs well in smooth domains, but struggles in non-smooth environments.

Research Hypothesis 2: An improperly biased simulation policy can lead to suboptimal play. The likelihood of this occurring is especially high in non-smooth domains and for low-variance rollouts.

4.3 Methodology

It is often very difficult to ascertain the success (or lack thereof) of a particular algorithm in some non-trivial domain. As an example, an improvement to the minimax algorithm proposed by Nau [1983] selects the correct action far more frequently than does minimax for the given class of problem. However, when the two methods compete against each other, the enhanced algorithm displays only minimal improvement, achieving win ratios of around 54%. The fact that the algorithm made correct decisions more often did not translate into a notable increase in win percentages. Thus the choice of success criterion can have a major influence on the conclusions drawn: in this case, one might conclude that the improvement to minimax is insignificant if the win percentage is considered alone.

This research takes the view that, ultimately, MCTS is an algorithm for selecting an available action at a given state. Thus, whether or not the correct action is selected will be the most common metric used in the experiments conducted. Naturally this will only hold where appropriate. For example, the problem of finding the global maximum of a function does not lend itself to this measure — in this case, a better measure may be the value MCTS perceives to be the maximum.

Another important decision is that of the MCTS variant. Since the focus of this research lies in the simulation phase, the MCTS algorithm is standardised across all experiments, with only the simulation policies changing. Unless otherwise stated, UCT is selected as the MCTS algorithm to

be used: being the most popular and well-known, any conclusions drawn stand to affect a large amount of existing work. Rewards in all domains are guaranteed to be in the range $[0, 1]$ so that the same exploration constant (set at a reasonable $C_p = 0.7$[1]) can be used throughout. Finally, the action returned by the algorithm is the edge that connects the root node to its most visited child.

4.4 Conclusion

This chapter laid out the two most important lines of enquiry of the research: when is MCTS appropriate and what is the effect of heavy playouts in a given domain? A definitive answer to these questions can assist in current and future work, providing insight into results which are often difficult to rationalise, as well as some indication as to whether a modification or improvement to MCTS is suitable for a given domain. A series of experiments designed to answer these questions is presented in the following two chapters.

[1] $C_p = 0.7$ is the default value for the UCT implementation of FUEGO when RAVE is not used.

Chapter 5

Value Function Smoothness

Section 4.2 provided some indication that the smoothness of the value function may play a large role in the performance of MCTS. This chapter describes a major advantage in assuming a certain class of value function smoothness. Although this advantage is shown to apply to a simple, non-UCT algorithm, it does suggest that smooth domains benefit sampling-based algorithms. This chapter also provides some insight into the manner in which UCT reacts to the smoothness (or non-smoothness) of the value function in a function optimisation task.

5.1 The Advantage of Smooth Domains

In this section, consider a large but finite MDP with a deterministic transition function. As briefly stated in Section 2.1.2, a well-known approach to learning the optimal value function in an MDP is that of *value iteration*. The algorithm updates the current state's value estimate by performing a one-step lookahead and selecting the maximum value that can be obtained thereafter.

Let $T : \mathbb{R}^N \to \mathbb{R}^N$ denote the *Bellman optimality operator* such that

$$V_{t+1}(s) = TV_t(s) = \max_{a \in \mathcal{A}(s)} \{R(s, a, s') + \gamma V_t(s')\}. \tag{5.1}$$

A well-known property of the Bellman optimality operator is that it is a contraction with respect to the L_∞ norm[1]: $\|TV - TV'\|_\infty \leq \gamma \|V - V'\|_\infty$ for any functions V and V' [Denardo 1967]. As a consequence of Banach's fixed-point theorem and T's contraction property, the value function will converge to the optimal value function V^* with repeated application of T.

For large domains, approaches such as linear function approximation (Section 2.1.5) are required to represent the state-space. In this case, an algorithm called *fitted value iteration* can be used to learn the weights of the function. Let the value function be given by $V(s) = \theta^T \phi(s)$, where ϕ is a feature mapping and θ are the weights to be learned.

[1] The infinity or maximum norm is defined as the maximum magnitude of the components of the input: $\|\mathbf{x}\|_\infty = \sup_i |x_i|$.

Algorithm 2 demonstrates fitted value iteration, which samples from a set of states $\sigma \subseteq \mathcal{S}$, applies the Bellman operator at these points, and then fits the weights to these values.

---

function FITTED VALUE ITERATION(ϕ, σ)
  $\theta \leftarrow 0$
  repeat
    for each $s^{(i)} \in \sigma$ **do**
      $y^{(i)} \leftarrow \max_{a \in \mathcal{A}} \left(R(s^{(i)}, a, s') + \gamma \theta^T \phi(s') \right)$
    end for
    $\theta' \leftarrow \operatorname{argmin}_{\theta} \max_{1 \leq i \leq |\sigma|} |\theta^T \phi(s^{(i)}) - y^{(i)}|$
    $\Delta_\theta \leftarrow \|\theta' - \theta\|_2$
    $\theta \leftarrow \theta'$
  until Δ_θ is small enough
  return θ
end function

Algorithm 2: The fitted value iteration algorithm. The procedure applies the Bellman backup to the subset of states, and then attempts to fit θ to the returns. The L_∞ norm is used here so as to ensure convergence. In practice, however, this is too difficult to calculate and the L_2 norm, which has no theoretical guarantees but often converges, is instead used. Note that the algorithm does not, in general, converge to the optimal value function.

Under the assumption of Lipschitz continuous value functions, it can be demonstrated that the error due to sampling only a subset of states is bounded, and decreases with an increase in the size of the set.

Let rewards in the MDP be in the range $[-R_{\max}, R_{\max}]$ and $0 \leq \gamma < 1$ be the discount factor. Denote T as the Bellman backup operator and $\hat{T}$ as a single iteration of the above algorithm, so that $V_{t+1} = \hat{T} V_t$. Let κ be an upper bound on the approximation error $|TV_t - \hat{T}V_t| = |TV_t - V_{t+1}|$ for all t, and let $\delta_\sigma = \sup_{s \in \mathcal{S}} \min_{s' \in \sigma} \|s - s'\|_1$ be the maximum distance from any state in $\mathcal{S}$ to its closest neighbour in σ. An assumption of locally Lipschitz value functions is made:

Assumption 1. *For any $s, s' \in \mathcal{S}$, if $\|s - s'\|_1 \leq \delta \Rightarrow |V(s) - V(s')| \leq \beta\delta, \quad \beta \geq 0$.*

The following theorem bounds the error between V_t and V^*:

Theorem 1. *For every $s \in \mathcal{S}$,*

$$|V^*(s) - V_t(s)| \leq \frac{2\beta}{1-\gamma}\delta_\sigma + \frac{2\gamma^t R_{max}}{1-\gamma} + \frac{\kappa}{1-\gamma}. \tag{5.2}$$

The above is a simpler variant of Theorem 2 due to Bai *et al.* [2010], and its proof follows a similar structure.

Proof (Theorem 1). Let $\epsilon_t = \max_{s \in \sigma} |V^*(s) - V_t(s)|$ be the largest error over the sample-space. For

any state $s \in \mathcal{S}$, let s' be the closest state in σ to s. By the triangle inequality,

$$|V^*(s) - V_t(s)| \leq |V^*(s) - V^*(s')| + |V^*(s') - V_t(s')| + |V_t(s') - V_t(s)|. \tag{5.3}$$

Applying Assumption 1 to V^* and V_t, and using the definition of ϵ_t, the above reduces to

$$|V^*(s) - V_t(s)| \leq 2\beta\delta_\sigma + \epsilon_t. \tag{5.4}$$

Using the triangle inequality together with the contraction property of the Bellman operator and Inequality 5.4, for any $s' \in \sigma$:

$$\begin{aligned} |V^*(s') - V_t(s')| &= |TV^*(s') - \hat{T}V_{t-1}(s')| \\ &\leq |TV^*(s') - TV_{t-1}(s')| + |TV_{t-1}(s') - \hat{T}V_{t-1}(s')| \\ &\leq \gamma\left(2\beta\delta_\sigma + \epsilon_{t-1}\right) + |TV_{t-1}(s') - V_t(s')|. \end{aligned} \tag{5.5}$$

The term $|TV_{t-1}(s') - V_t(s')|$ is known as the projection error and depends on how well the features are able to represent the Bellman backup operator. This approximation error can be made small by having a large enough feature space [Munos and Szepesvári 2008]. Assuming the algorithm performs N iterations, let $\kappa = \max_{0<t\leq N} |TV_{t-1}(s') - V_t(s')|$ be the largest such error. Thus,

$$|V^*(s') - V_t(s')| \leq \gamma\left(2\beta\delta_\sigma + \epsilon_{t-1}\right) + \kappa. \tag{5.6}$$

Substituting the above into the definition of ϵ_t and noting the initial condition $\epsilon_0 \leq \dfrac{2R_{\max}}{1-\gamma}$ produces the inequality

$$\epsilon_t \leq \gamma\left(2\beta\delta_\sigma + \epsilon_{t-1}\right) + \kappa, \text{ subject to } \epsilon_0 \leq \frac{2R_{\max}}{1-\gamma}. \tag{5.7}$$

Solving the above recurrence relation for ϵ_t and substituting the answer into Inequality 5.4 gives that for all $s \in \mathcal{S}$:

$$\begin{aligned} |V^*(s) - V_t(s)| &\leq 2\beta\delta_\sigma + \left(\frac{1}{1-\gamma}\right)\left((1-\gamma^t)(2\gamma\beta\delta_\sigma) + (1-\gamma^t)\kappa - \left(\frac{2R_{\max}}{1-\gamma}\right)\left(\gamma^t(\gamma-1)\right)\right) \\ &= \left(\frac{1}{1-\gamma}\right)\left(2\beta\delta_\sigma(1-\gamma) + (2\gamma\beta\delta_\sigma + \kappa)(1-\gamma^t) + 2\gamma^t R_{\max}\right) \\ &= \left(\frac{1}{1-\gamma}\right)\left(2\beta\delta_\sigma(1-\gamma^{t+1}) + \kappa(1-\gamma^t) + 2\gamma^t R_{\max}\right) \\ &\leq \frac{2\beta}{1-\gamma}\delta_\sigma + \frac{2\gamma^t R_{\max}}{1-\gamma} + \frac{\kappa}{1-\gamma}, \quad \text{since } \gamma < 1. \end{aligned}$$

□

While the above proof does not refer to MCTS in any way, it suggests that the smoothness of a domain can be beneficial to sample-based methods, as it allows them to calculate a fairly accurate approximation of the entire state-space without having to visit every state repeatedly. The assumption of smoothness is a fairly strong one, as it immediately constrains the value of an unseen state to be within some bound of its neighbour. Thus an agent is able to infer a great deal from only a few samples.

5.2 The FUNCTION OPTIMISATION Domain

Reasoning about the smoothness (or lack thereof) of an MDP is difficult for anything other but the vaguest of statements. To develop some method of controlling and visualising the smoothness, consider the task of finding the global maximum of a function. Different functions can then be used, each of which corresponds to some class of value function. Simple, monotonic functions can be seen as representing smooth environments, while complicated or periodic functions represent non-smooth domains.

Despite initial appearances, the task can indeed be formulated as an MDP, which can then be searched by UCT. For simplicity, the domain and range of the functions to be optimised are constrained to be in the interval $[0, 1]$. Each state in the MDP represents some interval within the unit interval, with the starting state representing $[0, 1]$. The actions at a state partition it into some subinterval, which then becomes the new state. Thus as the agent executes actions, it finds itself in smaller and smaller regions of state-space.

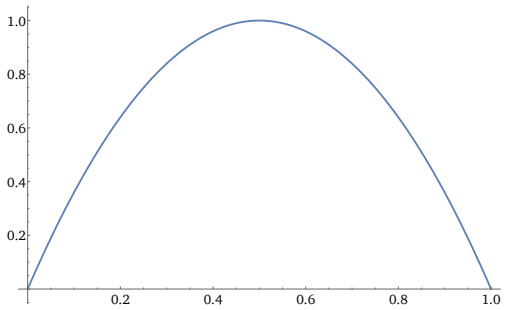
While this formulation allows for any number of actions, assume for these purposes that there are two available actions at each state $[a, b]$: the first action results in a transition to the new state $[a, \frac{a+b}{2}]$, while the second action transitions to $[\frac{a+b}{2}, b]$. This approach forms a binary tree that covers the entire state-space. As this partitioning could continue *ad infinitum*, the tree is truncated at a certain depth. The leaf nodes are those intervals whose size is less than some small threshold. In this instance, a state $[a, b]$ is considered a leaf if $b - a \leq 10^{-5}$.

The implementation of this domain differs to that of Coquelin and Munos [2007] in one respect: here simulations are explicitly performed.[2] Actions are executed until a leaf is encountered, at which point some reward is received. Let f be the function to be optimised and c be the midpoint of the leaf reached by the rollout. At iteration t, a binary reward r_t, drawn from a Bernoulli distribution $r_t \sim \text{Bern}\left(f(c)\right)$, is generated. That is, $p(r_t = 1) = f(x)$ and $p(r_t = 0) = 1 - f(x)$.

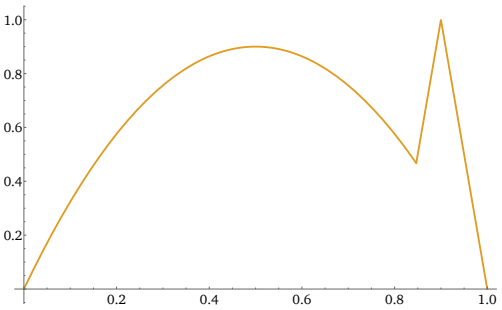
With the states, actions, transitions and rewards specified, UCT can be applied as usual. At the completion of the algorithm, the score is calculated by traversing the lookahead tree from root to leaf, selecting at each state the most visited child. The centre of the interval of the final node reached represents UCT's belief of the location of the global maximum.

[2]The alternative approach samples rewards directly from the expanded node, with error inversely proportional to the depth of the node (so that nodes near the root have larger errors than those further down the tree). As this method does not explicitly execute actions, it is not completely analogous to instances where a state-action mapping is used to bias rollouts.

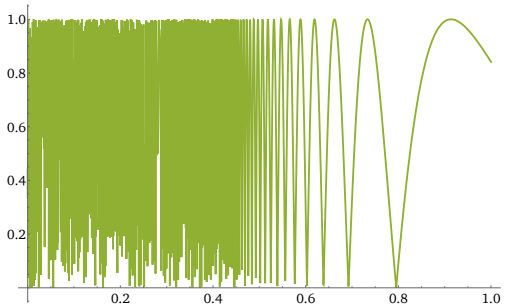
To illustrate the response of UCT to both simple and complicated functions, consider the following four functions as listed in Figure 5.1.



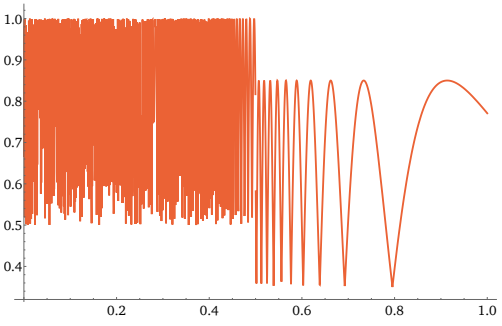
$$f(x) = 4x(1-x)$$



$$g(x) = \max\left(3.6x(1-x), 1 - 10|0.9 - x|\right)$$



$$h(x) = |\sin \tfrac{1}{x^5}|$$



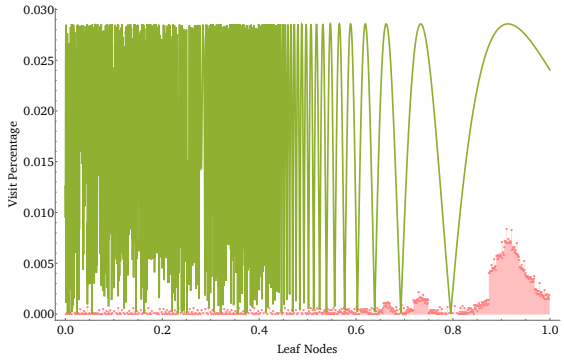
$$j(x) = \begin{cases} \frac{1}{2} + \frac{1}{2}|\sin \frac{1}{x^5}| & \text{if } x < \frac{1}{2} \\ \frac{7}{20} + \frac{1}{2}|\sin \frac{1}{x^5}| & \text{if } x \geq \frac{1}{2} \end{cases}$$

Figure 5.1: Plots of the various functions that are to be optimised. The complexity of the functions increases going through the table, with the right column representing a more complex variant of its left column counterpart.

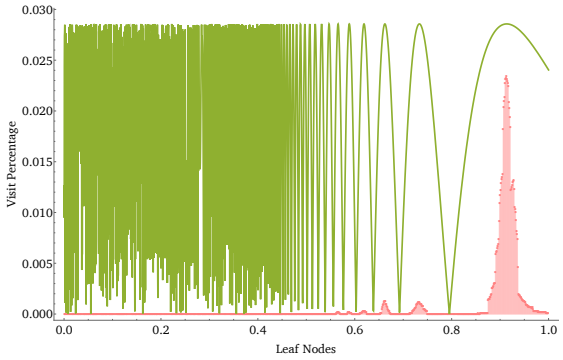
5.2.1 Preference for Smoothness

Notice that the frequency of the function h decreases as x increases. Since the function attains a maximum at many points, one can expect UCT to return the correct answer frequently. Visiting a "turbulent" region of the domain here is not too detrimental, since there is most likely still a state that attains the maximum in the interval.

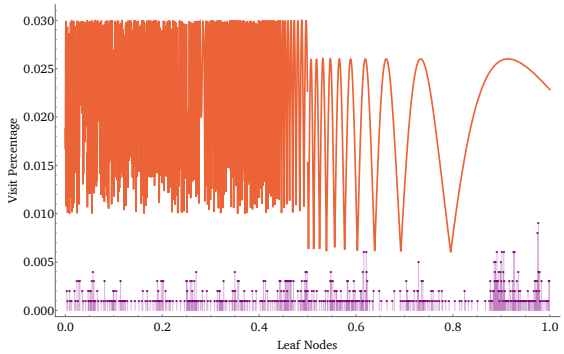
With that said, there is clearly a smoother region of the space that can be searched. In some sense, this is the more conservative space to search, since a small perturbation does not result in too great a change in value. Indeed, UCT prefers this region, as can be seen in Figures 5.2a and 5.2b, where the leaf nodes concentrate around this smooth area, despite there being many other optimal states at $x \leq \frac{1}{2}$. As shall be shown in the next chapter, this is only true for certain rollout policies — a deterministic policy, for example, has no preference when it comes to the smoothness of the region.



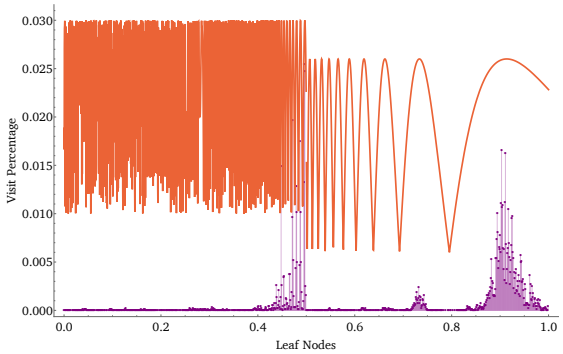
(a) Percentage of visits to leaves after 1000 iterations of UCT for the function $h(x)$.



(b) Percentage of visits to leaves after 50000 iterations of UCT for the function $h(x)$.



(c) Percentage of visits to leaves after 1000 iterations of UCT for the function $j(x)$.



(d) Percentage of visits to leaves after 50000 iterations of UCT for the function $j(x)$.

Figure 5.2: The percentage of total visits assigned to each leaf node in the state-space. A scaled version of $h(x)$ and $j(x)$ is overlaid for reference. For illustration purposes, leaves are evenly grouped into 1000 buckets, with the sum of the visits to the leaves in each bucket plotted.

With this knowledge, it is possible to construct a domain with which UCT will struggle. Consider the function $j(x)$, which has the same number of critical points as h. However, the "safer" interval of the function's domain (at $x \geq \frac{1}{2}$) preferred by UCT is now suboptimal. In this case, UCT finds it difficult to make the transition to the true optimal value, since it prefers to exploit the smoother, incorrect region (Figure 5.2c).

After a sufficient number of simulations, however, UCT does indeed start to visit the optimal region of the graph (Figure 5.2d). Since the value of nearby states in this region changes rapidly, robust estimates are required to find the true optimum. As borne out by UCT's average returns (Figure 5.3), this is not the case. For function j, UCT achieves a lower score than even that of the local maxima. This suggests that the search spends time at the local maxima, before switching to the region $x < \frac{1}{2}$. However, because most of the search had not focused on this space previously, it is forced to re-enter an exploration phase within the interval, resulting in very poor returns.

Functions h and and j are particularly complicated, with multiple critical points and, in the case of j, local maxima. Consider the simpler functions f and g instead. f has only one local maximum, which is also its global maximum, whereas g has one local and one global maximum. Despite the fact that g exhibits a far greater smoothness than either h or j, UCT also occasionally fails to find the true optimal value (Figure 5.4).

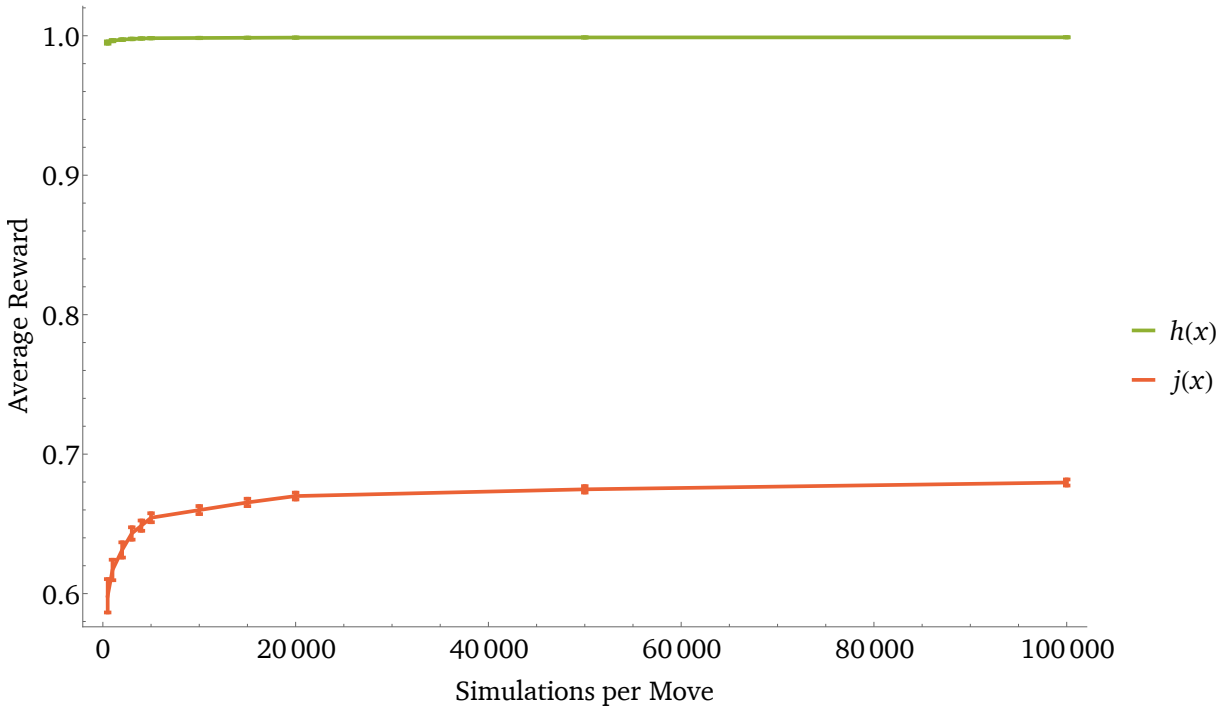


Figure 5.3: Average maximum value found by UCT for the functions h, j, with an increasing number of iterations. Results were collected and averaged over 100 runs.

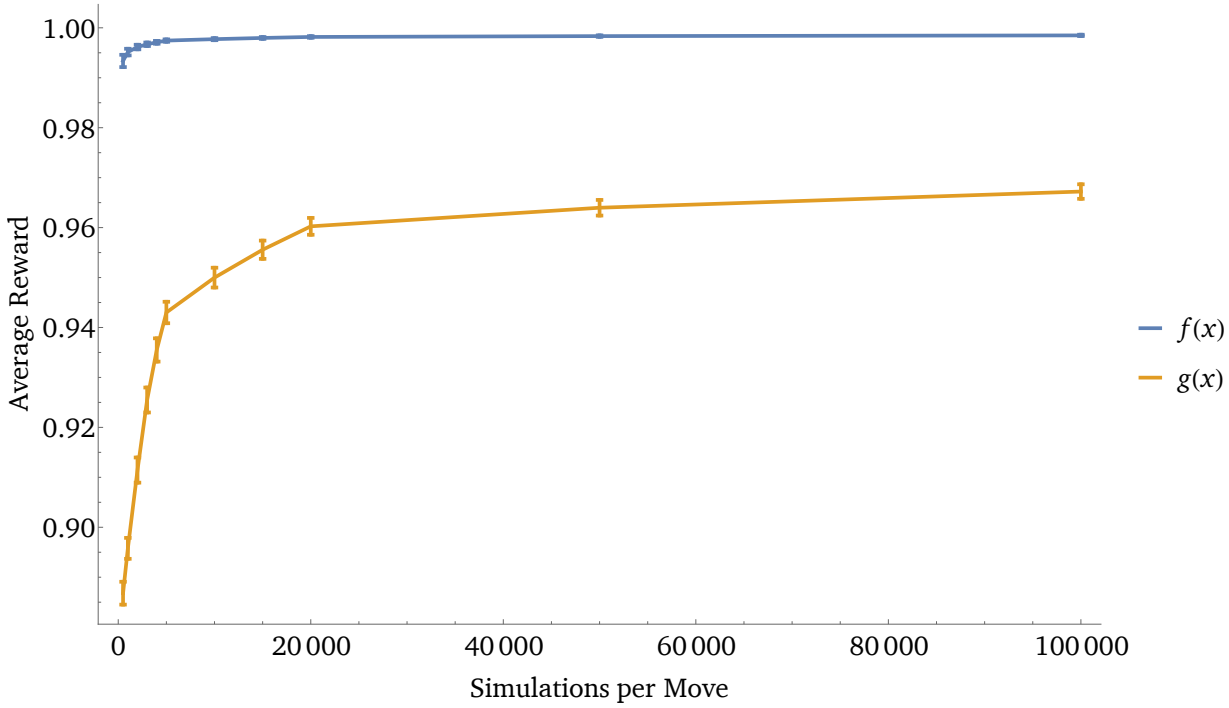


Figure 5.4: Average maximum value found by UCT for the functions f, g, with an increasing number of iterations. Results were collected and averaged over 100 runs.

5.3 Conclusion

This chapter demonstrated UCT's preference for smooth domains, as well as the inherent advantages of such a domain. Section 5.1 proved that an agent is able to calculate a good approximation of the optimal value function for smooth domains without having to sample the entire state-space. With such an advantage, it is unsurprising that UCT is able to act near-optimally in such a domain. In particular, it was demonstrated in Section 5.2.1 that the best-case scenario occurs when the correct answer is in some smooth region of the state-space, as UCT is able to build an accurate evaluation of the states' values. When this does not occur, however, the value of states is unbounded and UCT struggles to estimate accurately, resulting in poor performance.

Having investigated the types of domains to which UCT is well-suited, the next chapter examines what occurs when non-uniformly random rollouts are instead implemented.

Chapter 6

Bias in the Simulation Phase

Oftentimes rollouts that are not uniformly random are referred to as *biased* rollouts. Since the simulation phase is a substitute for the evaluation function, it is quite clear that almost *all*[1] simulation policies suffer from some bias, even uniformly random ones. As both deterministic and random rollouts — policies at opposite ends of the spectrum — are biased in some way, it is important to differentiate between the two. This chapter provides a discussion of the simulation bias and examples of dangerous or risky bias.

6.1 The Effect of Injecting Knowledge

To draw an analogy, consider the approach of *Bayesian inference*. Here a prior distribution, which represents the knowledge injected into the system, is modified by the evidence received from the environment to produce a posterior distribution. Arguments can be made, albeit primarily philosophical in nature, for selecting a maximal entropy prior — that is, a prior that encodes the minimum amount of information. Based on this *principle of indifference*, the posterior that is produced is primarily data-driven, with little bias owing to the injected knowledge.

Selecting a prior distribution that has small variance, for instance, has the opposite effect. Too "narrow" a prior, and far more data will need to be observed to change it significantly. Thus, a prior with low entropy can effectively overwhelm the evidence received from the environment. If such a prior is incorrect, this can result in a posterior with a large degree of bias.

In the context of MCTS, the rollout policy can be viewed as a kind of prior distribution — one which encodes the user's knowledge of the domain, with uniformly random rollouts representing maximal entropy priors, and deterministic rollouts minimal ones.

To illustrate the advantage of selecting a simulation policy with high entropy, consider the global optimisation domain of Section 5.2. Random simulations are biased by performing a one-

[1] A simulation policy has no bias only if it induces the optimal value function or any monotonically increasing transformation thereof.

step lookahead, selecting an action proportional to the value of the next state. Also considered is an inversely-biased policy, which selects an action in inverse proportion to its value. The choice of rollout policy affects the initial view MCTS has of the function to be optimised. The figures in Table 6.1 demonstrate this phenomenon for the random, biased and inversely-biased policies when optimising the function $y(x) = \frac{|\sin(5\pi x) + \cos(x))|}{2}$.

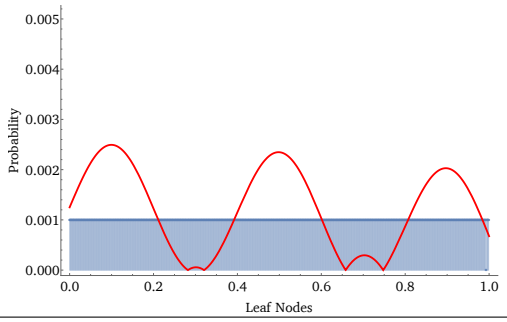
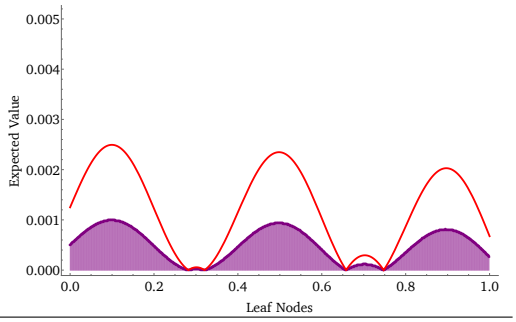
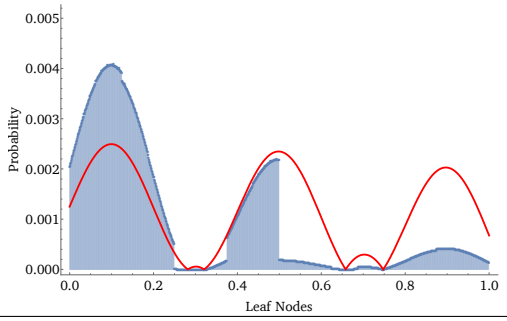
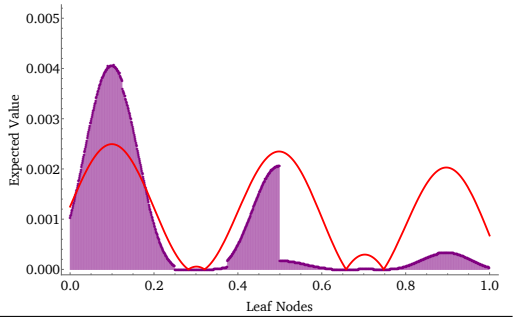
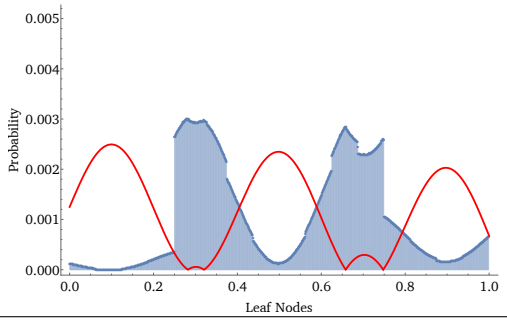
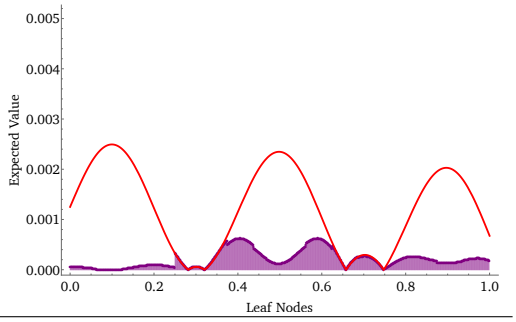
Policy	Probability	Expected Value
Random		
Biased		
Inverse		

Table 6.1: Plots illustrating the effect of the different policies on the overlaid function. The middle column indicates the probability of visiting each leaf node from the starting state under the relevant policy, while the last column shows the probability of reaching each leaf multiplied by its value.

The above graphs indicate the beliefs the different policies have regarding $y(x)$. Random rollouts perfectly represent the function, since their expected values depend only on the function's value itself, while the biased policy assigns greater importance to the region about the true maxi-

mum, but does not accurately represent the underlying function. This serves to focus the search in the correct region of the space, as well as effectively prune some of the suboptimal regions. For this particular example, this is not detrimental since the underestimated regions do not contain the global maximum. Were the optimal value to exist as an extreme outlier in the range $[0.5, 1]$, then the policy would hinder the ability of MCTS to find the true answer, as it would require a large number of iterations to correct this error. A sufficiently smooth domain would preclude this event from occurring.

Finally, the bottom two figures demonstrate how an incorrectly biased policy can cause MCTS to focus initially on a completely suboptimal region. This means that these suboptimal regions would need to be refuted before the correct ones are added to the tree and evaluated. Many iterations would therefore be required to redress the serious bias injected into the system, which is unlikely to be the case for medium to large domains.

6.2 Risky Simulation Policies

To illustrate the possible risk in selecting the incorrect simulation policy, consider a perfect k-ary tree which represents a generic extensive-form game of perfect information. The set of vertices represents the state-space $\mathcal{S}$ of the game, while the edges of the tree are simply the available actions at each state, so that the action-space is the set $\mathcal{A}(s) = \{0, 1, \ldots, k-1\}$. Rewards in the range $[0, 1]$ are assigned to each leaf node such that $\forall s, \pi^*(s, \lfloor \frac{k}{2} \rfloor) = 1$. For non-optimal actions, rewards are distributed randomly. To simplify, a k-ary tree of height h is referred to as a $[k, h]$ tree henceforth. An example of a $[3, 2]$ tree is illustrated by Figure 6.1.

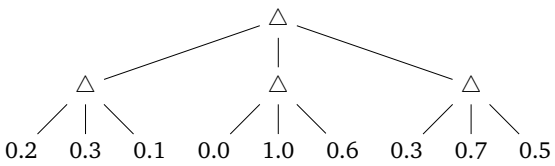


Figure 6.1: Example of the k-ARY TREE domain. The illustrated tree has depth 2 and branching factor 3, with rewards at each leaf node. Note that the optimal child is 1 everywhere.

Consider the case of a UCT agent attempting to maximise its score in the above domain. A uniformly random rollout policy π_{rand} is used to act as a baseline with which to compare the performance of other simulation policies. These policies select an action by sampling from normal distributions over the action space with varying mean (μ) and standard deviation (σ) — that is, policies are parameterised by $\beta \sim \mathcal{N}(\mu, \sigma)$ such that

$$\pi_\beta(s, a) = \begin{cases} 1 & \text{if } a = \lfloor \beta \rceil \bmod k \\ 0 & \text{otherwise,} \end{cases} \tag{6.1}$$

where $\lfloor \beta \rceil$ rounds to the nearest integer, away from zero. In other words, $\lfloor \beta \rceil = \text{sign}(\beta) \lfloor |\beta| + 0.5 \rfloor$.

Figure 6.2 presents the results of an experiment conducted on a $[5, 5]$ instance of the domain. The MCTS algorithm is limited to 30 iterations per move decision in order to simulate an environ-

ment in which the state-space is far greater than what would be computable given the available resources. Both the mean and standard deviation are incrementally varied from 0 to 4, and are then used to parameterise a UCT agent. The agent is then tested on 10 000 different instances of the tree and its returned action recorded.

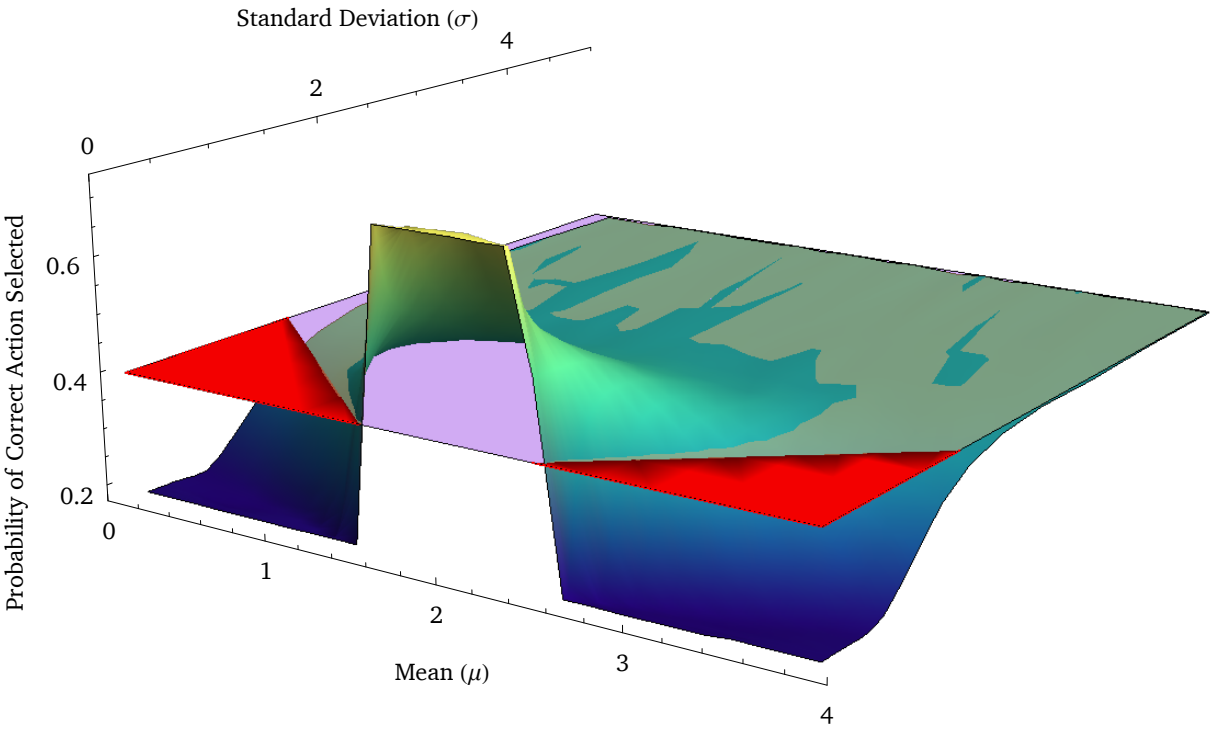


Figure 6.2: Results of rollout policies averaged over 10 000 $[5, 5]$ games. The x and y axes represent the mean and standard deviation of the rollout policy used by the UCT agent, while the z-axis denotes the percentage of times the correct action was returned. The performance of a uniformly random rollout policy (which returned the correct move 39.421% of the time) is represented by the plane, while the red region indicates policies whose means are more than one standard deviation from the optimal policy. The value of the graph at each point is mapped to the colour spectrum between blue and yellow, where blue represents the lowest value and yellow the highest. The above result is indicative of the outcome for different games, independent of branching factor and tree height.

The results demonstrate that there is room for bettering random rollouts. Quite naturally, the performance of the UCT agent is best when the distribution from which rollout policies are sampled are peaked about the optimal action. However, the worst performance occurs when the rollouts have incorrect bias and are over-confident in their estimation (that is, with small standard deviations), their performance dropping below even that of random. When the rollouts have too great a variance, their performance degenerates to that of random. There is thus only a small window for improvement, which requires the correct bias and low variance. One should be certain of the correct bias, however, as the major risk of failure occurs for low-variance, high-bias distributions.

This phenomenon also occurs in the FUNCTION OPTIMISATION domain (Section 5.2). Instead

of performing random simulations, consider a greedy playout strategy that selects the best action using a one-step lookahead and the value at the centre point of the next state (with ties broken randomly). More precisely, assume that the aim is to maximise function f, that state s represents the interval $[a, b]$ and that the two available actions are transitioning to the left (l) and right (r) subintervals. Then the deterministic greedy policy $\pi_{max} : \mathcal{S} \to \mathcal{A}$ is given by

$$\pi_{max}(s) = \begin{cases} l & \text{if } f(\frac{3a+b}{4}) \geq f(\frac{a+3b}{4}) \\ r & \text{otherwise.} \end{cases} \tag{6.2}$$

Recall from Section 5.2.1 that vanilla UCT prefers the smooth regions of the function $h(x) = |\sin \frac{1}{x^5}|$, avoiding the riskier states while still achieving the correct result. The same is not true of the greedy policy described above. In this case, the algorithm spends an inordinate amount of time visiting the interval $[0, 0.5]$, occasionally even failing to escape it. Interestingly, another deterministic rollout policy π_{min}, which selects the action with the minimum value, does not suffer as badly (see Figure 6.3). Indeed, it too spends many iterations in the riskier areas of the h, but not to the same extent as π_{max}. This suggests that the overestimation of suboptimal states is, in this instance, worse than the underestimation of optimal states. This underestimation encourages UCT to begin exploring other lines of action, and since the region at $x > 0.8$ is smooth, the minimum value that can be attained is bounded. Thus π_{min} continues to explore this smoother region, eventually settling near the optimal value. A possible explanation for it not finding the true optimal value is that the algorithm wasted a large number of cycles on a different region of the function, and consequently was not able to find the true maximum within the remaining number of iterations.

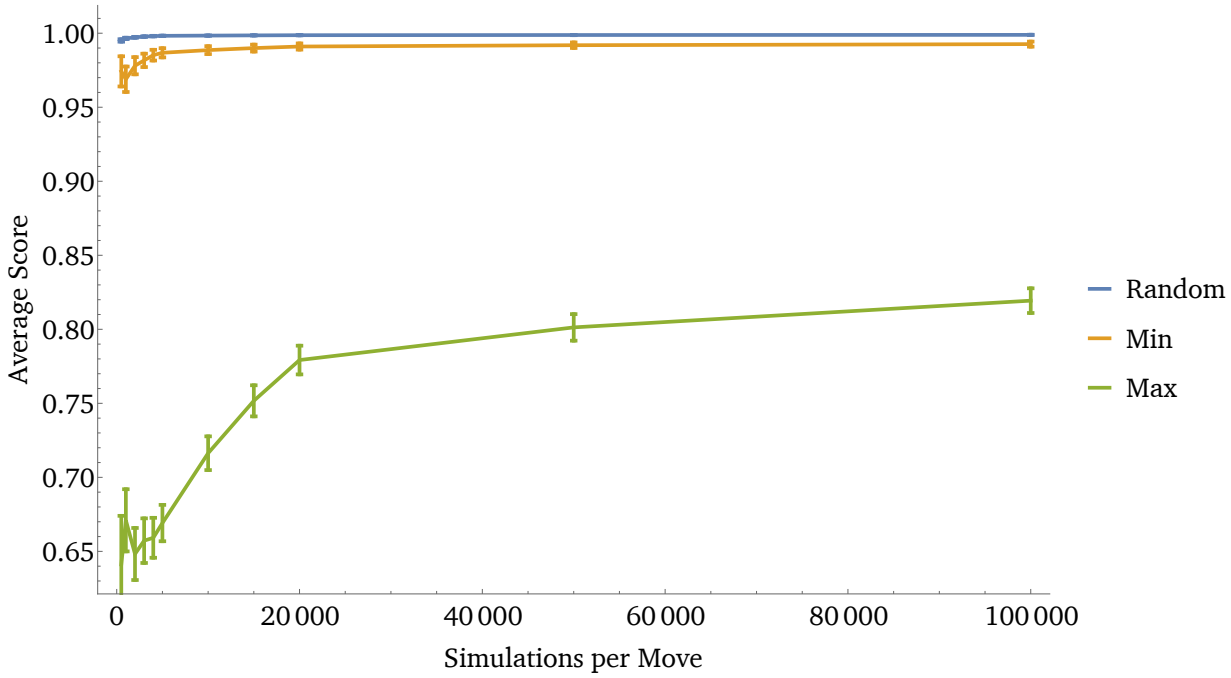


Figure 6.3: Results of the various rollout policies (as described above) for the function $h(x) = |\sin \frac{1}{x^5}|$ averaged over 100 runs.

6.3 Advantages of Biased Simulations

The previous sections suggest that the best course of action may be to select uniformly random rollouts and forego the associated risks that come with executing heavy playouts. However, not only can correctly biased rollouts greatly improve the performance of MCTS (refer Figure 6.2), but in certain domains (for example, MAGIC: THE GATHERING [Cowling *et al.* 2012]) they are all but required. These domains have extremely large state-spaces — even more so than, say, Go — usually as a result of stochasticity or hidden information in the game. In these domains, random trajectories represent only a vanishingly small part of the overall state-space. With so large a variance, these rollouts provide almost no information, and performing a sufficient number of them is simply infeasible. Informed rollouts are therefore sometimes unavoidable.

To illustrate that biased rollouts can indeed be of great benefit in the correct situation, consider again the k-ary tree domain of Section 6.2, with $k = 2$. In this instance, the available actions consist of selecting either the left or right child of a node — that is, $\mathcal{A}(s) = \{l, r\}$. Rewards are distributed such that action l is always optimal. Different rollout policies specified by the value assigned to the left action are applied to UCT, and their performances in a $[2, 20]$ domain recorded.

In such a relatively small and simple domain, it is difficult to ascertain the effect of the various policies. When simply recording whether the correct action was selected at the root node, all rollout policies resulted in the correct action selection consistently. Calculated instead is the ratio of visits to the correct action — that is, the percentage of times UCB1 selected the optimal action at the root node during the UCT algorithm. These results are illustrated by Figure 6.4 and demonstrate the "confidence" UCT has regarding the final action to select.

Plotted, too, is the bias at the leaf node (Figure 6.5) — the difference between the leaf node's value under the optimal policy and its value according to UCT. The results indicate that, with correctly biased simulation policies, UCT experiences the appropriate improvement, which increases with a decrease in policy variance. Note that despite the extreme bias of the incorrect rollout policies, they are still able to ultimately pick the correct action. This occurs because, while the values of both actions are incorrect, they are still such that action l is preferred to r.

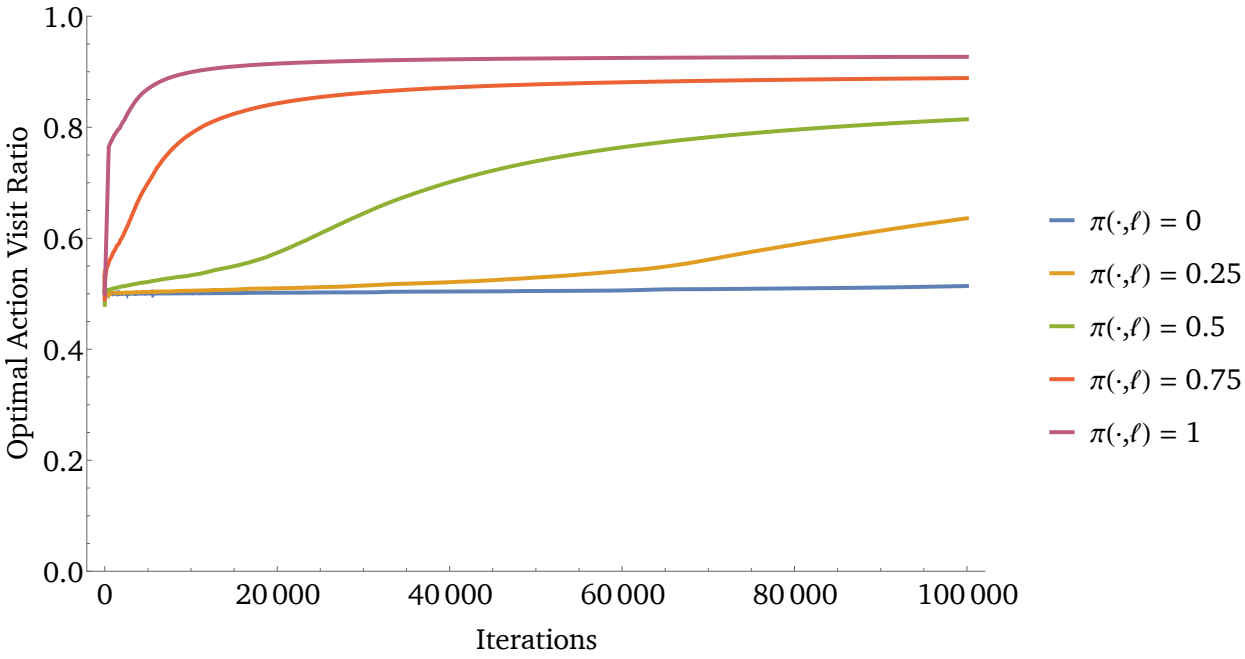


Figure 6.4: The ratio of visits to the optimal action for the various rollout policies (which specify the probability of selecting action l at every state) in a binary tree environment averaged over 100 runs. The values provide a measure of the confidence UCT has in selecting its action.

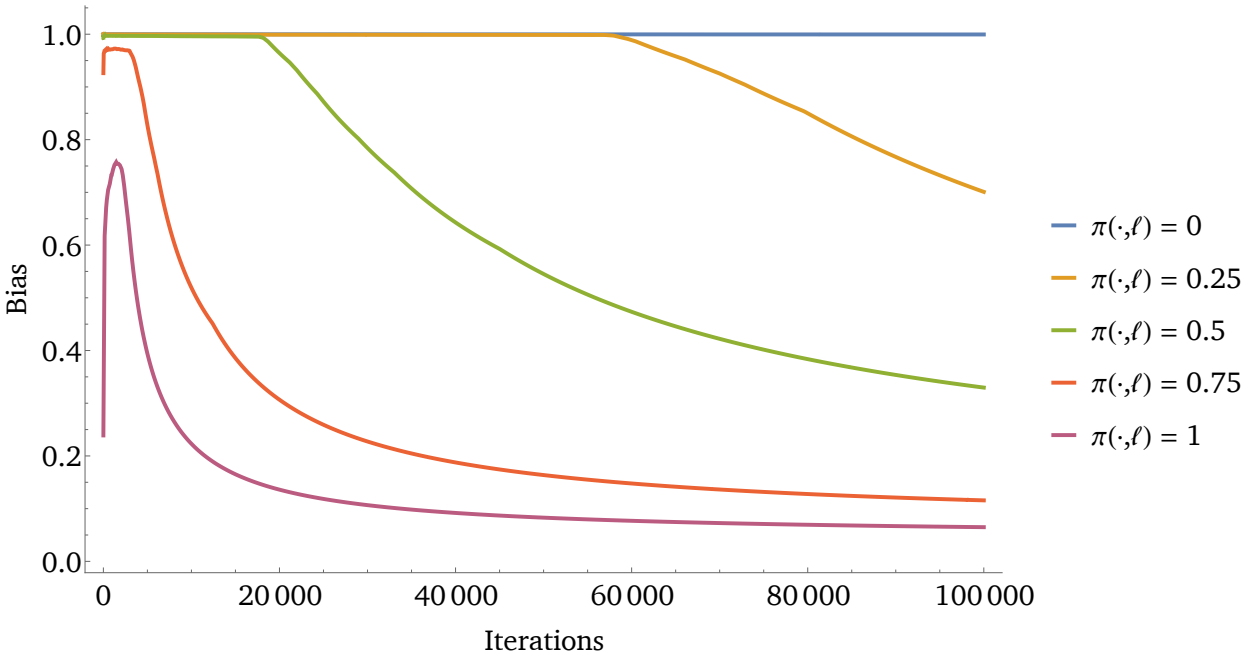
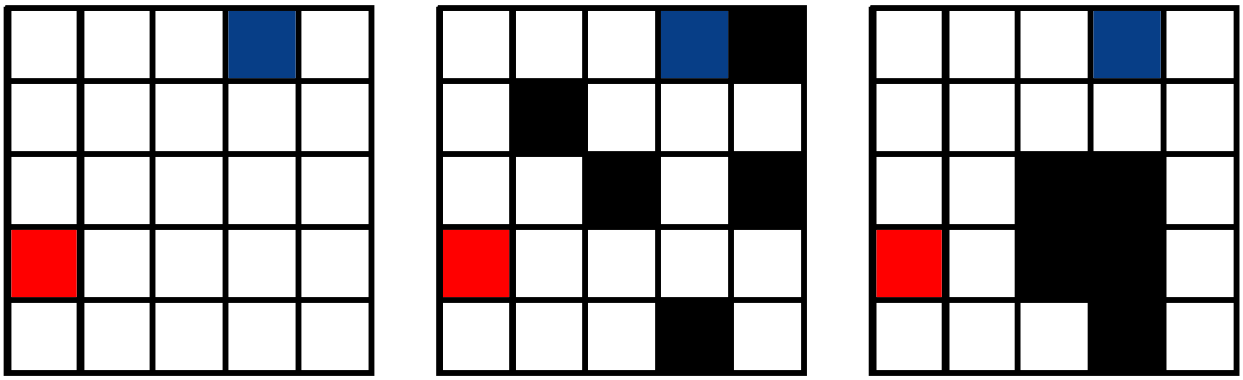


Figure 6.5: The error (bias) in the estimation of the root node for the various policies (which specify the probability of selecting action l at every state) averaged over 100 runs. As expected, the bias of the various policies maps to their visit ratios — the stronger policies that select the correct action more often suffer from less bias.

6.4 Incorrect Rollouts in Non-Smooth Domains

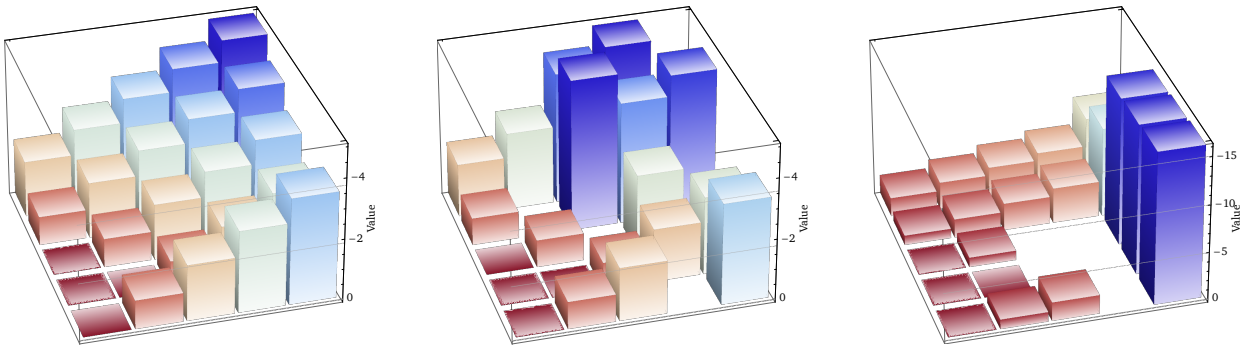
One interesting question is the manner in which the rollouts allow MCTS to handle noise or unexpected encounters in a domain. To investigate this, consider a GRID NAVIGATION domain in which an agent navigates a grid in an attempt to get from one square to another in the fewest number of steps (the start and end squares are randomly assigned). A number of obstacles may be placed on the grid according to two strategies. The first is to simply place obstacles randomly, while the second is to form a cluster or grouping of obstacles. Informally, clustered obstacles only affect an isolated region of the state-space, while the random placing of obstacles has the ability to affect the entire space. The clustered obstacles therefore create a localised non-smooth region, whereas the randomly-placed obstacles make the entire space non-smooth. Figure 6.6 illustrates the domain, while Figure 6.7 demonstrates the effect of the different obstacle structures on the value function.



(a) Grid with no obstacles. (b) Grid with random obstacles. (c) Grid with clustered obstacles.

Figure 6.6: Illustration of the GRID NAVIGATION domain on a 5×5 grid. The blue square indicates the starting state of the agent and the red the goal state, while the black squares represent obstacles.

The dynamics and reward structure of the domain are described thusly. First, the agent has four available actions at each state — `UP`, `DOWN`, `LEFT` and `RIGHT` — which result in it moving one square in the given direction. If the agent executes an action that would cause it to collide with an obstacle or leave the grid, it then remains in the same state and receives a reward of -10. An agent terminates and receives a reward of 0 if it enters the goal state, and -1 in all other cases. The value returned by a rollout is calculated by adding the rewards it receives at each simulated step, until either the goal state is encountered or the sum becomes less than -1000. The final sum of rewards is then mapped linearly to the range $[0, 1]$.



(a) Value of states in grid with no obstacles.

(b) Value of states in grid with random obstacles.

(c) Value of states in grid with clustered obstacles.

Figure 6.7: Illustration of the effect of obstacles in the GRID NAVIGATION domain on a 5×5 grid. The values plotted are the value function under a myopic policy which always heads in the direction of the goal, regardless of obstacles. The above clearly demonstrates how the clustered obstacles affect only a local region of the space, preserving the smoothness elsewhere, while the randomly placed obstacles affect the entire state-space in unexpected ways.

In order to create a policy for this domain, each action is assigned a value of 1 to begin with. Actions that lead to states closer to the goal (ignoring obstacles) have some additional weight added to them. In Figure 6.6, for example, `LEFT` and `DOWN` would have some extra value added to their weight, since they leave the agent closer to the goal. An action is then selected proportionally to its assigned value. For instance, if the additional value is x, then `LEFT` and `DOWN` would be selected with probability $\frac{x+1}{2+2(x+1)}$.

The policies are parameterised by the value that is added to these actions. For instance, π_0 represents a uniformly random policy, while π_∞ is a deterministic[2] greedy policy. The performances of four policies (π_0, π_1, π_5, π_∞) are then tested in a 10×10 grid with no obstacles, 15 obstacles and 15 clustered obstacles. The results are presented in Figures 6.8, 6.9 and 6.10 respectively.

With no obstacles, the results are fairly straightforward and expected. The greedy policy, which in this case is also the optimal policy, is the most successful, the random the least and the others in between. When obstacles are added randomly, the situation changes completely. Since the rollout policies have not changed and were constructed to head towards the goal without knowledge of any obstacles, their presence damages the performance of UCT. The worst performing agent in this case is clearly the deterministic policy, while the more conservatively biased policies are the best choice. Random also remains unaffected by the obstacles, with little difference between it and the biased policies.

When the obstacles are clustered together, they only affect a single region of the state-space. The results under these conditions are therefore not as drastic as the randomly placed obstacles. Although the deterministic policy again suffers somewhat, it is not to the same extent as previously.

[2]Ties are broken in a deterministic manner in the order `UP`, `DOWN`, `LEFT` and `RIGHT`.

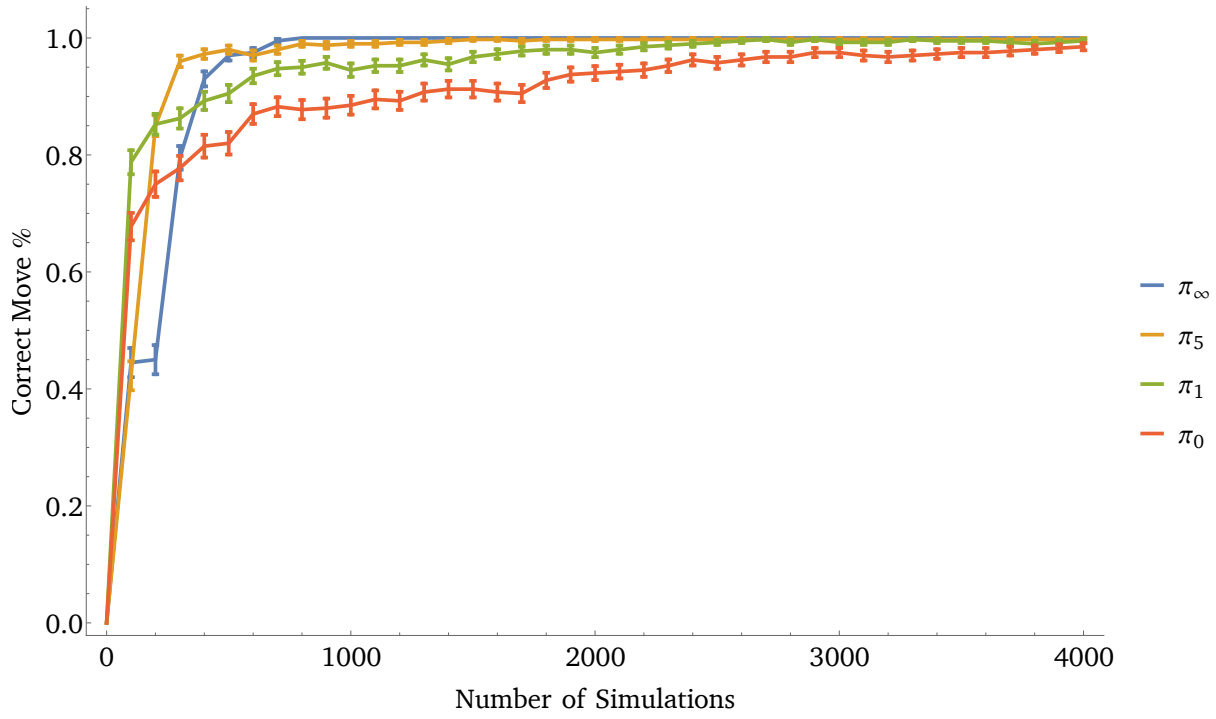


Figure 6.8: Results of various rollout policies in the GRID NAVIGATION domain in a 10×10 grid with no obstacles, averaged over 400 different instances of the environment.

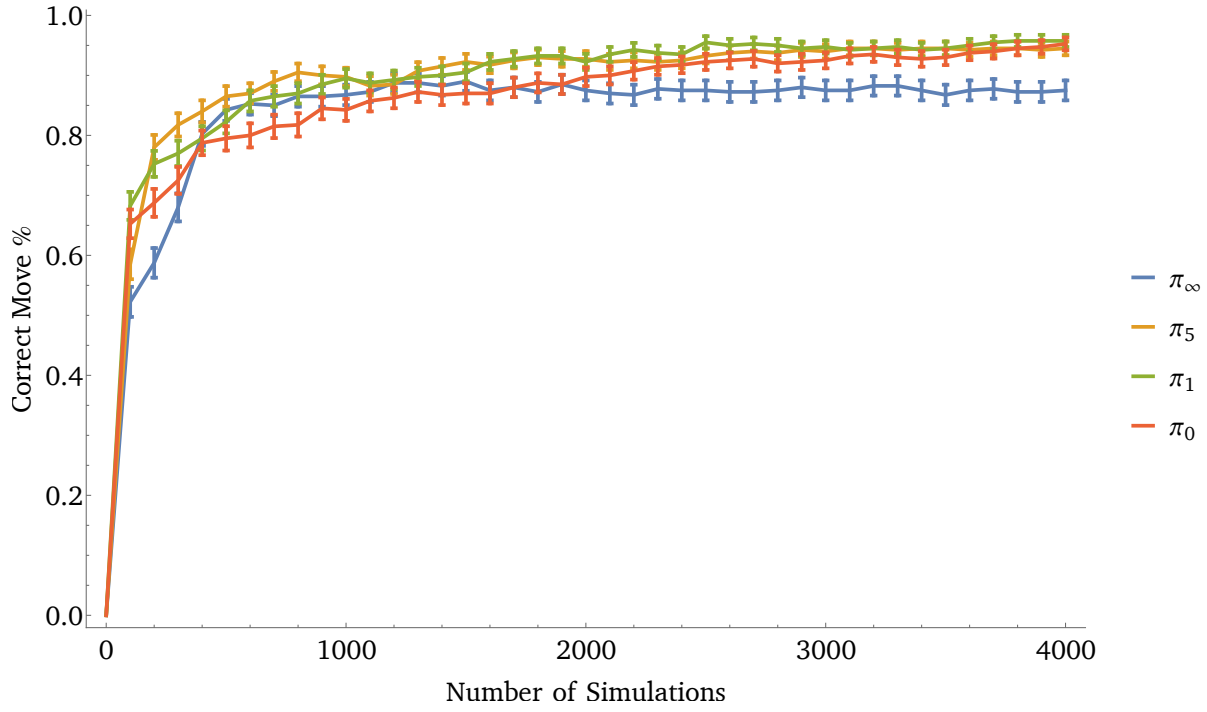


Figure 6.9: Results of various rollout policies in the GRID NAVIGATION domain in a 10×10 grid with 15 obstacles randomly scattered, averaged over 400 different instances of the environment.

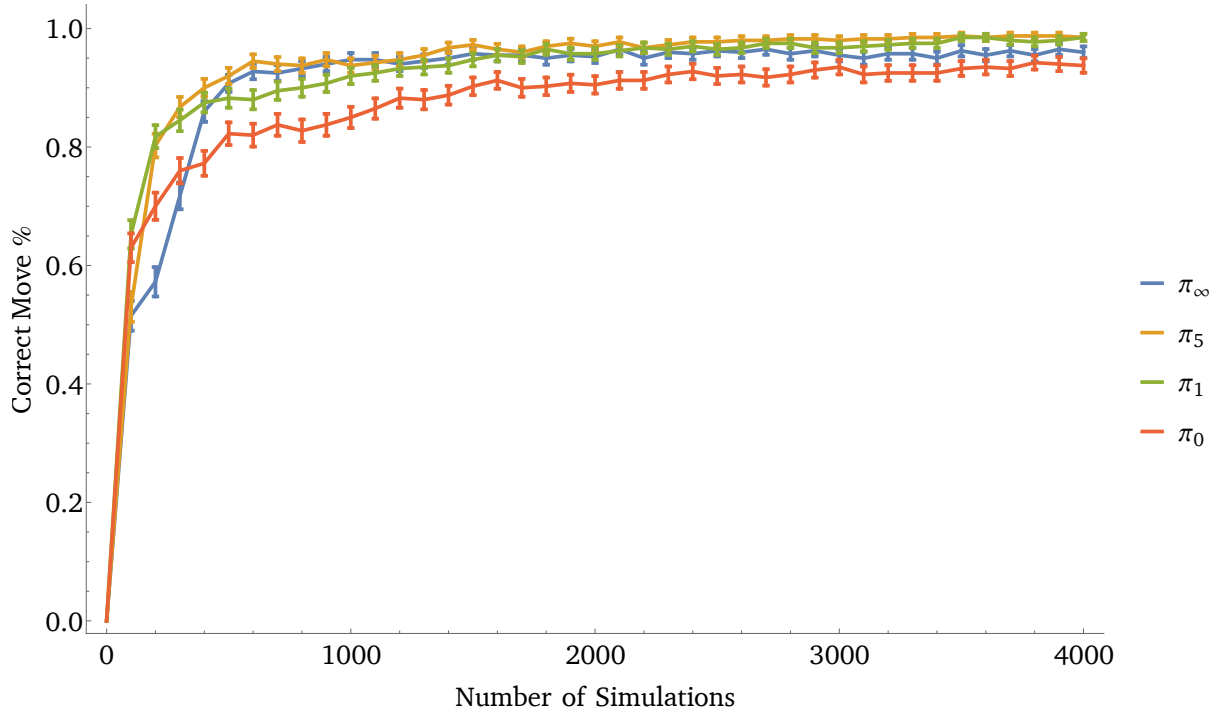


Figure 6.10: Results of various rollout policies in the GRID NAVIGATION domain in a 10×10 grid with 15 obstacles clustered together, averaged over 400 different instances of the environment.

These results appear to suggest that the random and more conservatively biased policies are resilient to unexpected events or noise. Plotting the performance of the policies with an increase in randomly placed obstacles reveals just that (Figure 6.11). Random rollouts appear completely unaffected by the presence of any number of obstacles, while the dropoff in performance of the other policies is inversely proportional to their level of stochasticity. This speaks to the dangers of a high-bias, low-variance policy. In domains where a good policy cannot be constructed, these results suggest using a higher-variance policy to mitigate against any noise or unforeseen stumbling blocks.

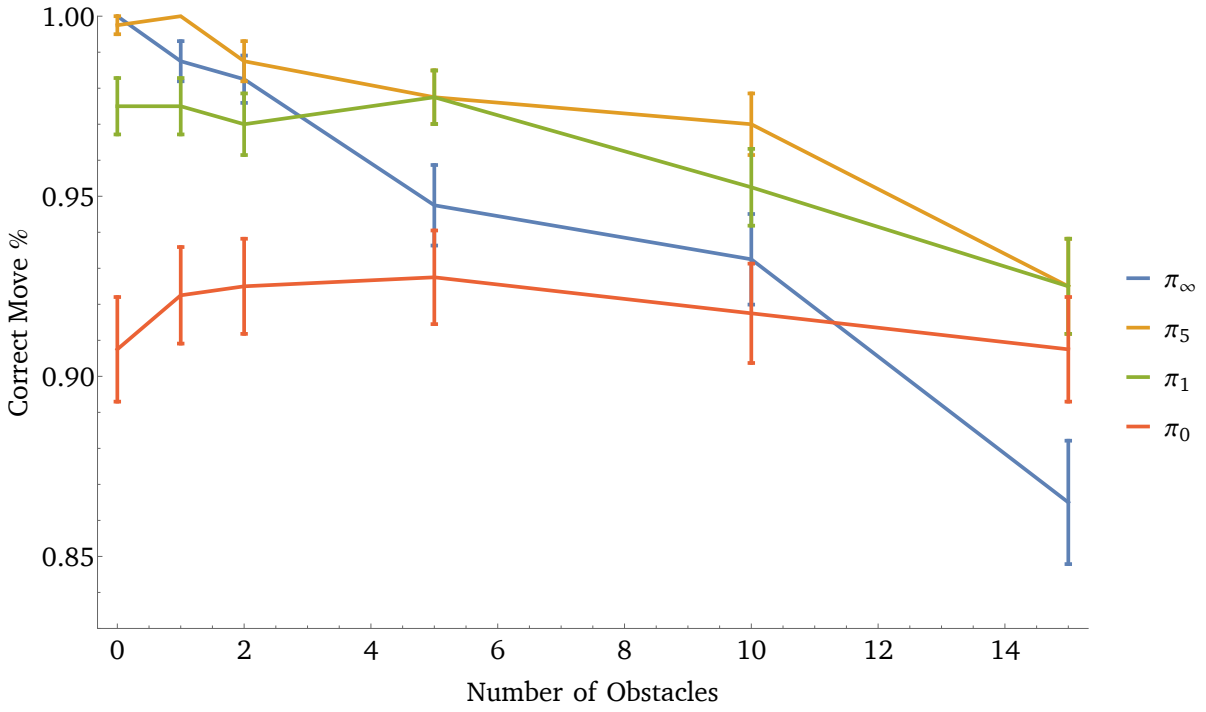
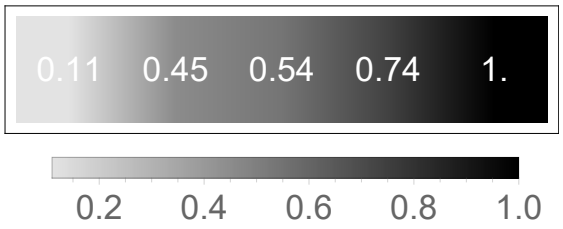
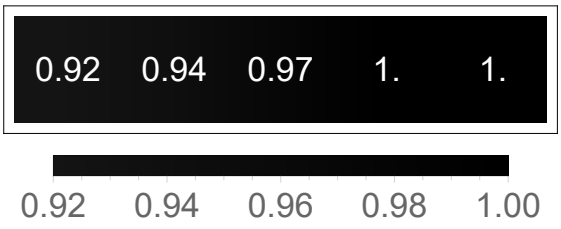


Figure 6.11: Results of various rollout policies in the GRID NAVIGATION domain in a 10×10 grid with an increasing number of randomly placed obstacles, averaged over 400 different instances of the environment. The UCT algorithms were allowed 2000 iterations per action decision.

One additional point of interest is the poor initial performance of π_∞, even when it is indeed the optimal policy. This occurs because the perfect policy actually provides less information than the random rollout, which is able to provide greater distinction amongst the action values. To illustrate, consider a 1×5 grid, where the goal state is the rightmost square and the only available actions are `LEFT` and `RIGHT`. Since the values of states are so close to one another under the optimal policy, UCT continues to explore for a longer period of time, resulting in poor performance at the beginning. The random policy, on the other hand, clearly differentiates between adjacent states, allowing UCT to begin exploiting much earlier. Figure 6.12 demonstrates this phenomenon.



(a) Value of states under a uniformly random policy.



(b) Value of states under the optimal policy.

Figure 6.12: Value of states under uniformly random and optimal policies. Values are scaled from the range $[-35, 0]$ to $[0, 1]$. The differentiation between states under the random policy is clearly evident, but not so in the optimal policy's case. Note, too, the difference in the scales of the figures.

As a final experimental domain, consider an extension of the GRID NAVIGATION task known as the TAXI domain. Here the agent has two additional actions (PICKUP and DROPOFF), which can only be executed at the appropriate state (the agent incurs a penalty of -10 otherwise). The agent's aim is to navigate to some state and execute the PICKUP action, before proceeding to a final state and executing DROPOFF. Figure 6.13 provides a simple illustration of the domain.

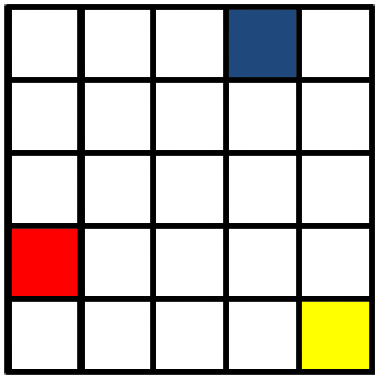


Figure 6.13: Illustration of the TAXI domain on a 5×5 grid. The yellow square represents the taxi which must first navigate to the blue square, pick up the passenger and then drop him off at the red square.

Initially, it may seem like results similar to the those of the GRID NAVIGATION domain can be expected, since this domain can be seen as two sequential instances of that task. However, one key difference is the critical requirement of executing a single action (PICKUP or DROPOFF) in a single state. Thus while the obstacles from the previous experiment provide some additional difficulty, they pale in comparison to the bottleneck caused by having to execute these critical actions at the correct time.

Adopting the same approach as in the previous domain, Figure 6.14 illustrates that the number of randomly-placed obstacles has some effect on the lower variance policies. When compared with Figure 6.11, hoewever, it is clear that this effect is minimal — executing PICKUP and DROPOFF at the proper time is evidently far more important than the presence of the obstacles.

This domain is therefore indicative of many real games, in that selecting the correct action at certain critical times is more important than playing well at all other times. This also supports the approach of rollout policies such as that of MOGO (Chapter 1), which is hardcoded to make critical moves when necessary, but plays randomly otherwise.

6.5 Conclusion

This chapter presented numerous results that tested the effect of biasing simulation policies in different domains. The results indicated that the the correct rollout policy can improve upon the performance of random. However, improperly biased rollouts can be extremely detrimental to the overall MCTS algorithm, with low-variance ones carrying the greatest risk. This is compounded when non-smooth domains are combined with high-bias, low-variance simulations. Furthermore, when these low-variance simulations do well, a more conservative policy often achieves similar

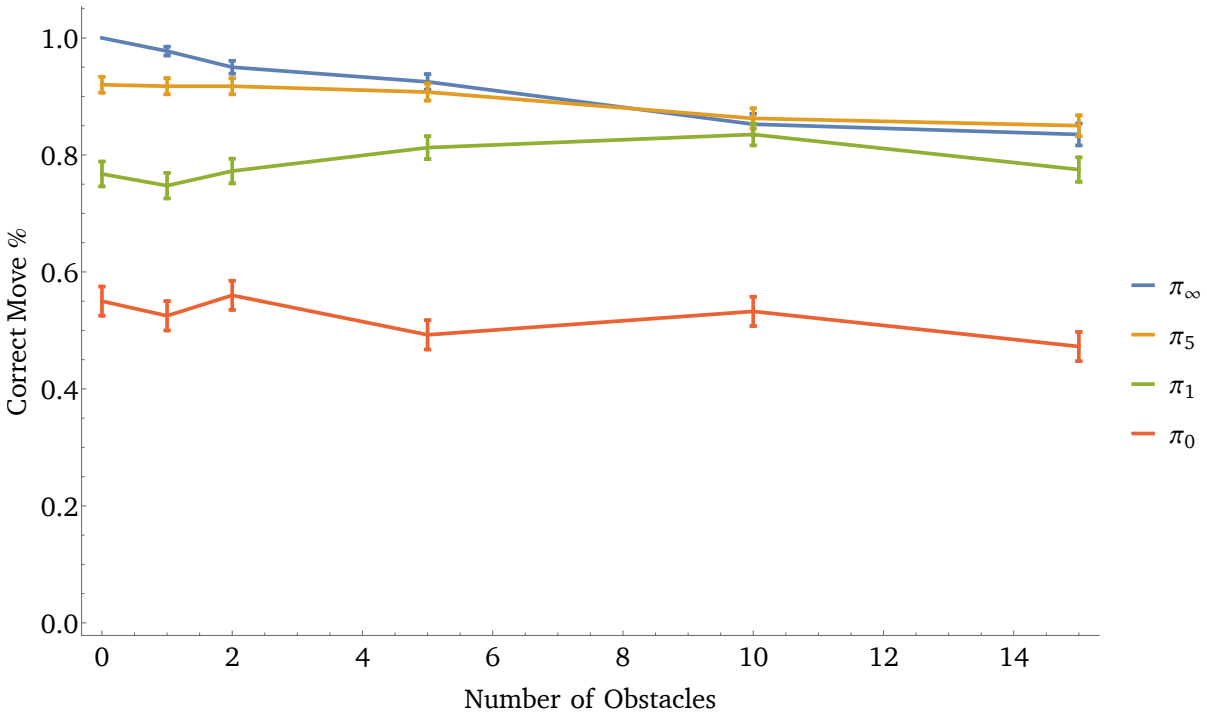


Figure 6.14: Results of various rollout policies in the TAXI domain in a 10×10 grid with an increasing number of randomly placed obstacles, averaged over 400 different instances of the domain. The UCT algorithms were allowed 2000 iterations per move.

performance, while at the same time being robust to noise or errors in the policy. Finally, results from the TAXI domain illustrated the case for low-variance policies. In domains where critical actions must be made at particular states, these policies excel despite being improperly biased.

Chapter 7

Conclusion & Future Work

As ALPHAGO's recent success against Go world champion Lee Se-dol indicates, Monte Carlo Tree Search provides a powerful framework for decision-making in complex domains. In ALPHAGO's case, the combination of MCTS with deep learning brought about improvements not expected for at least another decade. Integrating neural networks into MCTS poses a serious problem, however, in that it adds another layer of complexity and uncertainty into the working of the algorithm. This makes it even harder to reason about the success or failure of a particular experiment in a given domain. Thus any kind of insight into even the most basic of MCTS algorithms may be of huge value to future research. One advantage in particular is that any kind of knowledge about the domain and the expected performance of MCTS can reduce the number of different combinations of techniques and parameters that would otherwise need to be tested in hit-and-miss fashion. In situations that require large amounts of time for training or testing, this could be invaluable.

The large amount of literature available provides a solid, rigorous basis for UCT in terms of convergence guarantees in the limit. Less, however, can be said about its applicability to arbitrary domains and its performance given a finite, realistic number of iterations. Furthermore, a thorough analysis of non-uniformly random rollouts remains conspicuously absent, and this lack of understanding is likely to be aggravated with the addition of more sophisticated enhancements. This research therefore attempted to answer some questions about the performance of UCT and its simulation phase in MDPs.

Chapter 5 demonstrated the gains that can be achieved in smooth domains — an agent need not visit the entire state-space to achieve a good approximation of the value function — as well as UCT's preference for smooth state-spaces, which was demonstrated by attempting to find the global optimum of different functions. This supports the prevailing theories regarding the reason for its poor performance in chess, despite its strong showing in Go.

Also discussed was the effect of different rollout policies on the final action selection of the algorithm. This performance is again closely linked to the smoothness of the domain in question: for extremely smooth value functions, all policies do equally well. However, as the smoothness decreases, the more conservative policies (those closer to uniformly random) hold their own, while high-bias, low-variance policies suffer greatly. There is thus a trade-off, not only in the tree policy, but also in the choice of simulation policy. Selecting a low-variance policy can markedly improve the performance of MCTS in the correct domain, especially in those situations that require a

critical action to be selected at the correct state, but can also result in extremely poor performance, exacerbated by non-smooth domains.

Though this research has described situations when different rollout policies work both well and poorly, the real question is how this can be leveraged. The results from Chapter 6 hint at a particular path forward. Since any single rollout policy has the ability to decrease the performance of MCTS overall, it may be better to learn a distribution over policies. Thus one may start with a uniformly random distribution, updating it when new data is received (either from training samples or perhaps even self-play). The final distribution, which encapsulates some measure of uncertainty about the policies, can then be used to conduct the rollouts. This course of action has its own issues, least of all that policies are already distributions over actions. A distribution over policies may therefore be too abstract, essentially performing no better than random.

One other facet that remains unexplored is the mathematical representation of the rollout policy. Since it replaces, for all intents and purposes, the evaluation function used in a minimax search, it is clearly responsible for applying some kind of transformation to the values of a node's children. If this type of transformation can be determined in a given domain, it would immediately provide insight into the ultimate performance of the MCTS algorithm. For instance, a good transformation would be one that maintains the true ordering over action preferences, or even simply preserves the maximum action. Formulating the rollout strategy as a function applied to action preferences would thus provide a clear indication with regard to its effect on MCTS as a whole.

These are exciting times for the field of artificial intelligence. Dramatic increases in computing power, multicore processors and access to distributed systems suggest that MCTS, which owing to the independent nature of simulations is a good candidate for parallelisation, will continue to be a key point of interest going forward. Furthermore, its success in conjunction with that of the popular deep learning paradigm paves the way for new and interesting techniques and improvements. It is hoped that these results contribute, in some way, to future MCTS-based research and the continued progress of the field in general.

References

[Abramson 1987] B. Abramson. *The expected-outcome model of two-player games*. PhD thesis, Department of Computer Science, Columbia University, 1987.

[Allis 1994] L.V. Allis. *Searching for solutions in games and artificial intelligence*. PhD thesis, Rijksuniversiteit Limburg, Maastricht, 1994.

[Auer *et al.* 2002] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235 – 256, 2002.

[Bai *et al.* 2010] H. Bai, D. Hsu, W.S. Lee, and V.A. Ngo. Monte Carlo value iteration for continuous-state POMDPs. In *Algorithmic Foundations of Robotics IX*, pages 175–191. Springer, 2010.

[Bellman 1957] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Billings *et al.* 2002] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1):201 – 240, 2002.

[Björnsson and Finnsson 2009] 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.

[Browne *et al.* 2012] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1 – 43, March 2012.

[Brügmann 1993] B. Brügmann. *Monte Carlo Go*. Technical report, Syracuse University, 1993.

[Bubeck *et al.* 2011] S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvári. $\mathcal{X}$-armed bandits. *The Journal of Machine Learning Research*, 12:1655–1695, 2011.

[Cai and Wunsch II 2007] X. Cai and D.C. Wunsch II. Computer Go: A grand challenge to AI. In *Challenges for Computational Intelligence*, volume 63 of *Studies in Computational Intelligence*, pages 443 – 465. Springer Berlin Heidelberg, 2007.

[Campbell *et al.* 2002] M. Campbell, A.J. Hoane Jr., and F. Hsu. Deep Blue. *Artificial Intelligence*, 134:57 – 83, 2002.

[Chaslot *et al.* 2008] G.M.J.B. Chaslot, M.H.M. Winands, H.J. van den Herik, J.W.H.M. Uiterwijk, and B. Bouzy. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(3):343 – 357, 2008.

[Chaslot 2010] G. Chaslot. *Monte-Carlo tree search*. PhD thesis, Maastricht University, 2010.

[Clark and Storkey 2014] C. Clark and A.J. Storkey. Teaching deep convolutional neural networks to play Go. *arXiv preprint arXiv:1412.3409*, 2014.

[Coquelin and Munos 2007] P. Coquelin and R. Munos. Bandit algorithms for tree search. In *Uncertainty in Artificial Intelligence*, 2007.

[Coulom 2006] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games*, volume 4630, pages 72 – 83. Springer, 2006.

[Coulom 2007] R. Coulom. Computing Elo ratings of move patterns in the game of Go. In *Computer Games Workshop*, 2007.

[Coulom 2008] R. Coulom. Whole-history rating: A Bayesian rating system for players of time-varying strength. In *Computers and Games*, pages 113 – 124. Springer, 2008.

[Cowling *et al.* 2012] P.I. Cowling, C.D. Ward, and E.J. Powley. Ensemble determinization in Monte Carlo Tree Search for the imperfect information card game Magic: The Gathering. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(4):241–257, 2012.

[Curran *et al.* 2015] W. Curran, T. Brys, M. Taylor, and W. Smart. Using PCA to efficiently represent state spaces. *arXiv preprint arXiv:1505.00322*, 2015.

[Davies 1977] J. Davies. *The Rules and Elements of Go*. Ishi Press, 1977.

[Denardo 1967] E.V. Denardo. Contraction mappings in the theory underlying dynamic programming. *SIAM Review*, 9(2):165–177, 1967.

[Domshlak and Feldman 2013] C. Domshlak and Z. Feldman. To UCT, or not to UCT? (position paper). In *Sixth Annual Symposium on Combinatorial Search*, 2013.

[Enzenberger *et al.* 2010] M. Enzenberger, M. Müller, B. Arneson, and R.B. Segal. Fuego — an open-source framework for board games and go engine based on Monte Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259 – 270, 2010.

[Gelly and Silver 2007] S. Gelly and D Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning*, pages 273 – 280. ACM, 2007.

[Gelly and Silver 2011] S. Gelly and D. Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856 – 1875, 2011.

[Gelly *et al.* 2012] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, 55(3):106 – 113, 2012.

[Ginsberg 2001] M.L. Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:303 – 358, 2001.

[Guez *et al.* 2013] A. Guez, D. Silver, and P. Dayan. Scalable and efficient Bayes-adaptive reinforcement learning based on Monte-Carlo tree search. *Journal of Artificial Intelligence Research*, 48:841–883, 2013.

[Helmbold and Parker-Wood 2009] D.P. Helmbold and A. Parker-Wood. All-moves-as-first heuristics in Monte-Carlo Go. In *IC-AI*, pages 605 – 610, 2009.

[Huang and Müller 2013] S. Huang and M. Müller. Investigating the limits of Monte Carlo tree search methods in computer Go. In *International Conference on Computers and Games, LNCS*. Springer, 2013.

[Huang *et al.* 2011] S. Huang, R. Coulom, and S. Lin. Monte-Carlo simulation balancing in practice. In *Computers and Games*, pages 81–92. Springer, 2011.

[Hunter 2004] D.R. Hunter. MM algorithms for generalized Bradley-Terry models. *Annals of Statistics*, 32(1):384–406, 2004.

[Jones and Gittins 1972] D.M. Jones and J.C. Gittins. *A dynamic allocation index for the sequential design of experiments*. University of Cambridge, Department of Engineering, 1972.

[Karakovskiy and Togelius 2012] S. Karakovskiy and J. Togelius. The Mario AI benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012.

[Knuth and Moore 1975] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293 – 326, 1975.

[Kocsis and Szepesvári 2006] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, pages 282 – 293. Springer, 2006.

[Kocsis *et al.* 2006] L. Kocsis, C. Szepesvári, and J. Willemson. *Improved Monte-Carlo search*. Technical report, University of Tartu, Estonia, 2006.

[Krauth 1998] W. Krauth. Introduction to Monte Carlo algorithms. In *Advances in Computer Simulation*, pages 1 – 35. Springer, 1998.

[Kuhn 1953] H.W. Kuhn. Extensive games and the problem of information. *Contributions to the Theory of Games*, 2(28):193 – 216, 1953.

[Méhat and Cazenave 2010] J. Méhat and T. Cazenave. Combining UCT and nested Monte Carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271 – 277, 2010.

[Mohri *et al.* 2012] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. MIT Press, 2012.

[Munos and Szepesvári 2008] R. Munos and C. Szepesvári. Finite-time bounds for fitted value iteration. *Journal of Machine Learning Research*, 9(May):815–857, 2008.

[Nau 1982] D.S. Nau. An investigation of the causes of pathology in games. *Artificial Intelligence*, 19(3):257–278, 1982.

[Nau 1983] D.S. Nau. Pathology on game trees revisited, and an alternative to minimaxing. *Artificial intelligence*, 21(1-2):221–244, 1983.

[Osborne 2004] M.J. Osborne. *An Introduction to Game Theory*. Oxford University Press, 2004.

[Puterman 2009] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2009.

[Ramanujan *et al.* 2011] R. Ramanujan, A. Sabharwal, and B. Selman. On the behavior of UCT in synthetic search spaces. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, Freiburg, Germany*, 2011.

[Ramanujan *et al.* 2012] R. Ramanujan, A. Sabharwal, and B. Selman. Understanding sampling style adversarial search methods. *arXiv preprint arXiv:1203.4011*, 2012.

[Rendle 2010] S. Rendle. Factorization machines. In *2010 IEEE 10th International Conference on Data Mining (ICDM)*, pages 995–1000. IEEE, 2010.

[Rimmel *et al.* 2010] A. Rimmel, O. Teytaud, C. Lee, S. Yen, M. Wang, and S. Tsai. Current frontiers in computer Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):229 – 238, 2010.

[Robbins 1952] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527 – 535, 1952.

[Russell and Norvig 2010] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[Shannon 1950] C.E. Shannon. XXII. Programming a computer for playing chess. *Philosophical Magazine Series 7*, 41(314):256 – 275, 1950.

[Sheppard 2002] B. Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1):241 – 275, 2002.

[Silver and Tesauro 2009] D. Silver and G. Tesauro. Monte-Carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 945–952, 2009.

[Silver and Veness 2010] D. Silver and J. Veness. Monte-Carlo planning in large POMDPs. In *Advances in Neural Information Processing Systems*, pages 2164–2172, 2010.

[Silver *et al.* 2016] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484 – 489, 2016.

[Silver 2009] D. Silver. *Reinforcement learning and simulation-based search in computer Go*. PhD thesis, University of Alberta, Edmonton, Canada, 2009.

[Sutton and Barto 1998] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Tesauro and Galperin 1996] G. Tesauro and G.R. Galperin. On-line policy improvement using Monte-Carlo search. In *NIPS*, volume 96, pages 1068 – 1074, 1996.

[Togelius *et al.* 2010] J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 Mario AI competition. In *2010 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2010.

[Tromp 2016] J. Tromp. *Counting Legal Positions in Go*. `https://tromp.github.io/go/legal.html`, 2016. (Accessed 26 July 2016).

[Wang and Gelly 2007] Y. Wang and S. Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games*, pages 175 – 182, 2007.

[Weng and Lin 2011] R.C. Weng and C. Lin. A Bayesian approximation method for online ranking. *Journal of Machine Learning Research*, 12:267–300, 2011.

[Wistuba and Schmidt-Thieme 2013] M. Wistuba and L. Schmidt-Thieme. Move prediction in Go — modelling feature interactions using latent factors. In *KI 2013: Advances in Artificial Intelligence*, pages 260–271. Springer, 2013.

[Zobrist 1970] A.L. Zobrist. *Feature extraction and representation for pattern recognition and the game of Go*. PhD thesis, University of Wisconsin, Madison, WI, USA, 1970.

Appendix A

The Game of Go

Owing to the importance of Go in the development of MCTS techniques, this chapter explains the game of Go itself — its rules as well as some important concepts.

From a theoretical perspective, Go falls into the same class of game as chess. The game can be defined as a turn-based, two-person, zero-sum, deterministic game of perfect information.[1] It is also an extensive-form game and therefore solvable [Osborne 2004]. The major difference, then, between Go and chess is the branching factor and sheer complexity of Go, despite its relatively simple rules.

A.1 Rules of Go

There are a number of variations on the rules of Go, with Japanese and Chinese rulesets being the most popular — however, all variants agree on the same set of general rules. Illustrations are provided on a 9×9 board for conciseness, but apply to boards of any size. The rules listed here are adapted from Davies [1977].

Players

A game is contested by two players, Black and White. Black always moves first, alternating with White thereafter.

[1] Given that Go is not a game of chance, it is interesting to note that the strongest techniques are those based on random Monte Carlo simulations. One might expect these approaches to be more suited to stochastic games or those with imperfect information. In this instance, the stochasticity arises not from the domain itself, but from the agent's uncertainty regarding his own play as well as that of his opponent.

Board

The board (known as the *goban*) is a square grid consisting of a number of horizontal and vertical lines. The intersections of these lines are known as *points*, upon which players place their pieces (*stones*). Two points that are directly connected by a line segment are said to be *adjacent* (Figure A.1). *Connected* stones are those occupying adjacent points.

The size of a Go board is usually 19×19, but other sizes are permissible, with 9×9 and 13×13 being popular alternatives. The game begins with an empty board.

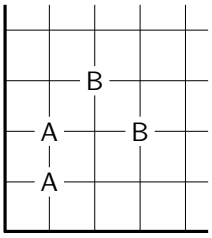
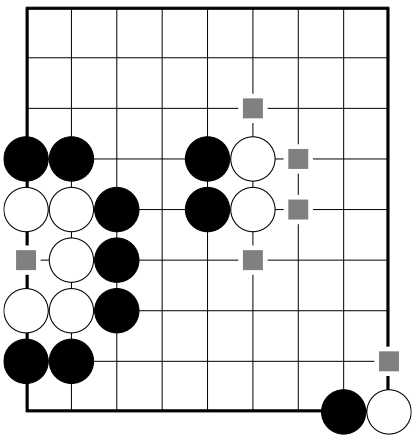


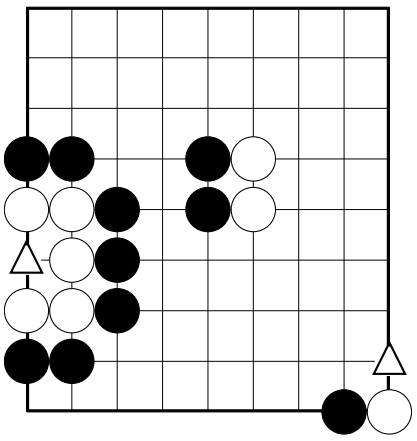
Figure A.1: The points labelled A are adjacent to each other, unlike the points labelled B.

Play

A player moves by placing his stone on an empty point of the board. Players may also choose to pass their move at any time. The number of *liberties* of a stone, or connected group of stones, is given by the number of empty points adjacent to it. Stones with no liberties are captured and removed from the board (Figure A.2).



(a) The marked intersections represent the liberties of the different white groups.

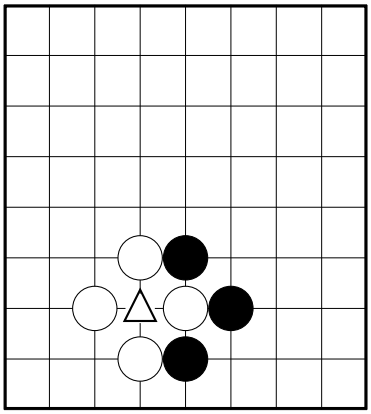


(b) Black can capture white stones by playing on either of the marked points.

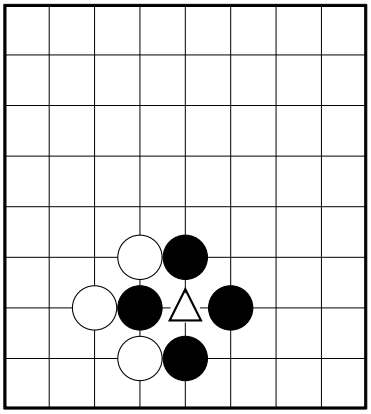
Figure A.2: Illustration of liberties and capture

There are certain restrictions regarding where stones may be placed: many rulesets prohibit moves that result in self-capture (known as *suicide*). Furthermore, no move may be played which

repeats a previous board position. A direct result of this rule is a situation known as a *ko fight*, illustrated by Figure A.3.



(a) Black can capture the white stone by playing at the marked point. This results in position (b).



(b) White cannot immediately capture the black stone by playing at the marked point, as it would result in the original position being repeated. After White plays, Black could then play at the point, winning the *ko*.

Figure A.3: Illustration of a *ko* fight

End of Game

The game ends when both players pass consecutively, at which point their scores are tallied to determine the winner. Two scoring systems exist, the simpler of which is known as *area scoring* and is used in Chinese rules.[2] Under this system, the score of each player is calculated as the number of points occupied by the player's stones and the number of empty points surrounded only by his stones (*territory*). A simple illustration of territory is given by Figure A.4 below. Furthermore, a bonus (known as *komi*) of around 6.5 points[3] is usually added to White's score to compensate for playing second.

[2]The scoring system under Japanese rules is known as *territory scoring* and is slightly more complicated. However, the two systems rarely differ by more than a single point.

[3]The value of *komi* is often a fraction to prevent draws from occurring.

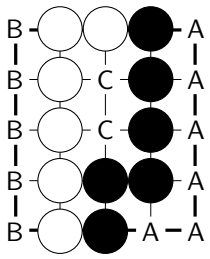


Figure A.4: In this position, Black controls the territory marked A, while White controls the territory marked B. Points marked C are neutral and belong to neither Black nor White.

A.2 Important Go Concepts

While the above rules are relatively simple, the game of Go is not. Many tactical positions and concepts have been analysed by players over the years, some of which are described below to provide an illustration of the strategic depth of Go.

Life and Death

One of the most fundamental aspects of Go is the status of distinct groups of stones. Groups are said to be unconditionally *alive* if they cannot be captured by the opponent, while they are deemed to be *dead* if they cannot escape capture. Groups with two secured internal liberties (*eyes*) can never be captured (unless the player fills these eyes himself) and are thus considered unconditionally alive (Figure A.5).

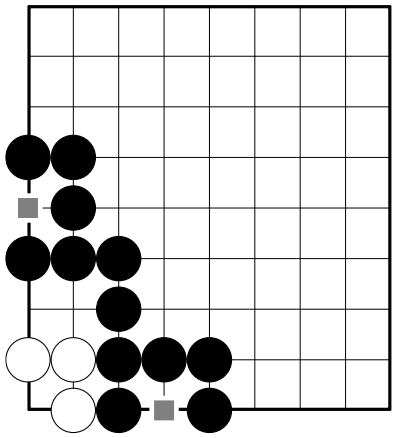


Figure A.5: Illustration of life and death. The white group cannot prevent capture and is thus dead, while the black group has two eyes at the marked points and is therefore alive.

A further variation on this concept is that of *seki* (mutual life). This occurs when a player is unable to capture a group while preventing his own group from being captured. Thus both players' groups are alive, even if they have no eyes. Figure A.6 illustrates this concept.

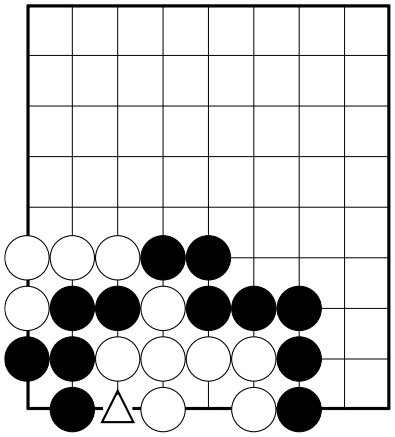


Figure A.6: Illustration of *seki*. Neither Black nor White can attempt to capture the opposing group by playing at the marked point, since it will result in their own group being captured. Thus both groups are alive.

Semeai

A Go concept that has great relevance to computer programs is *semeai* (capture race). Rimmel *et al.* [2010] suggest that, along with *seki*, much work needs to be done to handle *semeai* correctly. Figure A.7 provides a simple example of *semeai* in which both players attempt to capture the other's group first. Despite being fairly trivial to solve, MCTS is unable to do so.

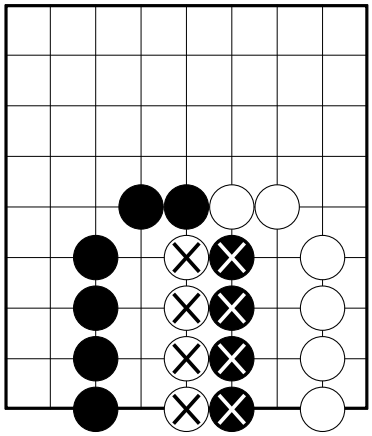
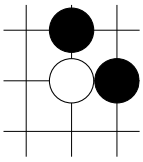


Figure A.7: Illustration of *semeai*. It is clear that the player next to play will win the *semeai* and capture his opponent's stones marked ×. However, MCTS fails to recognise this and assigns a win probability to each player of approximately 50% [Rimmel *et al.* 2010; Huang and Müller 2013].

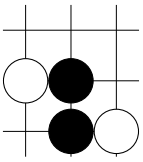
Patterns

Patterns are local configurations of stones and have been used in Go programs ever since their inception [Zobrist 1970]. Patterns may be either location-dependent or -independent, and usually

have a corresponding weight assigned to them. For example, David Silver's RLGO program makes use of a linearly weighted sum of 1×1, 2×2 and 3×3 patterns [Silver 2009].



(a) A simple *hane* pattern.



(b) A *split shape*, which is poor for White.

Figure A.8: Illustration of stone patterns

A.3 Lee Se-dol vs ALPHAGO

Section 2.3 states that moves in Go often have long-term influence, which makes it difficult for an agent to judge the true utility of a move. Unlike games such as chess, it is difficult to perform an accurate evaluation of a static position. To illustrate, consider Game 4 of the recent match between world champion Lee Se-dol and DeepMind's ALPHAGO.

Playing with the white stones, Lee Se-dol is behind when he plays White 78 in an attempt to wrestle territory away from Black at the top of the board (Figure A.9). Black's response is suboptimal, and eventually leads to it surrendering territory. By move 92, White has reversed the game and now leads (Figure A.10). Lee Se-dol is able to maintain this lead throughout the endgame, culminating in his first victory against ALPHAGO (Figure A.11).

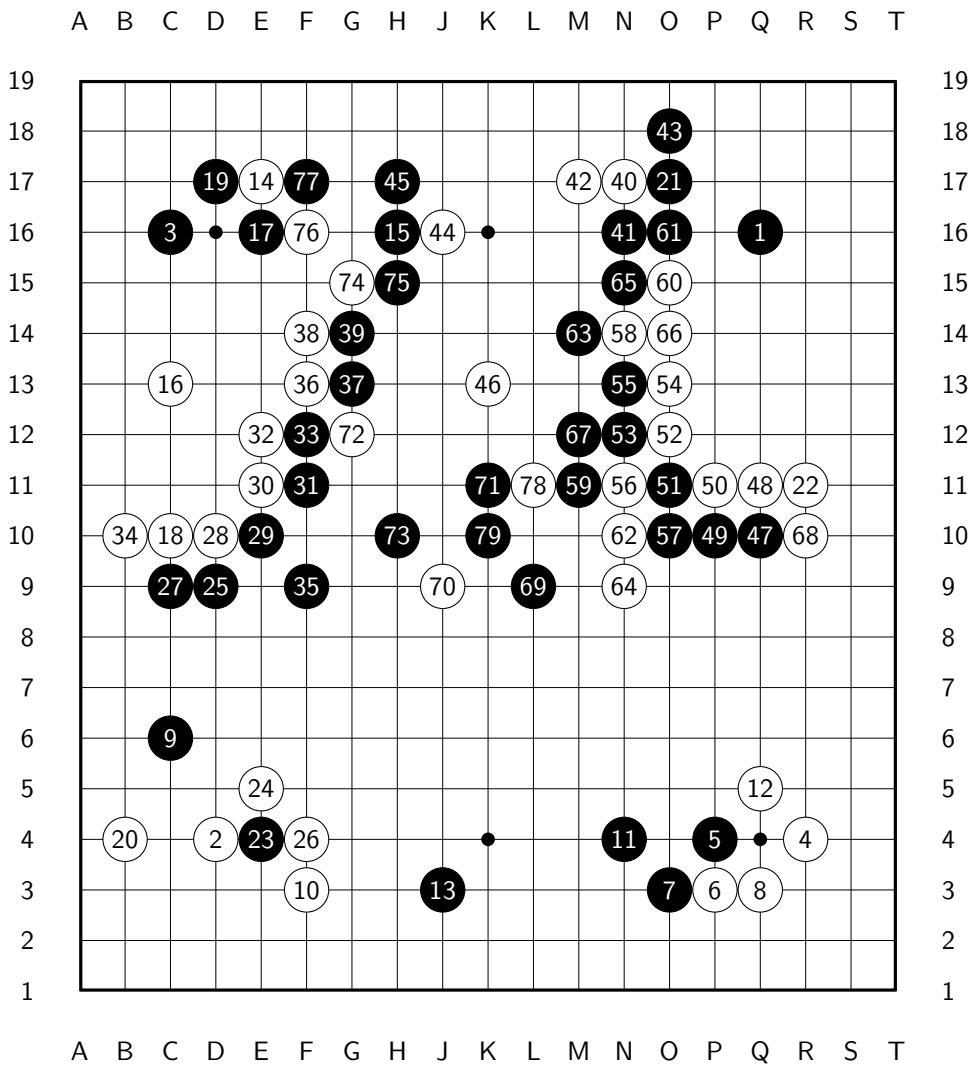


Figure A.9: Position after Black 79. Black should instead have played at L10.

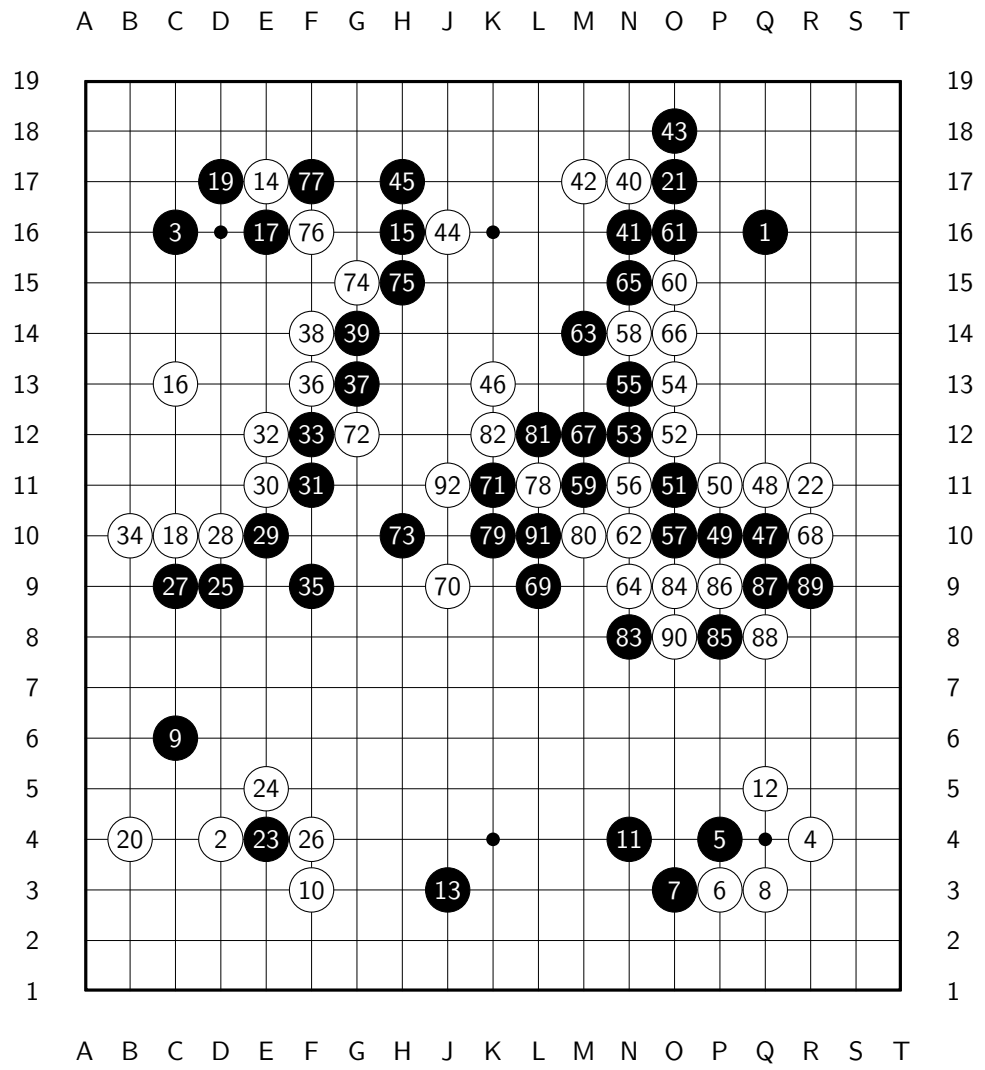


Figure A.10: By White 92, Lee Se-dol has reversed the game and now leads ALPHAGO.

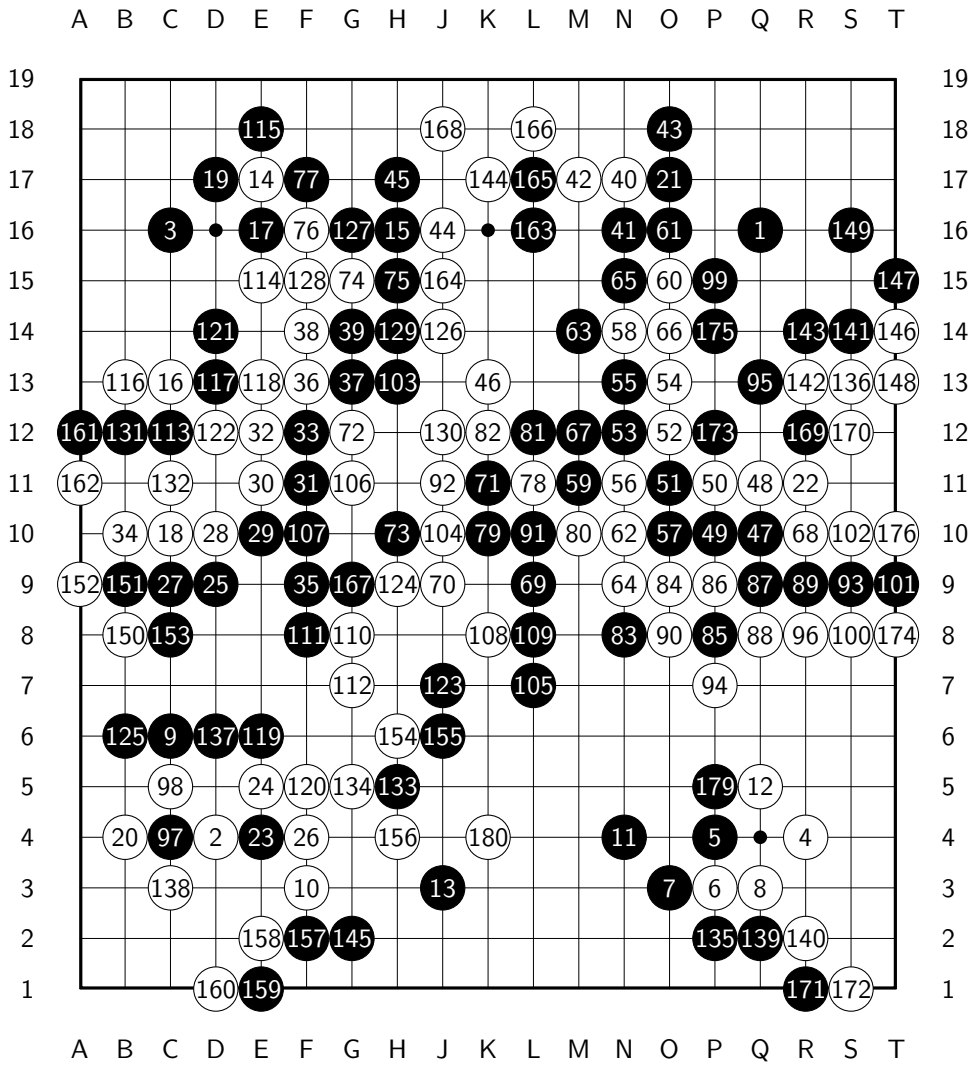


Figure A.11: The final position after Black resigns, handing Lee Se-dol his only victory in the 5-game series.