UNDERSTANDING SAMPLING-BASED ADVERSARIAL SEARCH METHODS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Raghuram Ramanujan

August 2012

UNDERSTANDING SAMPLING-BASED ADVERSARIAL SEARCH METHODS

Raghuram Ramanujan, Ph.D.

Cornell University 2012

Until 2007, the best computer programs for playing the board game Go performed at the level of a weak amateur, while employing the same Minimax algorithm that had proven so successful in other games such as Chess and Checkers. Thanks to a revolutionary new sampling-based planning approach named Upper Confidence bounds applied to Trees (UCT), today's best Go programs play at a master level on full-sized 19×19 boards. Intriguingly, UCT's spectacular success in Go has not been replicated in domains that have been the traditional stronghold of Minimax-style approaches. The focus of this thesis is on understanding this phenomenon. We begin with a thorough examination of the various facets of UCT in the games of Chess and Mancala, where we can contrast the behavior of UCT to that of the better understood Minimax approach. We then introduce the notion of shallow *search traps* — positions in games where short winning strategies for the opposing player exist — and demonstrate that these are distributed very differently in different games, and that this has a significant impact on the performance of UCT. Finally, we study UCT and Minimax in two novel synthetic game settings that permit mathematical analysis. We show that UCT is relatively robust to misleading heuristic feedback if the noise samples are independently drawn, whereas systematic biases in a heuristic can cause UCT to prematurely "freeze" onto sub-optimal lines of play and thus perform poorly. We conclude with a discussion of the potential avenues for future work.

BIOGRAPHICAL SKETCH

Raghuram Ramanujan is a PhD candidate in Computer Science at Cornell University, where he has collaborated with Bart Selman and Ashish Sabharwal on research problems related to algorithms for computer game playing. Prior to Cornell, he was an undergraduate at Purdue University, where he earned a B.S. in Computer Engineering, with a minor in Economics. His interest in Artificial Intelligence was stoked by his undergraduate research work in machine learning and planning, that was carried out under the supervision of Robert Givan and Alan Fern. In the distant past, he spent a couple of years in Singapore completing his GCE 'A' Levels as an SIA Youth Scholar. Outside of academia, he is an accomplished nature and wildlife photographer whose work has been featured on the website of the Cornell Lab of Ornithology and in a field guide to birding in the Finger Lakes region.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the valuable contributions of a number of people. Foremost among them is my advisor, Bart Selman, whose encouragement, patience, understanding and guidance were critical in my development as a researcher. Ashish Sabharwal, a former post-doctoral associate at Cornell and current research staff member at the IBM Watson Center, was a valuable sounding board for ideas, an inspirational role model and a dream collaborator. I am grateful to have had the opportunity of working closely with both of them. I would also like to thank Barbara Crawford and John Hopcroft for agreeing to serve on my thesis committee, and for their valuable input. Going further back, I would like to recognize Robert Givan and Alan Fern from my time at Purdue, for giving me my first taste of research in computer science and starting me on this road.

My teaching experiences in my time at Cornell have been equally fruitful and enjoyable, and have been a significant component of my professional development. Here again, I owe Bart Selman a significant debt of gratitude. During my time spent serving as a teaching assistant for Bart's Artificial Intelligence and Discrete Mathematics courses, he gave me the latitude to experiment and gain experience in course and assessment design. The guest lecturing opportunities in his classes also helped me improve my pedagogical skills. My participation in Barbara Crawford's *Innovative Teaching in the Sciences* course led me to think long and hard about what I wanted from an academic career. My subsequent interactions with Barbara, and David Way of the Cornell Center for Teaching Excellence, have been significant in shaping my career goals. I will always be thankful for the deep conversations on teaching and learning that I shared with the two of them.

I had the fortune of making some wonderful friends in my time in Ithaca, with whom I spent many hours laughing, cooking, climbing, hiking, chasing birds, chasing flying plastic discs, playing Bridge, tasting wine, and simply, talking. To all of you — thank you for making my stay here memorable. Finally, I would like to thank my family for their unwavering support throughout this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Humans have long held a fascination with the idea of imbuing automatons with human-like qualities. For most of human history, this fantasy was consigned to myths like Pygmalion's marble-hewn Galatea, or works of literary fiction like Mary Shelley's Frankenstein and Karel Čapek's Robots. The invention of the digital computer, however, offered the first viable substrate in which experiments to create machine intelligence could be attempted.

The earliest *Artificial Intelligence* (AI) investigators, searching for a challenging domain to drive research, looked to Chess. In fact, attempts to build Chess-playing programs predate the very invention of the term "AI", and modern programmable computers. Alan Turing famously authored one of the world's earliest Chess-playing programs; however, lacking any hardware that could actually run it, he took on the role of "computer" himself, meticulously executing the algorithm on paper to pick moves to play against his colleague and fellow computer scientist Alick Glennie [33][1]. The early adoption of Chess-playing as one of the canonical AI problems is unsurprising. Chess occupies a unique place in human culture among games. Thanks to its rich history and strategic complexity, it is often prescribed in schools as a form of "mental conditioning". The intellectual cachet that is attached to human players who can master the game made the challenge of creating competitive Chess playing programs an irresistible lure for early AI researchers. After all, what could be a better demonstration of "Artificial Intelligence" than outperforming the best humans in this

[1]Incidentally, the match was recreated in June 2012, as part of a celebration of Turing's 100^{th} birthday. This time, Turing's program, running on a digital computer, squared off against former world champion Garry Kasparov. Kasparov won in 16 moves, with the match lasting about 40 seconds [75].

most cerebral of pursuits?

Aside from the intrinsic allure of the problem, the task of designing machines to play Chess (or indeed, any other game) has other attractive properties. Game playing (also referred to as *adversarial search* or *adversarial planning*) is a discrete combinatorial search problem, and as such is well-suited as to digital computers. The state of a game and the set of rules that govern play can often be compactly represented. And yet, the mind-bogglingly large search space of most interesting games renders optimal decision making impractical. Indeed, as Russell and Norvig [85] have observed, games are interesting *precisely* because the search problem is so intractable but we are nevertheless required to make decisions that are "reasonable". If the goal of AI research is to produce machines that exhibit the cognitive sophistication of humans when solving everyday problems in uncertain, dynamic environments, then it is reasonable to begin by focusing our efforts on creating systems that can excel at substantially easier tasks, like games, first.

We also note that algorithmic advances for game playing have wide repercussions. For example, one approach to solving decision making problems in stochastic environments often casts the stochastic element in the role of an adversary. Sometimes termed as *games against nature* [69], such an approach to decision making yields *contingent plans* that guarantee success *regardless* of the outcomes of chance events. This is a common approach to planning in domains like military logistics where the cost of failure is high. From a computational complexity standpoint, generalizations of most commonly played two-player games are PSPACE-complete (and in some cases, EXPTIME-complete) — a class that includes important industrial problems such as hardware debugging and

formal verification. Thus, the development of better algorithms for game playing is likely to produce more immediate real-world payoffs as well.

The *Minimax algorithm*, with alpha-beta pruning, has been *the* approach of choice for building effective game-playing engines for much of the past half-century. It works by reasoning through the "tree of possibilities" that emerges from considering the moves and counter-moves that can be made from a given position. DEEP BLUE, the IBM program that defeated world Chess champion Garry Kasparov in a highly publicized duel in 1997, employed a form of Minimax searching, in combination with a specially compiled knowledge base and fast, custom-built hardware [24]. Despite decades of effort, however, this basic approach which has also been successful in a number of other games such as Checkers, has failed to make much headway in making computers competitive with humans in the game of Go. Two reasons are commonly cited for this failure — the large number of moves that are possible in any given Go position renders looking ahead more than a few levels, in order to analyze the strategic evolution of the game, impractical. Further, expert human knowledge, codified into the form of *static evaluation functions* or *heuristics* have been hard to design for Go.

One alternative approach to planning in such large combinatorial spaces has involved the use of *sampling*. Monte Carlo sampling techniques, in particular, have led to expert-level play in stochastic games with incomplete information such as Bridge [49] and Scrabble [95]. However, they have never outperformed traditional adversarial planning techniques such as Minimax in deterministic two-player games. This changed recently with the emergence of the *Monte Carlo Tree Search (MCTS)* paradigm, and the *Upper Confidence bounds applied to Trees (UCT)* algorithm, in particular [56]. UCT was used to produce the first

program capable of master level play in 9x9 Go [42, 43]. Since then, UCT has also proved promising in other domains such as Kriegspiel that were beyond the scope of any traditional planning techniques [29]. Unlike Minimax, which builds an iteratively-deepening search tree that examines all possible lines of play (up to some depth) in a systematic way, UCT grows a highly asymmetric, sparse tree in an opportunistic fashion, focusing on the most promising lines of play from a given state. In addition, UCT does not require a domain-specific heuristic to perform well, a property that has made it an attractive proposition for the task of general game playing [36]. Instead, it relies on so-called *random playouts* — random completions of games from a given position — to estimate the strength of positions. What's more, it accomplishes all this while still guaranteeing that the probability of making a sub-optimal decision approaches zero in the limit.

1.1 Contributions

Unfortunately, in contrast to the extensively studied Minimax search algorithm, the success of UCT is currently not well understood. The process of determining whether it is likely to succeed in any given domain is mostly a process of trial-and-error. Moreover, anecdotal observations have found that the set of domains where Minimax and UCT shine is largely disjoint; for any given search space, one or the other works well, but seldom both. This thesis presents work that sheds light on the reasons why these different approaches to adversarial search have such complementary strengths. In particular, we make the following technical contributions:

- In Chapter 3, we resolve some of the speculative claims that have been

made about various facets of UCT, and its performance in certain domains such as Chess, via a series of carefully designed experiments.

- In Chapter 4, we introduce the notion of *search traps* and use this idea to offer an explanation for UCT and Minimax's diverging results in Chess and Go.
- And finally, in Chapter 5, we complement our study of UCT and Minimax in real-world games, with a study in the context of synthetic games. This approach, in addition for allowing for more controlled experimentation, also allows us to derive insights via mathematical analysis.

The research presented in this paper was carried out in collaboration with Ashish Sabharwal and Bart Selman, and sections of the work have been published as part of the proceedings of the 21^{st} and 22^{nd} *International Conference on Automated Planning and Scheduling* [79, 81] and the 26^{th} *Conference on Uncertainty in Artificial Intelligence* [80][2]. We begin the rest of this thesis with a survey of the AI literature related to the problem of game playing.

[2]We would like to gratefully acknowledge the following sources of financial support our research has received: NSF Expeditions in Computing award for Computational Sustainability, 0832782, NSF IIS award 0514429 and IISI, Cornell University AFOSR grant FA9550-04-1-0151.

CHAPTER 2

BACKGROUND

In this chapter, we provide the necessary background for understanding the work and results that follow in subsequent chapters. We begin by defining some basic terms in Section 2.1. In Section 2.2, we trace the evolution of AI agents in a variety of popular games and survey the current state of the playing field. Sections 2.3 and 2.4 describe what is "under the hood" of state-of-the-art programs for playing Chess and Go respectively. We conclude with a brief discussion contrasting the computational approach to these two games in Section 2.5, and provide the motivation for the research efforts described in the remainder of the thesis.

2.1 Basic Terminology

The *planning* problem in AI is a search problem that is concerned with determining a sequence of actions that enable an agent to achieve a goal, or maximize its reward. In *multi-agent planning*, the task is complicated by the presence of other agents in the environment, who may or may not be friendly. *Adversarial planning*, or *game playing*, involves planning in settings where agents are actively attempting to thwart each other. We paraphrase Russell and Norvig [85] who define a game as a search problem with the following elements:

- an initial state, that specifies how the game is set up at the start,
- a model that specifies the rules of the game and their outcomes,
- a terminating condition that specifies when the game ends[1], and

[1] All games we consider here are finite and assumed to terminate

- a reward function that specifies the payoffs for each player at termination.

Games that have been commonly studied in AI can be categorized along several axes:

Number of players Games may involve one player (also known as *puzzles*), two players or three or more players.

Payoff structure A *zero-sum* (also known as *constant sum*) game is one where the net benefit to all the players in the game is zero; in other words, the winning player benefits by the amount of the combined expense of the others. In *non-zero sum* games, outcomes may yield net benefits greater or less than zero.

Determinism A *deterministic* game is one where actions have predictable outcomes; a *non-deterministic* or *stochastic* game is one that includes elements of luck, introduced via mechanisms such as die rolls or card deals.

Observability A *fully observable* game is one in which there is no hidden information and the complete state of the game is visible to all players at all times. In contrast, in a *partially observable* game, certain aspects of the state are hidden, such as the cards an opponent is holding.

Simultaneity In a *simultaneous* game, players make their moves at the same time. In contrast, a *sequential* game is one where players take turns making moves.

This thesis is primarily concerned with methods for planning in two-player, zero-sum, deterministic, fully observable, sequential games, though we will occasionally take diversions to examine other kinds of games when relevant.

We now introduce some common terms that are used in the AI literature on adversarial search. The *state space* of a game is the set of all legal positions that are reachable from the starting position. It can be visualized as a *game tree*, where each *node* represents a position (or *state*) in the game and each *edge* corresponds to a legal move. The *root node* of the tree represents the initial position. A *path* in the tree is a sequence of edges such that each adjacent pair of edges share a node, i.e., a sequence of edges $(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)$ represents the path from node n_1 to n_k. The *ancestors* of a node n are the nodes that occur on the path from the root node to n, and n is a *descendant* of its ancestors. The closest descendants and ancestor of a node are its *children* and *parent*, respectively. The *depth* of a node n is the length of the path from the root node to n. It is measured in *plys* (or *half-moves*, or *levels*). A *terminal position* is one where the game has ended and the outcome is known; the corresponding node is referred to as a *terminal node*. Nodes that are not terminal are referred to as *internal* nodes.

The *search tree* is the portion of the game tree that is examined by the search procedure. The search tree is built up in discrete steps, by *expanding* nodes, which has the effect of adding one or more children of the current node to the tree. Nodes in the search tree that do not have any children are *leaf nodes* — they are either terminal, or have not been expanded yet. The *subtree* rooted at a node n is a tree composed of the descendants of n, and all the edges connecting them[2]. The *search depth*, or *lookahead depth*, is the depth of the deepest leaf node measured in plys.

[2]It will be apparent from the context whether we are referring to a subtree of the *search tree* or the *game tree*

2.2 The Current Status of Computer Game Playing

In this section, we offer a brief history of the evolution of computer agents in a variety of popular games. Table 2.1 summarizes the current strength of computer play in these domains.

Connect Four

Connect Four is a two-player game that belongs to the family of connection games, which includes others such as Tic-Tac-Toe and Hex. Each player assumes a disc color; they then proceed to take turns dropping discs into a vertical grid that is seven columns wide and six columns high. Under the force of gravity, the discs come to rest in the highest unoccupied cell of the selected column. The objective of the game is to be the first player to connect four of one's own discs either horizontally, vertically or diagonally. Connect Four is *weakly solved* [4] — the outcome of the game from the initial position was exhaustively analyzed and determined to be a win for the first player by James Allen and Victor Allis in 1988. The game was then *strongly solved* in 1995 by John Tromp [107]; computer programs today are therefore capable of perfect play on standard sized Connect Four boards from *any* starting configuration.

Checkers (English Draughts)

Checkers is a classic two-player board game played on an 8×8 squared board. Each side begins with twelve identical pieces that can only be moved diagonally. They can capture the opponent's pieces by jumping over them. The game ends when either player has no remaining legal moves (or neither player can force a win). Checkers was among the earliest challenge problems tackled by AI researchers owing to its relative

simplicity. Arthur Samuel's Checkers program from 1959 led to genuine optimism among many AI researchers. It combined limited lookahead search with a heuristic function that was learned via self-play to achieve a respectable playing strength, comparable to that of solid amateur players [87]. By the early 90s, computer Checkers players were challenging the best humans. In 1992, the CHINOOK program contested Dr. Marion Tinsley for the world championship title, but lost by a margin of 4-2 [92]. Notably, these were two of only seven defeats Dr. Tinsley had suffered in over 40 years of prior competition [90]. After a 1994 rematch, CHINOOK was crowned the world champion. More recently, after almost two decades of non-stop computation, the game of Checkers was weakly solved [91]. It was discovered that perfect play by both sides from the beginning of the game leads to a draw.

Backgammon

Among the oldest board games in the world, Backgammon is a stochastic two-player game where player moves are influenced by die rolls and each side attempts to eliminate the other's pieces. Despite featuring a substantial luck component, Backgammon is ultimately a game of skill; the better player will win over a large sample of games. One of the earliest successful computer Backgammon players was Hans Berliner's BGK 9.8 system. Indeed, the first ever defeat of a human world champion in any board game occurred in 1979, when Luigi Villa was defeated by a points margin of 7-1 by BGK 9.8 [16]. However, the optimism generated by the outcome was tempered by the fact that Villa had been unlucky with the die rolls, and the match had only featured a small number of games. In the early 90s, Gerald Tesauro's TD-GAMMON created a far bigger splash by losing

narrowly to some of the world's top Backgammon players (including several former world champions) over a much larger set of games [104, 105]. Strikingly, the program's high level of performance was attained by making it play hundreds of thousands of training games against itself. Using temporal difference learning [103], these game traces were used to tune the weights of a multi-layer neural network that served as a heuristic positional evaluator. Interestingly, the positional judgement of TD-GAMMON was so good that very little lookahead search (2 plys or less) was used. State-of-the-art computer players today, such as the freely available GNU BACKGAMMON[3], play at a level that is superior to the best human players and are widely used to analyze and improve human play.

Mancala (Kalah)

Mancala is an ancient family of two-player games that is popular in many parts of Asia and Africa. Though there are many rule variations, the game is usually played on a board consisting of two rows of pits. Players take turns picking up stones from pits on their side of the board and "sowing" them in adjoining ones, with the objective of capturing more stones than their opponent. Thanks to its relatively simple rules and small search space, Mancala has received a significant amount of attention from the AI community. In 1968, Alex Bell designed an early system for playing the game [13]. In 1970, the game was used as a test domain for investigating the efficacy of the newly proposed M&N algorithm [98]. More recently, the game was weakly solved for a variety of board sizes and starting configurations; the first player was discovered to hold the advantage in a majority of cases [51]. Thus, computer Mancala agents today are capable of flawless play.

[3]Available to download at `http://www.gnu.org/software/gnubg`

Othello (Reversi)

Reversi is a two-player strategy board game played on an 8×8 grid. Players take turns placing their colored stones on the cells of the board with the objective of holding more stones at the end of the game. Stones can be captured by bracketing a "span" of the opponent's stones with one's own. Over the course of a game, there are typically a large number of capture and counter-capture moves that render the state of the board highly dynamic and changeable. This limits the extent to which humans can effectively plan ahead. Computer players have thus enjoyed an advantage over humans in Othello since the 80s [57]. Michael Buro's LOGISTELLO [21, 22], that combines efficient lookahead search techniques, an extensive opening book database, and a sophisticated evaluation function (tuned using machine learning techniques), is among the best known computer Othello players. In one of the most one-sided man-machine encounters, LOGISTELLO defeated the reigning world champion Takeshi Murakami by a margin of 6-0 in 1997 [22]. Modern day computer Othello programs play at a level far superior to that of the best human players.

Bridge

Contract Bridge, or simply Bridge, is one of the world's most popular card games with millions of aficionados worldwide. Like most card games, it is a stochastic game of imperfect information played by two teams of two players each. In the first phase of the game, teams bid to win a "contract" that defines a trump suit and a declaration of how many tricks they expect to win. The side that wins the contract is termed the "declaring side". In the second phase, play proceeds in a similar fashion to other trick-taking games like Hearts or Whist, with the caveat that one player on the declar-

ing side plays with her cards face up. Initial Bridge playing programs adopted a knowledge-based approach, trying to recognize the class into which a deal fell and playing accordingly. This was supplemented with ideas from work on Hierarchical Task Network (HTN) planning in the BRIDGE BARON program that won the World Bridge Computer Challenge in 1997 [99]. Matt Ginsberg's GIB [49] introduced a number of innovative techniques and became the first program to achieve expert-level play in Bridge in 1998. GIB makes use of Monte Carlo simulations in its card play procedure: it generates hypotheses about the possible distribution of unseen cards that is consistent with the history of the game and computes the best line of play for each hypothesis using a method known as *partition search*. The action that yields the most positive outcome, among all hypothetical deals, is then played. GIB used a similar simulation-based approach, combined with a database of rules, in the bidding phase as well; however, bidding is a more subtle skill, and this aspect of the game was deemed to be GIB's main weakness. Nevertheless, GIB (and other modern-day successors) can hold its own against expert human players.

Scrabble

Scrabble is a word game in which players score points by placing lettered tiles on a board divided into a 15×15 grid, forming words in a crossword-like fashion. Players draw tiles at random from a bag to form their rack and cannot see the tiles drawn by their opponents; as a result, Scrabble is a stochastic game of imperfect information. While the rules permit two to four players, computer Scrabble program development has focused on the two-person case. Brian Sheppard's MAVEN is the preeminent computer Scrabble player, having consistently defeated world champion opponents

since 1998 [95]. The program uses short simulations (at most 4-plys deep), combined with 1-ply lookahead, to make its move decisions in the early and middle phases of the game. In the endgame, once all the tiles in the bag have been exhausted, Scrabble becomes a game of perfect information; in this phase MAVEN employs B^* search [15] to guide its decision making, rather than an exhaustive alpha-beta style search due to the extremely large branching factor. Given the significant advances in computing power over the last 14 years, it is believed that today's best human players are no longer a match for the best computer players.

Chess

Chess occupies a unique position in human culture — thanks to its long and rich history and the complexity of its gameplay, it is widely treated with a respect accorded to few other games. Benjamin Franklin famously declared Chess to be more than "idle amusement", and a form of mental training [38]. Chess' association with high levels of cognitive ability thus made it a perfect test-bed for early work in machine intelligence. The first algorithms for playing Chess were described by Alan Turing [109] and Claude Shannon [94] before programmable computers capable of implementing them even existed. The imprint of Shannon's ideas, in particular, can still be seen in the architecture of most modern Chess engines. Major progress in computer Chess occurred in the 70s, spurred by competitions such as those organized by the Association for Computing Machinery (ACM). The initial contests were dominated by the CHESS 4.X series of programs from Northwestern University, developed by David Slate and Larry Atkin [8]. This influential family of programs introduced a number of novel ideas such as the use of bitboards, transposition tables and

iterative deepening alpha-beta search. CHESS 4.5 became the first computer program to win a human Chess competition in 1976. In 1988, the DEEP THOUGHT system from Carnegie Mellon, running on specialized hardware and using clever selective search approaches to augment the basic alpha-beta procedure, became the first program to defeat a human Grand Master [6]. In 1997, its successor DEEP BLUE defeated the reigning world champion Garry Kasparov by a margin of 3.5-2.5 [24, 65]. While the DEEP BLUE system incorporated numerous state-of-the-art algorithmic techniques, the biggest factor in its success was the hardware it employed. Running on 480 specially designed chips, DEEP BLUE's ability to carry out a highly parallelized brute-force search was unprecedented — it could examine up to 200 million positions per second. Today, thanks to further growth in computing power and even better heuristics, Chess engines such as RYBKA and HOUDINI play at a level superior to that of the best humans, while running on standard desktop hardware.

Go

The game of Go, which is hugely popular in Asia, is perhaps second only to Chess in the amount of interest it has generated in the AI community. One of the key drivers for this interest in Go has been its extreme intractability — the game is played on a 19×19 grid, with players taking turns placing stones of their chosen color on one of the intersections on the board. This results in a large branching factor at each level of the game tree which renders Minimax-style lookahead search infeasible. Moreover, heuristics have been notoriously difficult to construct for Go since many moves have extremely delayed impact on the state of the game (unlike in, say Chess, where features such as piece counts can be a useful proxy for

the impact of a move) [19, 70]. As a result, the standard of computer Go play languished at the level of a weak amateur as recently as 2007, despite decades of effort [44]. The development of so-called Monte-Carlo Tree Search (MCTS) techniques suddenly changed this landscape; two trail-blazing programs, CRAZY STONE [31] and MOGO [42, 43, 45], demonstrated the utility of this new approach. The latter, based on an MCTS algorithm named Upper Confidence bounds applied to Trees (UCT) [56], went on to become the first program to achieve master level play and beat a human professional at 9×9 Go. Today's top Go programs such as ZEN all employ MCTS approaches, and are competitive with top professional players on 9×9 boards and capable of winning 19×19 games against professionals with modest stone handicaps [44].

Poker

Alongside Go, Poker is widely considered to be the "final frontier" in research in planning in adversarial domains. A stochastic game of imperfect information, expert poker play requires mastery of a number of skills: deception, opponent modeling, unpredictability and a firm grasp of probability, a set of demands that is pretty unique among popular human games. Early approaches to computer Poker playing used knowledge-based and simulation-based approaches, but these met with little success [18]. Today's dominant Poker bots use game-theoretic approaches that find optimal strategies to approximations of the original game [52]. In the Heads-Up (two-player) Limit Texas Hold'Em variant of Poker, computer agents are now on-par with world-class professional players [52]. The gap between man and machine is however substantial in the more strategically rich No-Limit and multi-player variants, where the best programs

Game	Game Tree Complexity[4]	Standard of Computer Play
Backgammon	—	Better than the best humans
Bridge	—	Comparable to human experts
Checkers	$\sim 10^{31}$	Infallible from initial position
Chess	$\sim 10^{123}$	Better than the best humans
Connect Four	$\sim 10^{21}$	Infallible from any position
Go (19×19)	$\sim 10^{360}$	Comparable to advanced amateurs
Mancala	$\sim 10^{18}$	Infallible from initial position
Othello	$\sim 10^{58}$	Better than the best humans
Poker	—	Comparable to professional players[5]
Scrabble	—	Better than the best humans

Table 2.1: The current standard of computer play in some popular games. For deterministic games, we also list their estimated game tree complexity [111]

are beaten by advanced amateurs.

2.3 The Anatomy of Chess Playing Programs

In this section, we present an overview of some of the common techniques that are used in state-of-the-art Chess playing engines.

[4]The *game tree complexity* is the minimum number of leaves that would need to be examined to prove the Minimax value of the root node of a game tree [5]

[5]This is for the 2-person, Limit Texas Hold'em variant. Computers are significantly weaker than humans in the No-Limit and multi-player settings.

2.3.1 The Minimax Algorithm

The *Minimax theorem*, first proven by John von Neumann in 1928 [113, 114], is a fundamental result in game theory. It states that in two-person, zero-sum games with a finite action space, there exists a value v such that:

1. the first player can guarantee himself a payoff of v, irrespective of the second player's strategy, and
2. the second player can guarantee himself a payoff of $-v$, irrespective of the first player's strategy.

While both players strive to maximize their own payoff (or equivalently, minimize their opponent's), it is traditional to label one player as the maximizing player (henceforth, Max) and the other as the minimizing player (Min). The value v is referred to as the *Minimax value* of the game, and represents the payoff that would be awarded to the victor of the game (and the penalty assessed of the loser) if both players were to act optimally. This outcome is also referred to as the *Nash Equilibrium* — a scenario in which neither player stands to gain anything by unilaterally changing their strategy [71].

In games like Chess where players alternate turns, the optimal first action for the player on move from a given position s — indeed, the entire sequence of optimal moves by both players from s that leads to the equilibrium outcome — can be computed in a recursive fashion using the *Minimax algorithm* [85]. Intuitively, the algorithm determines the best action for the player on move at the initial position (assumed to be Max) by a process of reasoning that goes as follows: *"What move can Max initially make, such that regardless of Min's response,*

Algorithm 2.1: The Minimax algorithm

```
1:  procedure MINIMAX(state, depthToGo, side)
2:      if state is terminal or depthToGo == 0 then
3:          return EVALUATE(state)
4:      end if
5:      if side == Max then
6:          score ← −∞
7:          for each successor s′ of state do
8:              eval ←MINIMAX(s′, depthToGo − 1, Min)
9:              score ← max(score, eval)
10:         end for
11:     else
12:         score ← ∞
13:         for each successor s′ of state do
14:             eval ←MINIMAX(s′, depthToGo − 1, Max)
15:             score ← min(score, eval)
16:         end for
17:     end if
18:     return score
19: end procedure
```

Max can make a counter-move, such that regardless of Min's response to that counter-move, . . ., such that in the end, Max's payoff is maximized?" This is accomplished by expanding the entire game tree in a depth-first fashion, starting at position s. The outcome of each possible line of play, which is given by the value of the terminal node reached (for example, −1 representing wins for Min, +1 for wins for Max and 0 for draws), is then backed up all the way to the root node, while assuming that players pick the move that is most beneficial to them every step of the way. The move leading to the position with the best backed-up value (i.e., the Minimax value) is then the optimal move at position s. Unfortunately, the number of positions examined by this procedure grows exponentially; in a

game with a constant branching factor of b that lasts d plys, this naïve algorithm will expand b^d nodes, which is impractical for all but the most trivial games. In games like Chess, Minimax searches are thus truncated at some maximum depth and a *heuristic evaluation function* is applied at the leaf nodes of this search tree. These leaf value estimates are treated as if they were "true" Minimax values and are backed-up as before, to guide decision-making at the root node. The pseudocode for this depth-bounded search procedure is shown in Algorithm 2.1. The procedure is invoked by supplying the starting state, the side on move (Max or Min) and a maximum lookahead depth (*depthToGo*). Figure 2.1 demonstrates the outcome of this procedure when applied to a mid-game position in Tic-Tac-Toe.

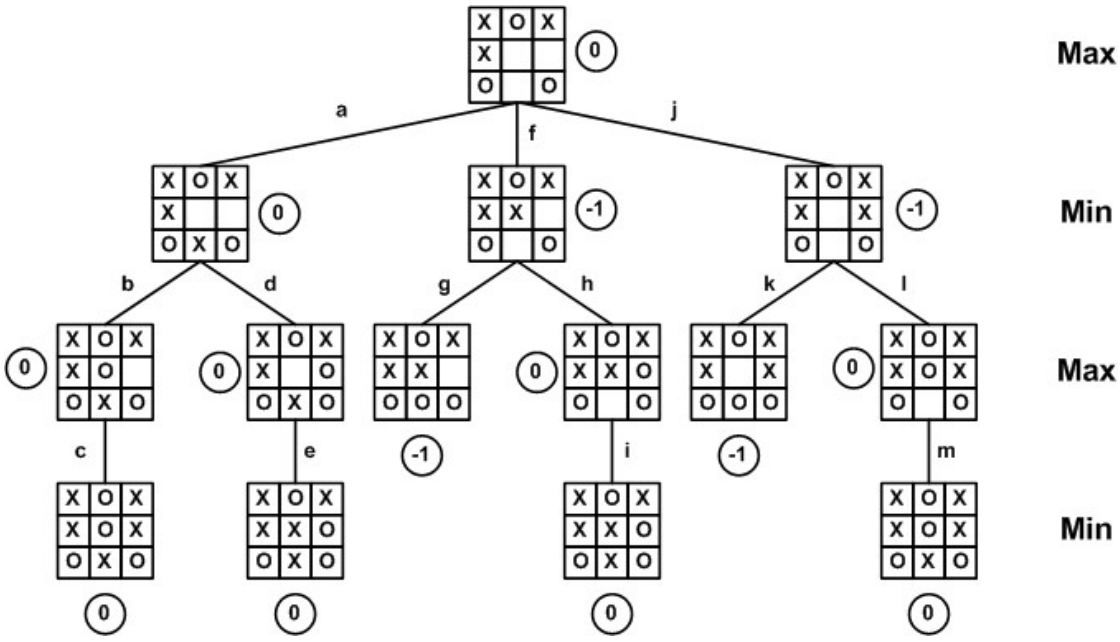

Figure 2.1: Tree labeling produced by executing Algorithm 2.1 on the given Tic-Tac-Toe position. The order of edge expansion can be inferred from the alphabetic ordering of edge labels.

It is unsurprising then that the performance of practical game-playing programs is heavily dependent on how well the heuristic function approximates the true utility of a state. Given that modern Chess engines typically examine many millions of positions to make a single move, this heuristic function must be fast to compute. A typical approach to heuristic construction is to first calculate *features* of the state description — in Chess, this may be concepts like the material imbalance, whether castling has occurred, whether pawns are blocked and so on. The individual contributions of these features are then combined in a weighted fashion to produce an overall score for a given position. Unfortunately, this design process is more of an art than an exact science, requiring heavy input from domain experts. DEEP BLUE, for example, used about 8000 features in its evaluation function [24], with the weights requiring extensive hand-tuning. Automatic methods for tuning weights have produced some promising results — variants of TD-learning, employed in the programs KNIGHTCAP [11] and MEEP [112], and genetic algorithms used in BLONDIE25 [37], have produced programs that play Chess at the master level. However, these are still no match for the hand-crafted heuristics used in the top-of-the-line programs. Modern programs also supplement the stock heuristic function with *opening books* and *endgame databases*. These help with planning in the early and late stages of the game, where static evaluation of positions is harder, but a large body of knowledge is readily available thanks to centuries of intensive human study [64].

2.3.2 Alpha-beta Pruning

Alpha-beta pruning is an enhancement to the basic Minimax procedure that significantly improves the runtime of the algorithm, while preserving its optimality guarantee. First proposed by John McCarthy in 1955, and independently rediscovered several times since [74, 64, 83], it exploits the fact that there are often large parts of a search tree that have no impact on the Minimax value of the root node because the positions they represent will never be reached in optimal play. These branches can therefore be safely *pruned*, i.e., ignored without explicit expansion.

The pseudocode for alpha-beta enhanced Minimax search is presented in Algorithm 2.2. The procedure is initially invoked with parameters α and β — which form bounds on the Minimax value of the root node — set to $-\infty$ and $+\infty$ respectively. The intuition behind how pruning decisions are made is best understood via an example. Figure 2.2 presents the same Tic-Tac-Toe position as in Figure 2.1, after it has been searched using the alpha-beta procedure, with grayed-out portions representing the pruned sections of the tree. We will trace through the execution of the algorithm starting from the moment after the expansion of edge g. At this point, Max is already aware that move a has a pay-off of 0 for him (assuming our depth-first search expands nodes from left-to-right). After the expansion of node g, he can further deduce that were he to make move f, then Min, by making counter-move g, can ensure that Max makes *at most* -1. Thus, Max can safely make the decision to ignore the remainder of this subtree, since he cannot expect to obtain a better payoff than he can gain from making move a (assuming Min plays optimally). By a similar argument, Max can prune moves l and m as well.

Algorithm 2.2: The Minimax algorithm with alpha-beta pruning

```
1:  procedure ALPHABETA(state, depthToGo, side, α, β)
2:      if state is terminal or depthToGo == 0 then
3:          return EVALUATE(state)
4:      end if
5:      if side == Max then
6:          for each successor s′ of state do
7:              α ← max(α, ALPHABETA(s′, depthToGo − 1, Min, α, β))
8:              if β ≤ α then
9:                  break
10:             end if
11:         end for
12:     else
13:         for each successor s′ of state do
14:             β ← min(β, ALPHABETA(s′, depthToGo − 1, Max, α, β))
15:             if β ≤ α then
16:                 break
17:             end if
18:         end for
19:     end if
20:     return score
21: end procedure
```

The first formal analysis of this algorithm was carried out by Knuth and Moore [55]. They showed that in the best-case scenario, alpha-beta pruning will only expand $O(b^{d/2})$ nodes[6], reducing the effective branching factor of the game from b to $\sqrt{b}$. Judea Pearl later proved that alpha-beta was asymptotically optimal, in that no complete search strategy can achieve a lower branching factor.

[6] In the worst-case, alpha-beta pruning will expand as many nodes as a full-width Minimax search.

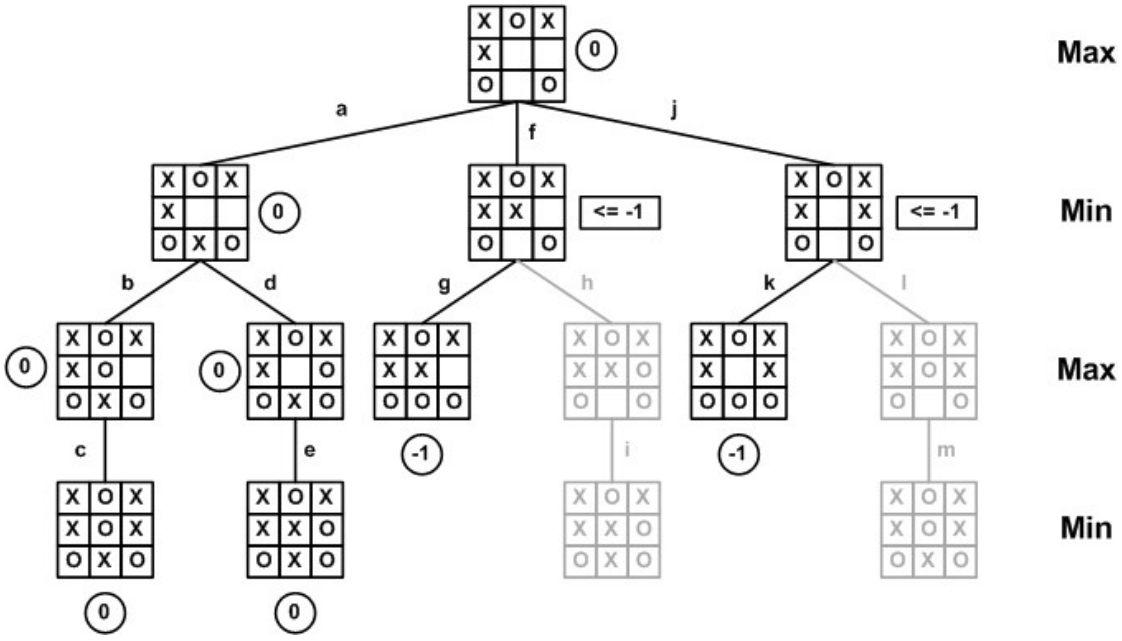

Figure 2.2: Tree labeling produced by executing Algorithm 2.2 on the same Tic-Tac-Toe position as in Figure 2.1. The grayed-out portions represent the parts of the tree that are pruned.

Move Ordering

The effectiveness of alpha-beta pruning is heavily dependent on the order in which moves are examined. The maximal reduction in branching factor is achieved when the best moves are examined first at every level of the game. Thus, a significant component of modern Chess programs is a procedure for sorting moves. There are two common approaches:

Static Ordering

These methods use domain-dependent heuristics to determine what moves to try first. For example, in Chess, checking moves, captures and promotions often produce large swings in the positional evaluation. They are more likely to cause alpha or beta cutoffs that allow for pruning, and

are usually tried first before other moves.

Dynamic Ordering

These methods sort moves on the basis of information uncovered as the search proceeds and are domain independent. Two common heuristics here are the *killer heuristic* [48] and the *history heuristic* [89]. Both these methods work on the assumption that moves that cause cutoffs in one part of the search tree are likely to cause cutoffs elsewhere, and should therefore be tried early. During the search process, a table of these potential "killer" moves is maintained and updated. The difference between the two is that the former tries moves that were previously found to be effective at the same search depth, whereas the latter rewards moves that cause deeper rather than shallower cutoffs.

Narrow Window Searching

The size of the interval between the alpha and beta values is termed the *search window* and it plays a crucial role in determining the runtime of alpha-beta pruning. When a search is carried out with a narrow search window, it induces a lot of pruning and the procedure terminates quickly; unfortunately, the typical outcome of such a search only indicates whether the search *failed high* (i.e. the Minimax value is greater than β) or *failed low* (the Minimax value is less than α). A wide window search can compute the exact Minimax value of a node, but will induce fewer cutoffs and take longer to complete. However, if a reasonably good move ordering scheme is available, the following approach can be used to balance these two extremes: run a wide window search on the first move to evaluate its utility perfectly, but run narrow window searches on the remainder to

simply verify that they result in worse outcomes. If the narrow window search fails in the wrong direction (i.e., a later move is found to be better than the first move), then a search with a full window is performed. Many algorithms based on variations of this idea have been developed over the years, including Scout [76], aspiration search [8], Negascout [82] and Principal Variation Search (PVS) [66]. The latter two — which are equivalent — take the narrow window search to the extreme, by searching the first move at each level (the *principal variation*) with a full window of $[-\infty, +\infty]$, and all subsequent moves with a *null window* of $[\alpha, \alpha + 1]$ at Max levels, or $[\beta - 1, \beta]$ at Min levels. Negascout and PVS are the most widely used search algorithms in modern Chess engines.

Alternatives to Alpha-Beta

Alternative approaches to depth-first alpha-beta style search have also been studied down the years. George Stockman's SSS* [100] and Hans Berliner's B* both utilize a *best-first* approach to search tree expansion. The former, in particular, was shown to consistently expand smaller trees than vanilla alpha-beta search; however, the space and time overhead associated with having to maintain a sorted list of frontier nodes proved to be a poor trade-off, when compared to the savings in tree size [84]. Another novel best-first search approach was David McAllester's *conspiracy number search* [67, 68], which expanded the search tree in a fashion that stabilized the estimated utility of the root node. One benefit of this approach is that it naturally searches *forced lines of play*, where static evaluations tend to be unstable, to a deeper level without any domain engineering. Nevertheless, conspiracy search has not produced agents that are competitive with traditional PVS-based programs. Finally, we would like to note the work

of Pearl [77] and Dana Nau [73] who studied the potential of *probabilistic* backup strategies, based on the product rule. This was shown to be particularly useful in certain artificial games where traditional Minimax search suffered from the so-called *lookahead pathology* — a phenomenon where searching deeper (without encountering the end of the game) degrades performance. However, these techniques do not scale well to real-world games. *Sampling-based search* techniques offer a completely different way of tackling the task of planning in combinatorially vast spaces; these approaches are discussed in greater detail in Section 2.4.

2.3.3 Search Control Techniques

A number of additional techniques are used by modern Chess programs to use the time available to them effectively and to search in a "smarter" manner. In this section, we review three of these techniques — *iterative deepening search*, *forward pruning* and *selective search extensions*.

Iterative Deepening

One key drawback of fixed depth alpha-beta search (and its variants), as presented in Algorithm 2.2, is the unpredictability of its runtime. This is particularly problematic in games like Chess where strict time controls are employed, since the search procedure may not complete in the allotted time. Alpha-beta search is thus combined with *iterative deepening*, to make the algorithm "any-time" [8]. Starting with a 1-ply search, the current position is repeatedly searched with deeper depth bounds as each previous search terminates, until time expires, i.e., from the current position, the program carries out a 1-ply

search, a 2-ply search, a 3-ply search and so on. Due to the exponential growth in the size of the trees being searched on each step, the overall run-time of iterative deepening depth-first search is dominated by that of the deepest search, and the runtime overhead compared to a single, deep search is not significant. In fact, using the outcome of shallow searches to order moves for deeper searches often means that the iterative deepening search terminates faster (or can search deeper).

Forward Pruning

The pruning methods we have encountered so far have all been sound, in that they provide the same result as a full-width Minimax search. In his seminal paper from 1950, Shannon termed this brute-force approach 'Type A' [94]. In the same paper, he described another approach one could take to programming a Chess playing computer — the 'Type B' programs. Rather than explore *every* possible move at each level of the search tree, these programs were to be more selective in the lines of play that they would search deeper. Indeed, most early Chess playing programs were of Type B. However, as the speed of computing hardware increased, Type A programs such as CHESS 4.5 [8] and TECH [48] came to dominate the field. The Type B programs were handicapped by the fact that the problem of heuristically determining the utility of exploring a certain line deeper was unreliable and slow [1, 17, 64, 108].

However, there are a few forward pruning techniques that are employed by many modern Chess engines. *Null move pruning* [12, 34] is applied to nodes one ply up from the leaves of the tree (known as *frontier nodes*). It determines whether the opponent's position is strengthened by allowing the current player

to "pass" his turn; if not, the current node is not expanded further. This approach assumes that doing nothing on one's turn *always* hurts the player's position, although this is known to generally not be true. Many games, including Chess, have so-called *zugzwang* positions where the best move *is* to pass one's turn, if the rules of the game allow it. Futility pruning [88] is also applied to frontier nodes: it works by bounding the expected change in the evaluation of the current position, and determines if this is likely to fall outside the current α and β bounds. If not, then searching further from this position is deemed futile and not pursued. Finally, Michael Buro's ProbCut technique uses offline statistical models built from a large database of positions to guide in-game pruning decisions. While it has produced mixed results in Chess, it has been quite effective in LOGISTELLO, one of the world's strongest Othello programs [22].

Quiescence Search and Selective Extensions

Game playing programs, particularly for Chess, are susceptible to the *Horizon Effect* [14]. This occurs when a program mistakenly believes that an inevitable catastrophe can be avoided by making a series of ineffective (and potentially, more damaging) delaying moves. For example, consider a position where a player will lose his queen in eight plys, or if he chooses an immediate rook sacrifice, will lose the queen in twelve plys. A program that only looks ahead eight plys in this situation will decide to sacrifice the rook, thinking that the queen has been saved; however, the loss of the queen has simply been postponed to a point beyond the search "horizon", and the program is now in an even weaker position having unnecessarily lost a rook as well. This problem was identified very early on by Turing [109] and Shannon [94] who recommended waiting for

positions to become "quiet" before applying static positional evaluation methods. Every good Chess engine today employs a *quiescence search* routine, that ensures that the current position is stable before evaluating it; this entails checking for potential captures, promotions and kings in check.

Another approach that is used to search unstable and interesting lines of play deeper is the use of *selective extensions*. One common approach is to keep count of the number of "interesting" moves that were made as we descend a line of play. If this number exceeds a pre-determined threshold at the leaf node, then this line is searched for another ply or more. This technique is referred to as *fractional ply extension* [58]. Another approach, that was used in DEEP BLUE, is termed the *singular extension* [6]. This technique extends the search depth of moves that are deemed *singular*, i.e. moves that produce positions whose evaluations are better (or worse) than that of all their siblings by a significant margin. Such moves are usually indicative of interesting positions worthy of deeper analysis.

2.3.4 Data Structures

We conclude this section with a brief discussion of some important data structures that are used in Chess engines. Arthur Samuel pioneered the use of *bitboards* in his Checkers program [87], and these are now commonly used in many games. With a small set of 64-bit integers, one can capture the entire state of a Chess board. Each bit is associated with a single cell of the board, and a 64-bit integer is used for each class of pieces. For example, the position of all the white pawns on the board can be captured by simply setting the bits that correspond

to the cells occupied by the the pawns to 1, leaving the remainder unset. This representation is highly space efficient. Moreover, many common tasks such as move generation and game board hashing can be accomplished through the clever use of logical operations on 64-bit integers, that most modern processors handle very efficiently.

A second important data structure that almost every Chess program uses is a *transposition table* [64]. Despite the misleading name, the Chess game tree is really a graph; often, there are multiple paths from a given starting configuration to a goal position, due to move transpositions. To avoid wasting time repeatedly searching the same positions, a hash table is used to record previously encountered states. Zobrist hashing [116], a simple, fast and incremental hashing scheme with a low rate of collisions, is commonly used to index into the table. The table stores several pieces of information about positions, such as the depth to which the position has been searched, its backed up score, whether this is an exact value or a bound, and so on. Checking for a transposition table entry for any newly encountered state, before exploring it further, results in significant time savings for search procedures.

2.4 The Anatomy of Go Playing Programs

In this section, we provide an overview of the techniques that underpin the success of today's best Go playing computer programs.

2.4.1 Monte Carlo Simulation

The *Monte Carlo method* was invented by Stanislaw Ulam and John von Neumann [35], as part of their work on the Manhattan Project, just as the first digital computers were emerging. The approach works on the principle that repeated *random simulations* can often provide good approximations for solutions to complex problems. This approach has been effective in computing difficult integrals, numerical optimization, and modeling applications, among many others. Bruce Abramson was one of the earliest proponents of the idea of using Monte Carlo methods in game-playing [2]. Given a specific board position, his scheme involved playing out the game to completion many times using random moves for both players, and using the average outcome of these *playouts* (or *rollouts*) as an estimator of the utility of the state.

Over the years, sampling techniques have proven quite successful, particularly in games with an element of chance or imperfect information. However, they are usually not used in the way Abramson envisioned. For example, sampling is used in the card games Bridge [49] and Skat [23] to draw perfect information scenarios from the current belief state. This "determinized" game is then solved using conventional game tree search techniques to determine the best action for the particular scenario that was considered. The process is repeated by drawing many samples, and the action that consistently leads to the best outcomes is then played. In Backgammon [105] and Scrabble [95], strong evaluation functions are already available. In these games, simulations are used at the leaf nodes of the search tree, but are stopped after just a few plys. The outcome of the simulation is then estimated using the heuristic function. This augmentation of static positional analysis with playouts yields more stable eval-

uations in these dynamic domains.

2.4.2 Monte-Carlo Tree Search (MCTS)

Consider the following algorithm for decision making in stochastic domains with large state spaces (we assume that a *generative model*, i.e., a simulator for the environment is provided, together with a bounded reward function that supplies the immediate utility of each move): in the initial state s, perform each action a, C times. Use the provided simulator to generate the next state in each case. Repeat this process at these new states, to grow the tree to some depth h. Use *expectimax* backups [85] (with discounting of future rewards) to estimate the value of each action at state s. Can this approach yield the optimal decision? In 2002, Kearns et al. proved that this *Sparse Sampling* (SS) approach can be used to produce *probably-approximately correct* (PAC) decisions at the root [53]. Moreover, they proved that the runtime of their algorithm was *independent* of the size of the state space, though it scales exponentially with the desired accuracy of the approximation and desired confidence. While the algorithm was never practicable, it was nevertheless an important proof of concept that highlighted the potential of sampling-based planners.

Upper Confidence bounds applied to Trees (UCT) [56] belongs to the family of so-called *Monte-Carlo Tree Search (MCTS)* methods, and is inspired by this approach of building a sparse tree using sampling. However, there are a number of key differences. Firstly, in contrast to SS which expands a search tree in a depth-first fashion, UCT is *sequentially best-first* [44]: it proceeds in discrete iterations or episodes, with the outcome of each iteration guiding the growth of

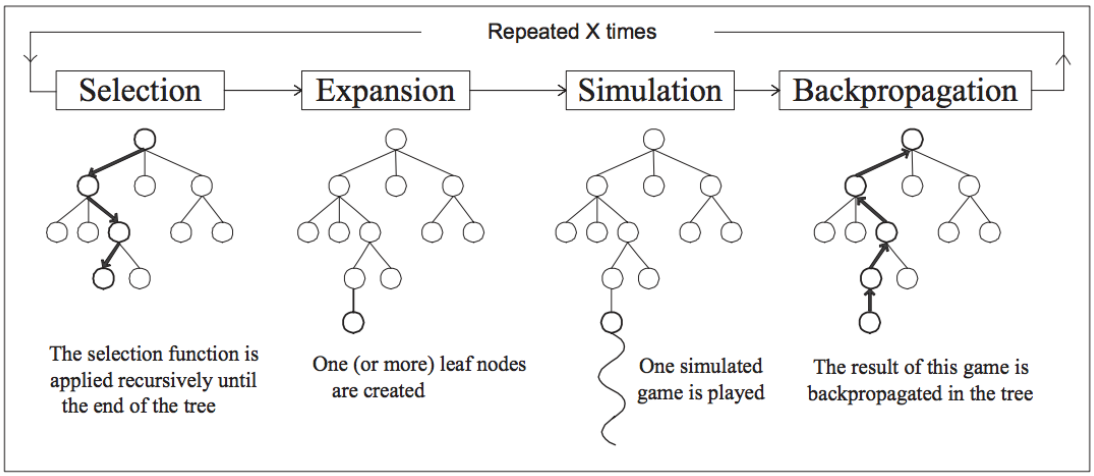

Figure 2.3: The stages comprising a single iteration of Monte Carlo Tree Search (adapted from [26])

the tree on subsequent iterations. Thus, UCT does not uniformly sample all actions like SS. Moreover, it also modifies how leaf node utilities are estimated and propagated up the tree, to account for the fact that our domain of interest is now a two-player game with delayed rewards, rather than a single-agent planning problem. MCTS ideas (though not specifically UCT) were first successfully demonstrated in Rémi Coulom's CRAZY STONE [31] program. Shortly after, the UCT-based MOGO [42] appeared on the scene to kickstart the MCTS revolution in earnest. In Section 2.4.3, we provide a more detailed description of the UCT algorithm, followed by a survey of some of the common enhancements used in MOGO and other top Go playing programs.

2.4.3 Upper Confidence bounds applied to Trees (UCT)

The UCT algorithm [56] builds a search in an iterative, best-first manner. Figure 2.3 offers a useful visualization of the steps comprising a single UCT itera-

tion. We describe each of those component steps in greater detail below.

Selection

On each iteration, UCT starts at the root node of the search tree and recursively descends the tree by applying a *selection operator* (alternately referred to as the *tree policy*). During this process, care is taken to carefully balance the exploitation of known good moves against the exploration of under-sampled moves. This trade-off is done in a principled fashion through the use of the *Upper Confidence Bounds 1 (UCB1)* bandit algorithm [9].

At a state s, UCT selects an action a that maximizes an upper confidence bound on the utility of the action value according to:

$$\pi(s) = \underset{a}{\operatorname{argmax}} \left(Q(s,a) + c \cdot \sqrt{\frac{\log n(s)}{n(s,a)}} \right)$$

The algorithm maintains three pieces of information at each node s in the search tree: $Q(s,a)$ is the current estimate of the utility of taking action a in state s, $n(s)$ is the number of previous visits to state s, and $n(s,a)$ is the number of times action a was selected on previous visits to state s. If $n(s,a) = 0$ for an action a, then we stop the recursion and proceed to the second step of the process, the *expansion* step described below. The constant c is tuned empirically to balance the exploration-exploitation tension that was alluded to earlier. Note that at states where the opposing player is on move, the action that *minimizes* a symmetric lower confidence bound is picked, i.e., the selection operator is:

$$\pi'(s) = \underset{a}{\operatorname{argmin}} \left(Q(s,a) - c \cdot \sqrt{\frac{\log n(s)}{n(s,a)}} \right)$$

Note that are several other candidates for this selection operator; for example, the original implementation of CRAZY STONE picked actions greedily with a probability that was proportional to their estimated utility. Other proposals have called for the use of alternative bandit strategies such as EXP3 or UCB1-Tuned [9], or for Bayesian approaches [106].

Expansion

The recursive descent down the tree stops when a leaf node in the tree is reached; in the case of UCT, a leaf node is any node in the search tree that has not been *completely expanded* (or a terminal node that cannot be expanded any further). If the current leaf node is non-terminal, then one of its unexplored children is added to the search tree. Under this scheme, the size of the search tree grows by one node at the end of every iteration[7].

Simulation

After the expansion step has been completed, one or more random playouts are performed starting at the newly added node, to obtain an estimate R of its utility. In the event that the tree policy terminates in terminal node, R simply corresponds to the outcome of the game — a value drawn from the set $\{-1, 0, +1\}$, to represent a loss, draw or win for Max, respectively. The moves in the random playout are chosen according to a pre-defined *default policy* or *playout policy*. In the simplest case, this policy chooses uniformly at random among all possible legal moves. However, other sophisticated selection schemes are possible as well: an idea we will pursue further in Section 2.4.5. Note that this state utility estimation process does not *have* to employ a playout. A heuristic function, if available, would work just as well. Indeed, this is an approach we will explore later in this

[7]The tree size grows assuming that the tree policy ended at a non-terminal leaf node.

thesis. However, this is rarely encountered in practice since in the majority of UCT's application domains, good heuristics are either unavailable or difficult to encode.

Backpropagation

The final step in a UCT iteration uses the simulation outcome R to update the utility estimates and visit counts of all the nodes t on the path from the leaf to the root node. The node visit and action count updates are straightforward:

$$n(s) \leftarrow n(s) + 1$$

$$n(s, a) \leftarrow n(s, a) + 1$$

The utility estimate is updated as follows:

$$Q(s, a) \leftarrow Q(s, a) + \frac{(R - Q(s, a))}{n(s, a)}$$

Over time, this update has the cumulative effect of associating with each state-action pair the average reward accrued from every UCT episode that passed through it.

The above steps are repeated until the budgeted search time expires, at which point the action leading to the state with the highest mean utility is executed. Kocsis and Szepesvári showed that in the limit, the estimated utilities that UCT computes approach the true Minimax utilities [56]; in other words, just like in the case of SS, UCT approaches optimal decision-making when given enough time. We make a few concluding remarks about the nature of UCT:

1. The algorithm grows *asymmetric* trees. Thanks to the use of the bandit-based selectivity operator, lines of play that appear to consistently yield

high rewards are visited far more frequently, than other variations. The resulting search tree is thus much deeper in some parts than others.

2. In its vanilla form, the algorithm is *aheuristic*. Purely random playouts can be employed to estimate leaf utilities, and UCT's convergence guarantee still holds — though this may severely limit performance in practice.
3. Finally, the algorithm is *anytime* by design, since it implements a form of iterative deepening.

Algorithm 2.3 provides detailed pseudocode for the procedure described.

2.4.4 Rapid Action Value Estimation (RAVE)

One drawback with the UCT approach is that in state spaces with large action sets such as Go, a lot of time is spent sampling from sub-optimal moves; this is necessary to ensure that UCT's explore-exploit decisions are made on the basis of statistically meaningful utility estimates. The *Rapid Action Value Estimation (RAVE)* heuristic attempts to speed up the convergence of move utility estimates by generalizing simulation outcomes across subtrees [44]. This idea was first employed in the Go program GOBBLE [20], where it was called the *All Moves As First (AMAF)* heuristic, and is closely related to the history heuristic that was introduced in Section 2.3.2. Most moves in Go tend to have localized effects, at least when they are first played — as a result, it is possible to obtain a rough valuation of a move by examining it without any consideration of the *context* in which it is played. The RAVE heuristic operates on this principle. It assumes that the utility of playing a move in the current state will be similar to the utility of playing the same move in any descendant of the current state.

Algorithm 2.3: The UCT Algorithm

procedure UCTSEARCH($state, c$)
  while time remains **do**
    UCTRECURSE($state, c$)
  end while
  return SELECTMOVE($state, 0$)
end procedure

procedure UCTRECURSE($state, c$)
  if *state* is terminal **then**
    $val \leftarrow$ EVALUATE($state$)
  else if *state* is not in search tree **then**
    $val \leftarrow$ EVALUATE($state$)
    Add *state* to search tree
  else
    $m \leftarrow$ SELECTMOVE($state, c$)
    $val \leftarrow$ UCTRECURSE(MAKEMOVE($state, m$), c)
    $n(state, m) \leftarrow n(state, m) + 1$
  end if
  $n(state) \leftarrow n(state) + 1$
  $Q(state, m) \leftarrow Q(state, m) + \frac{val - Q(state,m)}{n(state,m)}$
  return *val*
end procedure

procedure SELECTMOVE($state, c$)
  $A \leftarrow$ set of legal moves in *state*
  if *Max* is on move in *state* **then**
    $m \leftarrow \underset{a \in A}{\operatorname{argmax}} \left(Q(state, a) + c \cdot \sqrt{\frac{\log n(state)}{n(state,a)}} \right)$
  else
    $m \leftarrow \underset{a \in A}{\operatorname{argmin}} \left(Q(state, a) - c \cdot \sqrt{\frac{\log n(state)}{n(state,a)}} \right)$
  end if
  return m
end procedure

By sharing statistics on the utility of moves played within each subtree, RAVE is able to quickly hone in on a plausible list of the best candidate moves in a given position. This is then used to help UCT focus its search more quickly than would otherwise be possible.

In MoGo, RAVE is implemented as follows: two new pieces of information — a RAVE estimate $Q_r(s,a)$ and RAVE visit count $n_r(s,a)$ — are maintained in each search tree node, in addition to the original statistics [44]. The basic UCB1 selection operator is then modified to instead select moves according to:

$$\pi(s) = \underset{a}{\operatorname{argmax}} \left((1 - \beta(s,a))Q(s,a) + \beta(s,a)Q_r(s,a) + c \cdot \sqrt{\frac{\log n(s)}{n(s,a)}} \right)$$

When propagating the outcome of a simulation up the search tree, in addition to the standard updates to the utility function estimates and visit counts, the following RAVE update is applied as well:

$$n_r(s,a) \leftarrow n_r(s,a) + 1$$

$$Q_r(s,a) \leftarrow Q_r(s,a) + \frac{(R - Q_r(s,a))}{n_r(s,a)}$$

Some careful bookkeeping ensures that the updates are performed in a manner that is also consistent with which side made the move. There a number of viable candidates for the weighting function $\beta(s,a)$, though they all have the property that $\beta(s,a) \approx 1$ when $n(s)$ is small, and $\beta(s,a) \to 0$ as $n(s)$ grows larger. Thus, UCT's move selection process when constructing the search tree is guided initially by context-free value estimates of moves; as the search progresses, the bias is shifted towards the evaluation of the move in the specific context. Gelly and Silver note that when combined with RAVE, there is no need for an exploration bonus term in UCT (i.e. the constant c is set to 0) — apparently, the implicit exploration of the search space provided by RAVE is sufficient.

While, RAVE has made a huge impact on the overall performance of UCT in Go [42, 43, 44], its wider applicability is somewhat unclear. Notably, Sturtevant reported that RAVE performs poorly in Chinese Checkers and Hearts [101] — these are games where pieces tend to move around a lot and the utility of moves is highly context-sensitive.

2.4.5 Incorporating Domain Knowledge

Modern MCTS based Go programs also exploit the extensive amount of domain knowledge that has been compiled by human experts with decades of effort. We present the two prominent knowledge-based enhancements that most programs utilize.

Search Seeding

One method of incorporating existing Go heuristics into UCT-style approaches has been in a manner similar to that of RAVE — the initial estimated utility of a state-action pair $Q(s,a)$ is initialized to a value determined by applying the heuristic at hand, rather than the default value of 0.5. The initial visit count $n(s,a)$ is given by a *heuristic confidence function* that reflects our confidence in the quality of the evaluation — in most programs, this is just a constant function that is tuned empirically [44]. In this case, the contribution of the heuristic to the utility estimate automatically gets weighted less as more simulations are performed. An alternative approach is to discard the confidence function and to instead use a global weighting function that naturally tapers off with added simulations, in a similar vein to the RAVE biasing schedule β. This approach is dubbed

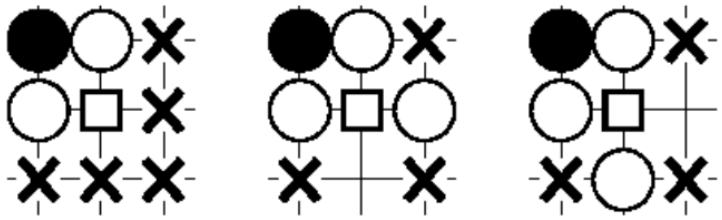

Figure 2.4: An example of a feature used in MoGo [46]. This feature evaluates to `true` if the left-most pattern is matched, and the latter two are not.

progressive biasing [27].

Playout Enhancement

From the very beginning, computer Go researchers have understood the importance of moving beyond using purely random playouts in UCT [46, 45] to estimate utilities. Using slightly more "intelligence" in the move selection process in playouts has produced significant gains in the strength of Go programs [44]. Typically, extensive pattern-based feature databases are used to produce the playout policy. Figure 2.4 shows a sample 3×3 pattern feature. Thousands of such features are used to tile the board, centered around empty intersections, to evaluate the strength of particular moves. To keep runtime costs reasonable, the feature matching begins close to the location of the last move, and progressively widens the search neighborhood [46]. There has also been success in automatically extracting such features from databases of high quality games between humans [27, 32]. Automated approaches that tune policies so that they combine well with UCT (known as *simulation balancing*) are also an active line of research [96]. The problem is made tricky by the fact that simply find-

ing *stronger* simulation policies — ones that win more games against other simulation policies, without search — is insufficient, since strong policies do not correlate with strong play when integrated with UCT [42, 44].

2.4.6 Parallelization

Finally, much like iterative deepening alpha-beta search, the quality of play produced by UCT increases with increased thinking time. The ubiquity of multicore architectures in modern computing hardware, and the natural ease with which UCT lends itself to parallelization, has encouraged work on multi-threaded implementations of UCT. Three main types of parallelization have been studied in the literature — *leaf*, *root* and *tree* parallelization.

Leaf parallelization tackles the component of the search procedure that is most amenable to parallelization — the random playout phase for utility estimation. In this approach, multiple random playouts from a single starting position are performed in parallel across different threads. The outcome of all these simulations is then averaged and used for backpropagation purposes [25]. In *root parallelization*, entirely parallel UCT searches are carried out across different threads, with no communication between the processes. Essentially, an ensemble of independent UCT search trees is constructed. When all the searches complete, the statistics in all the trees are combined together to pick a move at the root node [25]. The final technique is *tree parallelization* [26], which is the trickiest to implement of the three. In this approach, multiple threads collaborate on building a single search tree, with each thread running a separate iteration of UCT. Data consistency is ensured through the careful use of mutexes.

2.5 Discussion

Since the time MCTS techniques made their first appearance in the programs CRAZY STONE and MOGO, there has been a glut of new work in capplying and adapting these algorithms to other challenging problems. Ciancarini and Favini demonstrated the success of UCT in Kriegspiel, an imperfect information variant of Chess [29]. Finsson and Björnsson's CADIAPLAYER used UCT based planning techniques to win the AAAI General Game Playing contest in 2007 and 2008 [36]. Balla and Fern described how UCT could be used for planning in the real-time strategy game Wargus, a domain with an extremely large action space [10]. They reported that their system outperformed several baseline planners, as well as experienced human players, in a number of scenarios. Sturtevant extended UCT to the multi-agent setting, in the process producing an expert level Hearts player [102]. And the success stories extend beyond games: UCT has been adapted for a number of other interesting problems such as planning in large, Partially-Observable Markov Decision Processes (POMDPs) [97], domain independent probabilistic planning [54], feature selection for machine learning [41], and natural language phrase generation [28]. Interestingly, there appears to be a clear dichotomy in the domains where UCT-style and Minimax-style approaches succeed. Minimax approaches haven't scaled well in domains like Go, whereas UCT has not translated its spectacular success in Go into domains like Chess and Checkers, where Minimax has traditionally dominated. The quest to understand this rift is the primary motivation for the work that is presented in the following chapters.

CHAPTER 3

TRADE-OFFS IN SAMPLING-BASED PLANNING

Much of the research that is carried out in the area of adversarial search is done in competitive settings, where the overall goal is to build a winning player for games such as Chess, Go, or Poker. Such competition has clearly pushed the development of search techniques but it has come at a certain price. Our understanding of why one method outperforms another, and in what domain, is somewhat limited and anecdotal. This is in part because a winning player has to incorporate a whole range of techniques, including clever domain specific heuristics, to be competitive, which makes it difficult to study techniques in a more pure form.

In this chapter, we establish the veracity of some of the speculative and informal observations that have been made by researchers about the behavior and properties of UCT, using carefully designed experiments. In Section 3.1, we address the commonly-held view that "UCT does not perform well in Chess". We show that the reality is a little more nuanced; once we eliminate all the carefully engineered bells and whistles found in modern Chess engines (in this case, GNU CHESS[1]) and equalize the playing field, UCT actually *outperforms* Minimax search. However, once we introduce some domain-specific knowledge, Minimax is able to exploit this much more efficiently. We also show that random playouts, rather surprisingly, can provide useful information even in a domain as dynamic as Chess.

In Section 3.2, we carry out of a dissection of the UCT algorithm in the domain of Mancala. The choice of Mancala as a testbed is motivated by the fact that

[1] `http://www.gnu.org/software/chess`

both UCT and Minimax perform reasonably well in this game with a minimal amount of knowledge engineering. This is important when we are benchmarking the performance of these algorithms against each other, since we know that both procedures produce players of reasonable strength; it is trickier to carry out such studies in domains like Chess and Go, where one algorithm is head and shoulders above the other. We examine some of the trade-offs that one encounters in practice when using UCT and demonstrate that:

1. the exploration-exploitation balancing performed by UCT does indeed make a noticeable impact on game-playing performance,
2. that given a fixed computational budget, one is better off spending time building larger trees, than on accurate evaluation, and
3. that a Minimaxing backup operator is more robust when good heuristics are available, in contrast to the traditional averaging approach.

3.1 Experiments in Chess

The success of UCT in Go raises the natural question of whether UCT is also effective in other adversarial reasoning domains. In this section, we address this question by studying UCT in the context of Chess. We choose Chess as our evaluation domain mainly because standard Minimax search works so well for it. We can therefore study the behavior of UCT by comparing its performance with traditional Minimax results as the "gold standard". As we will see, UCT *per se* is not competitive in Chess. However, there are promising aspects of UCT that may be used to complement more traditional search.

3.1.1 Knowledge-free Setting

We begin by exploring the extent to which UCT-style search methods can compete with Minimax search in a completely knowledge-free setting. This situation arises, for instance, in reasoning about Quantified Boolean Formulas (QBF) where all we have as input is a formula, without any information about the semantics of the variables or the specifics of the problem domain that the formula is encapsulating. This also occurs in the problem of general game playing [36].

For our empirical exploration of the behavior of UCT and Minimax, we begin with the setting of Chess, but modify Minimax to avoid using any Chess-specific heuristic information, pretending that the domain is unknown. In addition, we do not use selective extensions, quiescence searching or move ordering heuristics (these are disabled in our adaptation of GNU CHESS). The Minimax agent does, however, retain alpha-beta pruning. We use MM-k-nR ($k, n \geq 1$) to denote the Minimax player that performs a Minimax search of depth k, uses values from the set $\{-1, 0, +1\}$ at a leaf if it corresponds to a terminal state, and uses the average outcome of n random playouts if the leaf corresponds to a non-terminal state. This produces a player that is aware of winning (losing) positions within its search horizon, but otherwise has the same playout style information as is available to UCT. To keep runtimes reasonable, we terminate any playout trajectories that exceed 300 plys in length, and treat them as draws.

The results are reported in Table 3.1, which gives the *success rate* of the column player against the row player. The success rate, throughout this section, is computed by assigning a score of 0 to each game lost, 1 to each game won, and 0.5 to each game that resulted in a draw. Note that if m games are played between two players, the sum of the success rates of the two players will be pre-

Table 3.1: UCT and a purely Random player compared against Minimax without domain knowledge. Table reports the success rate of the column player against the row (Minimax) player.

Minimax		**UCT**	**Random**
depth	#nodes		
MM-2-1R	1,000	74%	6%
MM-4-1R	10,000	94%	0%
MM-6-1R	200,000	96%	0%

cisely m. Further, if each of players A and A' wins 3/4 of the non-drawn games against B but A draws fewer games, then the success rate of A will be higher than that of A' — a desirable property. In this and all experiments, unless otherwise stated, we report the average success rate over a total of 100 games played from the default starting position of Chess, with 50 played as White and 50 as Black. The variation amongst games is induced by the stochastic nature of (at least one of) the players.

The players used for comparison are UCT with random playouts (UCT) and the "random" player that simply selects a legal move uniformly at random. The UCT player is given roughly the same amount of computation power, measured using the number of nodes explored (rather than runtime, in order to discount any implementation differences), as the Minimax player it is competing against. We observe that even though MM-k-1R acts without much information in many situations, it is far from a trivial player as evidenced by its clear success against the random player. Also, searching deeper improves the performance of MM-

k-1R; not only is the success rate of MM-6-1R against the random player higher than that of MM-2-1R, in a direct playoff (not shown in the table), MM-6-1R has a success rate of 66% against MM-2-1R. Finally, UCT significantly outperforms MM-k-1R, demonstrating the potential of UCT in completely knowledge-free settings.

3.1.2 Boosting UCT with Heuristic Information

We now consider the setting where we do have prior domain knowledge. We are interested in the extent to which this can be exploited to enhance UCT. Heuristics have already provided promising results when combined with UCT in Go. Typically the heuristic value is used to initialize the value of leaf nodes to bias the selection process on early visits to the node. However, since current heuristics in Go are not very strong, UCT is set up to fairly quickly override the heuristic value with playout estimates once the node has been visited sufficiently many (typically a few dozen) times. In contrast, for Chess, we have heuristics that are much more powerful, and we explore how much they can boost the performance of UCT. We also compare this against the gains that Minimax achieves, when it is given access to a heuristic.

To evaluate this, we consider the player UCT_H that uses the board evaluation heuristic of GNU CHESS at the leaves visited by UCT, rather than the $\{-1, 0, +1\}$ values obtained from random playouts; in other words, we fully replace playouts with heuristic evaluations. This still preserves the convergence properties of UCT, i.e., with sufficiently many iterations, UCT_H will converge to the true Minimax value of each node. There is however one subtle issue: *how should*

we normalize heuristic values to make them comparable in effect to the $\{-1, 0, +1\}$ *values returned by playouts in standard UCT?* We found that on non-terminal nodes, GNU CHESS typically assigns heuristic values in the order of 100's to low 1000's, while for terminal boards, it assigns $\pm 32,767$ – a very large value in comparison to the other heuristic values. This causes the value backup/update operation of UCT to become artificially drastic whenever even a single terminal node is encountered. To mitigate this effect, we use "squashed" heuristic values: terminal positions are assigned values $\pm 6,500$ (the highest empirically observed heuristic valuation of any position) and the heuristic value of non-terminal boards is kept unchanged.[2]

Table 3.2: Success rate of UCT_H against UCT.

	UCT_H		
	50 iter	100 iter	1,000 iter
UCT (10,000 iter)	55.0%	85.5%	96.5%

Table 3.2 summarizes the results. We report the success rate of UCT_H when compared against UCT with random playouts. Not only is UCT_H significantly faster than UCT (because it doesn't do playouts and thus avoids relatively expensive repeated move generation), it needs drastically fewer iterations to be competitive with UCT. For example, with only 50 iterations, UCT_H is already competitive with UCT with 10,000 iterations; with 1,000 iterations, UCT_H almost always defeats UCT with 10,000 iterations. These results demonstrate that

[2]We also tried other scaling options, such as the sigmoid function and other fixed values for terminal boards, but this simple scheme seemed to generally work fairly well in our experiments.

a well-designed heuristic can provide a significant boost to UCT.

A natural question to ask, then, is how well does UCT_H actually compete as a player against Minimax? Unfortunately, for games such as Chess where Minimax is *the* successful strategy, even UCT_H doesn't fare too well. We found that with 5,000 iterations, UCT_H is only about as powerful as MM-2, a 2-ply Minimax search employing the GNU CHESS heuristic that only examines about 500 nodes. About 100,000 iterations are needed to get UCT_H on-par with MM-4 and almost no reasonable amount of compute time can close the gap to MM-6. This suggests that the difference in the performance of UCT in Go versus Chess is not only due to the quality of the heuristic, but perhaps more importantly, due to the different nature of the two underlying search spaces. We explore this hypothesis in greater detail in Chapter 4.

3.1.3 Enhancing Random Playouts

We now focus our attention on one of the key facets of UCT, the use of *random playouts*, and ask whether such playouts can provide useful information in domains such as Chess where we already have well-designed state evaluation heuristics. An interesting question in the context of playouts is, *is it at all possible to obtain useful information about a strong player by doing several playouts between two weak players?* We find that random playouts tend not to provide any more information than Chess heuristics themselves, but a slightly more powerful playout—namely a playout between two MM-2 players—*can*, surprisingly, reveal information that is often visible only to a significantly deeper Minimax player such as MM-8. We quantify this in terms of a strong correlation between

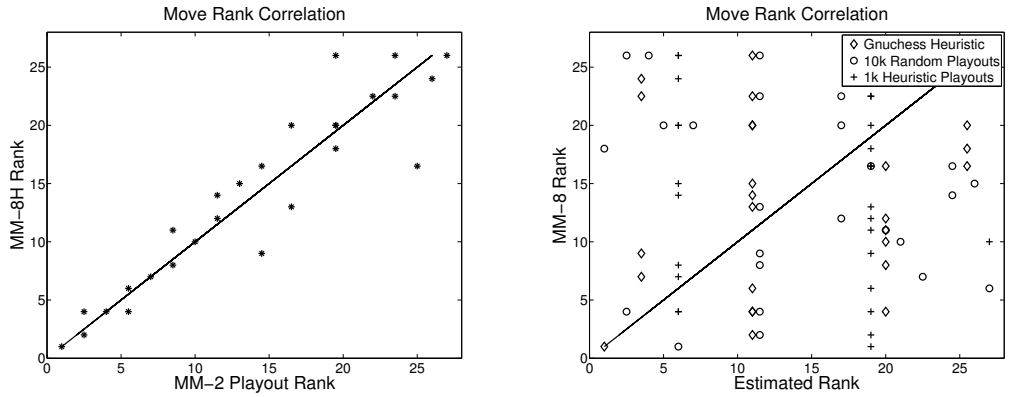

Figure 3.1: Correlation of move rankings of various players (x-axis) against MM-8 rankings (y-axis). Left: playouts using MM-2. Right: GNU CHESS heuristic, random playouts, heuristic playouts.

move rankings obtained by the two players.

In Figure 3.1, we explore how good heuristics and various kinds of playouts are in obtaining information that is visible to a strong player, such as a deep MM-*k* player (for experimental purposes, we use MM-8 as the gold standard). For this evaluation, we consider boards taken from games between Chess Grandmasters and compute the *ranking* from best to worst (1 being the best) of the possible moves as given by an MM-8 evaluation of each resulting state. Note that during actual game-play, only the relative ordering of moves matters; it is for this reason that we choose to study the correlation of the move rankings rather than their raw estimated values. This also helps circumvent the problem of comparing leaf value estimation methods whose outputs do not map to the same range of values. For each kind of estimation method, we apply smoothing by considering estimates within some ϵ of each other as ties and assigning them the same rank.

Figure 3.1 shows the results for a typical board drawn 16 moves (31 plys) into

a game between Chess Grandmasters. In the left panel, we compare for each child, its MM-8 ranking (y-axis) against the ranking obtained based on playouts using two MM-2 players (x-axis). The points being almost on the diagonal shows that the two rankings are very well correlated, especially in the region of most interest—the bottom-left region, representing moves that are considered very good by both players. In contrast, the right pane of the figure shows that the rankings obtained using the GNU CHESS heuristic, random playouts, or playouts between heuristic players (x-axis) are much more loosely correlated with MM-8 rankings (y-axis). For example, points in the top left corner represent moves that MM-8 thinks are very poor but the other player thinks are quite good. Similarly, points in the bottom right corner indicate good moves, as identified by MM-8, that are dismissed as bad moves by the weaker player.

Overall, this demonstrates that playouts between slightly informed players, namely MM-2 players in this case, *can* have a strong correlation with information that is usually visible only to a much stronger player, namely MM-8 in this case. A natural question to ask at this stage, *how does the ranking induced by an MM-2 search itself compare to that induced by a playout between two MM-2 players*? We have discovered that there are in fact situations in which a playout of two MM-2 players uncovers information that an MM-2 search does not. Example 1 describes a concrete occurrence of this phenomenon.

Example 1. *Consider the Chess board shown in Figure 3.2. We will follow the standard algebraic chess notation in our discussion, where rows (ranks) are labeled 1-8 and columns (files) are labeled a-h, with a1 being the bottom left corner. In the given state, the Black king is in check with Black on move and an MM-2 search recommends that the king be moved to h8. However, this allows White a devastating counter-move: moving its pawn on file f to f5 and thereby trapping Black's rook. Black can stall for two moves*

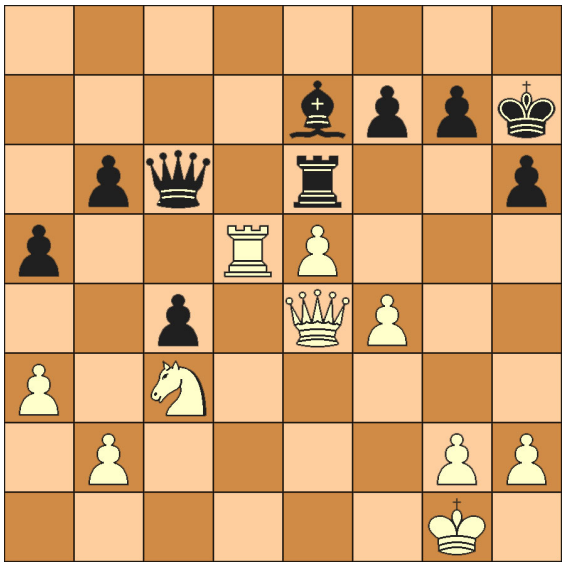

Figure 3.2: A board where playouts with MM-2 players are able to discover a soft trap visible at depth 9 while complete MM-2 search misses it.

by using its bishop to place the White king in check, and subsequently freeing its rook to escape up file e. In this case, White simply follows Black's fleeing rook by moving its own rook to the same rank as the Black rook. This sets up a situation where Black is at minimum forced to trade its queen and rook for the White queen. Sub-optimal sequences of play result in much costlier piece exchanges for Black. The correct move in the original position is for Black to move its pawn on file g to g6, thereby nullifying White's pawn threat—this is the move prescribed by a complete MM-8 search, as well as an MM-2 playout.

3.2 Experiments in Mancala

We are interested in examining to what extent the various facets of the UCT algorithm — exploration-exploitation balancing, random playouts, and the av-

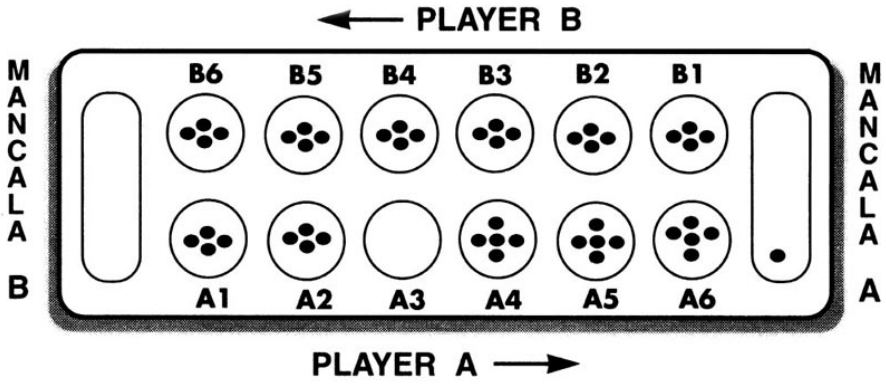

Figure 3.3: A sample Mancala game state

eraging utility propagation mechanism — contribute to the success of the algorithm. In this section, we explore these issues in the setting of Mancala.

3.2.1 The Rules of Mancala

Mancala is a deterministic, two-player game that is thousands of years old and popular in many parts of the world. It is played on a rectangular board like the one depicted in figure 3.3. Initially, the six pits on each side of the board contain 4 stones each (though variants with different numbers of pits and stones exist). The two larger pits at the ends (termed the "stores") hold any stones that the players capture.

A move consists of a player picking up all the stones from a pit on his side and placing them one at a time in each of the following pits, in a counter-clockwise order. The player's own store is included in this "sowing" process, but the opponent's store is skipped. A stone that lands in a store is deemed captured and is permanently removed from circulation. The game is not strictly turn-taking — depending on where the last stone is placed, one of three things

may happen:

1. If the last stone lands in the player's own store, then he goes again.
2. If the last stone lands in an empty pit on the player's own side, then a capture occurs — the single stone is immediately moved to the player's store, as are any stones in the pit directly across from the empty pit (on the opponent's side).
3. Otherwise, the turn ends.

Figure 3.3 shows the state of the game after player A has made the first move of the game, *A*3. After this move, player A would go again, since his last stone landed in his store. The game ends when either side has no legal moves left. At this point, the player with the emptied pits captures any stones on the opponent's side of the board. The winner is the player with the greater number of stones in his store at the game's end. Mancala has previously received some attention from the AI community. Most notably, the game has been weakly solved for a variety of starting configurations [51].

A pertinent question at this point is: *Why Mancala?* Minimax reigns supreme in games with strong tactical components, such as Chess, and attempts to reach similar levels of performance in these domains using UCT have generally been unsuccessful (as we saw in Section 3.1). UCT's notable successes so far have been in domains where Minimax-based approaches have fallen short. These are typically games with large branching factors such as Go or Hex [7] and/or for which good heuristics are unavailable. One exception here is the game Lines of Action, a domain previously dominated by Minimax, where UCT with a significant amount of knowledge engineering is now competitive [115]. But by and

large, the set of domains where Minimax and UCT both produce competent players is disjoint. To our knowledge, Mancala is the first domain where UCT and Minimax *both* produce players that are competitive with nearly *no enhancements* to the basic algorithms. This latter feature is particularly important since it allows us to study these two algorithms side-by-side in their purest form.

In the remainder of this section, we will take a closer look at the three steps that comprise an iteration of UCT — targeted expansion of nodes in the search tree, the use of random playouts to evaluate leaf node positions and the propagation of information from the leaves to the internal nodes through averaging. For each step, we will look at how it contributes to UCT's decision making process and consider the tradeoffs involved in applying alternate approaches.

3.2.2 Experimental Methodology

In the sections that follow, we present data that averages the outcome of thousands of Mancala games between different search algorithms. Here, we outline some of the methodology that was used to run and score those games.

Since Mancala is deterministic, we need a way to introduce variance between individual games. We accomplish this by generating random initial configurations as follows: starting with an empty board, we place a stone uniformly at random in one of the pits (excluding the stores), until there are no stones left. This may however generate positions that are heavily biased in favor of one player. To combat this, we play duplicate games. Given a starting position, two games are played, with the players' sides swapped after the first game. Player A is deemed to have won if he beats player B's result while playing from one

side, while at least matching player B's result when the sides are switched. Otherwise, we disregard the entire board. For example, if player A wins the first game, and draws the second, he is the winner on that board; if player A were to lose the second game, then the board would be deemed too skewed to be useful. We report the win-rates for the different algorithms — this is simply the ratio of the number of duplicate games won to the total number of non-skewed boards that were used.

The other important variable we control for in comparisons between a Minimax agent and a UCT agent is the amount of search effort. We measure this in terms of the number nodes expanded by the algorithm. In all comparisons involving a UCT player and a Minimax player, the former is allowed to expand at most the number of nodes its opponent would expand at *the given position in the game*, unless stated otherwise. This is important given the wide variation in the branching factor of the tree as the game progresses. Also, our Minimax player always uses alpha-beta pruning.

3.2.3 Full-Width Search vs Selective Search

In many applications, it has been observed that the value of the exploration bias parameter c in the selection step of UCT has a large role in the overall performance of the algorithm (for example, see [60]). We concretely illustrate this phenomenon in Mancala. Figure 3.4 depicts the nodes expanded by a an 8-ply lookahead Minimax player (MM-8) on a typical Mancala position. The graph layout, which was generated using GRAPHVIZ [40], places nodes at the same depth from the root (denoted by the blue square) at the same radius. The

board evaluation heuristic applied at the leaf nodes is the difference of the stone counts in the stores of the two players. This simple heuristic is nevertheless quite effective and difficult to improve upon with additional features such as mobility [47].

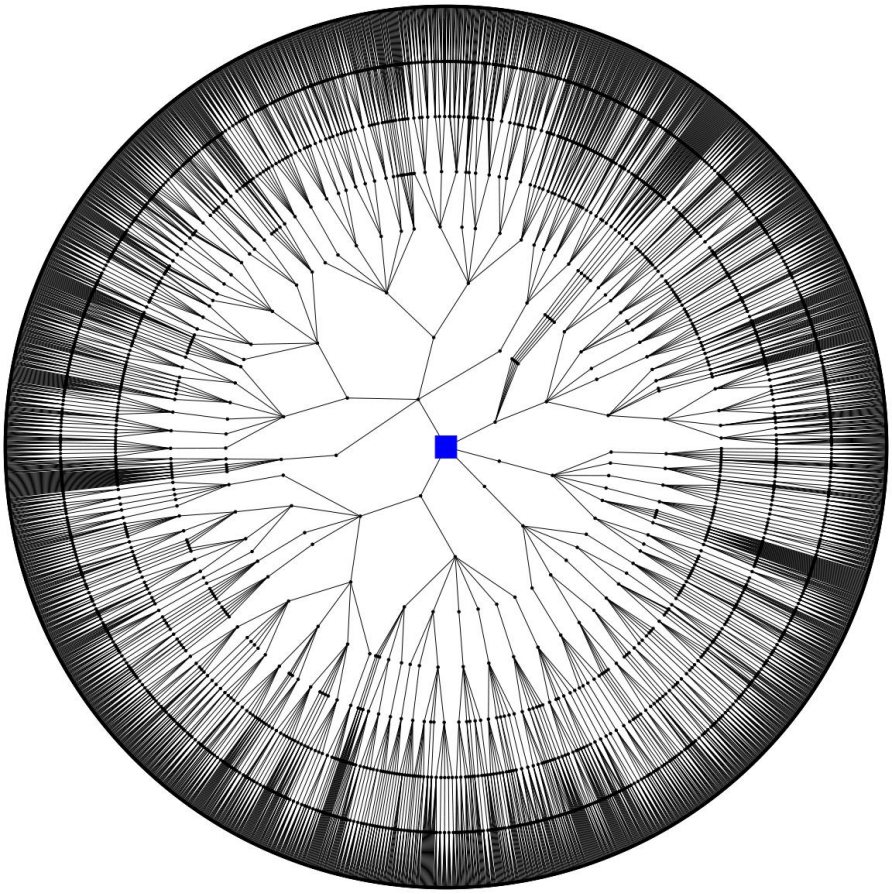

Figure 3.4: Search tree expanded by MM-8 with alpha-beta pruning

We now consider the search trees that are created by a variant of UCT that uses the same leaf evaluation heuristic, rather than random playouts (i.e., UCT$_H$). Figure 3.5 shows the trees that are built by UCT$_H$ for different values of c, for the same board that was used in Figure 3.4. With $c = 0$, the tree reaches some very deep positions, but is sparse at any given level. With too much exploration ($c = 20$), the tree becomes more regular and "Minimax-like". At $c = 2.5$,

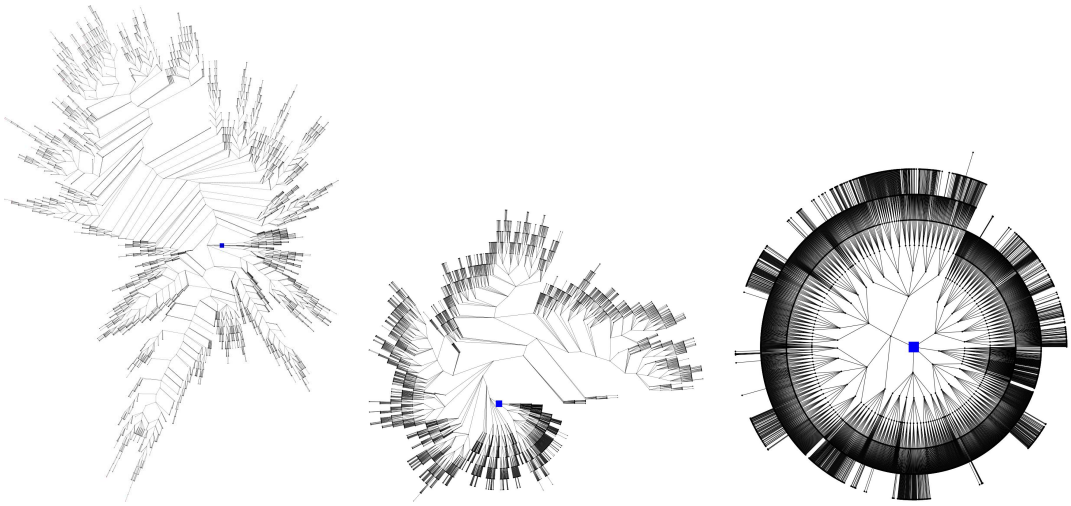

Figure 3.5: From left to right: search trees expanded by UCT_H with $c = 0$, $c = 2.5$ and $c = 20$ respectively

the best-performing value for this domain, we obtain a tree that strikes a balance between the two extremes; it is dense at shallow depths, but more sparse and focused at deeper levels.

This has important implications for the performance of the UCT player. In figure 3.6, we plot the win rate of UCT_H against a fixed Minimax opponent (MM-8), while varying the value of c. Each data-point is the average of several hundred games. The trend is unmistakable — for small values of c, UCT misses too many moves and does poorly. With too large a c, UCT is not sufficiently focused and the action utilities do not converge quickly enough to produce competent play. There is a clear optimal setting for the value of c. This establishes that Mancala is a challenging domain for a planning agent, and any algorithm that tries to balance exploration-exploitation must do so carefully. Furthermore, as we will show in later sections, a variant of UCT that builds trees similar to that shown in the middle panel of figure 3.5 outperforms Minimax search in this domain. This is striking — traditional wisdom in the game-playing com-

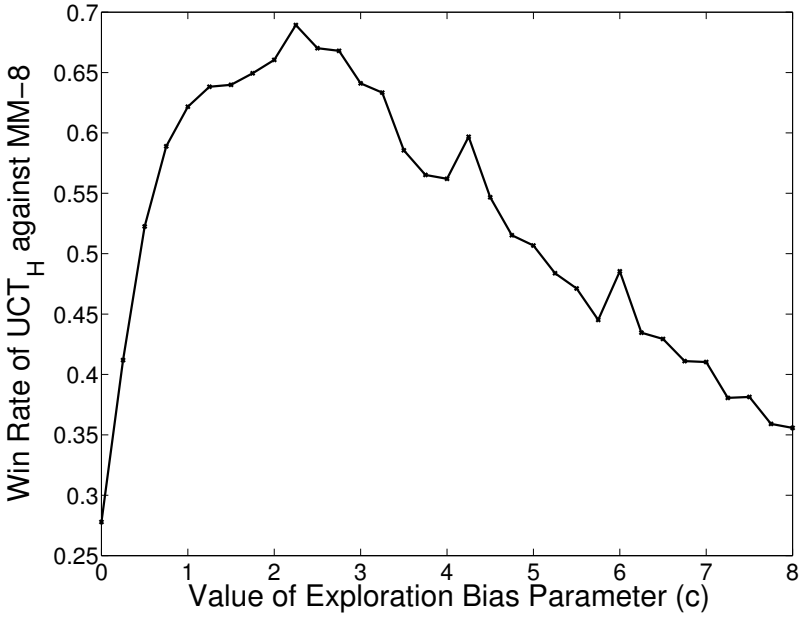

Figure 3.6: Win rate of UCT against an MM-8 opponent, while varying the exploration bias parameter (c)

munity is that search methods that employ such selective search or forward pruning techniques generally perform much worse than their full-width search counterparts (when the latter approach is feasible) [8, 17, 64, 108]. Yet, as these results demonstrate, UCT is capable of performing effective forward pruning in a completely domain-independent manner and still be competitive with complete search methods.

3.2.4 Playouts vs Nodes Expanded

In its original incarnation, the UCT algorithm uses a single random playout to estimate the utility of leaf nodes in the search tree. This is particularly useful in games where good heuristics to statically evaluate positions are not known,

since playouts offer a domain-independent solution to this problem. While any number of playouts may be performed from a node to estimate its utility, typically only a single playout is performed due to run-time considerations. In this section, we address the question: *Given a fixed budget, what is the best way to allocate the playouts?* Should we expand fewer nodes in the search tree while evaluating each one carefully, or should we look at more nodes without being overly worried about the accuracy of their evaluation?

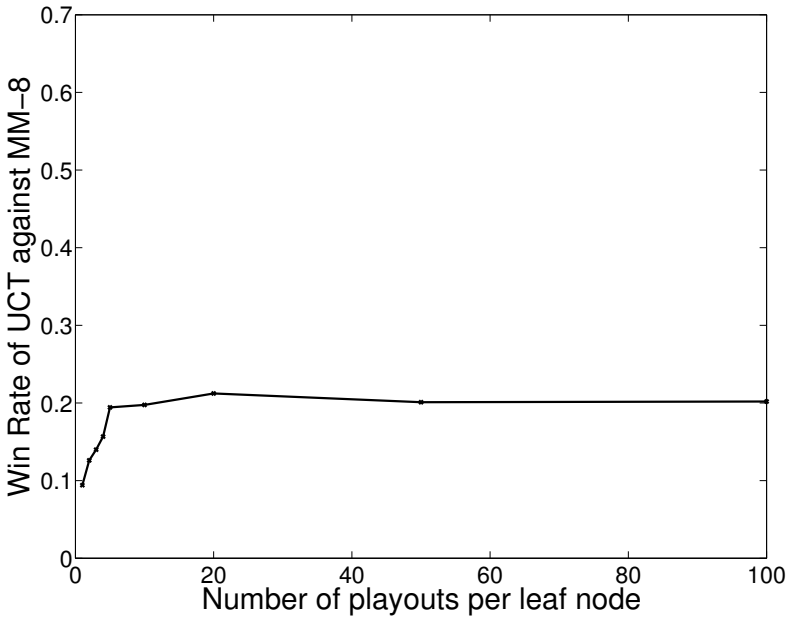

Figure 3.7: Win Rate of UCT against MM-8, while varying the size of the UCT search tree

In our first experiment, we play a UCT agent using random playouts against a fixed opponent (an MM-8 search with alpha-beta pruning using the hand-engineered heuristic) in a series of games. We vary the number of random playouts that are averaged to produce the leaf utility estimates for UCT, holding all other parameters constant. The results are presented in Figure 3.7. In a second experiment, we fix the number of random playouts at 1, while varying the

number of iterations for which we run UCT. The results of this experiment are shown in Figure 3.8.

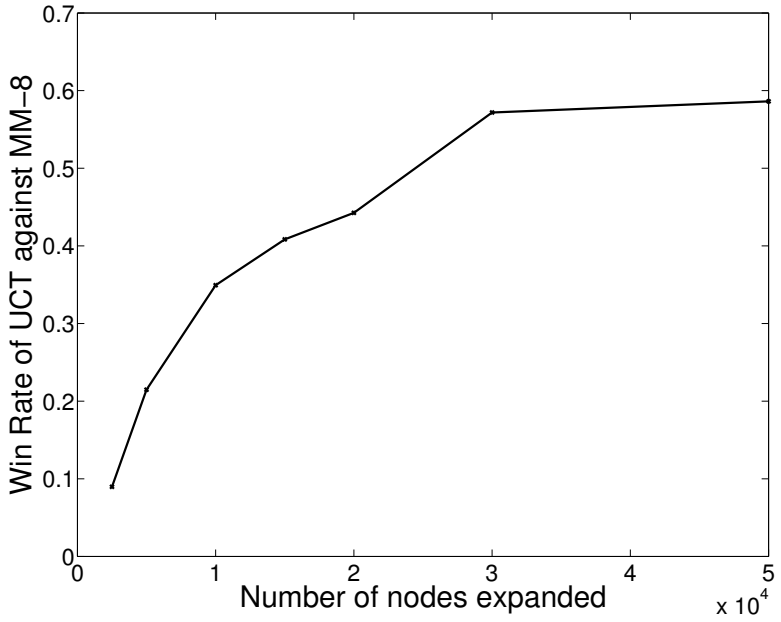

Figure 3.8: Win Rate of UCT against MM-8, while varying the size of the UCT search tree

The two plots taken together highlight an interesting phenomenon. Increasing the number of playouts per leaf only has a modest impact on the playing strength of the agent (quickly leveling off), while increasing the number of UCT iterations (i.e. nodes expanded) yields a much more substantial improvement. This suggests that given a fixed computational budget (in a domain where heuristics are unavailable), it is far better to run more iterations of UCT with fewer playouts per leaf, than to run fewer iterations with more playouts. This is illustrated concretely in figure 3.9: against MM-8, we give UCT a fixed total playout budget and vary the number of random playouts per leaf. For example, with 2 playouts per leaf, we run UCT for $10,000$ iterations, with 4 playouts per leaf, we run UCT for 5000 iterations and so on. Note that in this experiment, we

are not concerned with normalizing the number of nodes expanded by the two algorithms. We are only interested in the impact that different allocations of the playout budget has on the playing strength of UCT, against a fixed opponent.

These findings are consistent with past observations that random playouts do contain some information but the quality of their feedback is limited (see Section 3.1.3). Thus, increasing the number of playouts per leaf only helps to some extent. Why does UCT perform better with fewer playouts, but more iterations? The answer lies in the fact that as UCT builds up its search tree, it begins to "freeze" into a principal variation in the higher reaches of the tree, which introduces a progressive bias in subsequent random playouts. Put another way, as the search advances, random playouts are carried out starting from nodes deeper in the tree, after fixing a few early moves to "good" ones, which increases their evaluative accuracy.

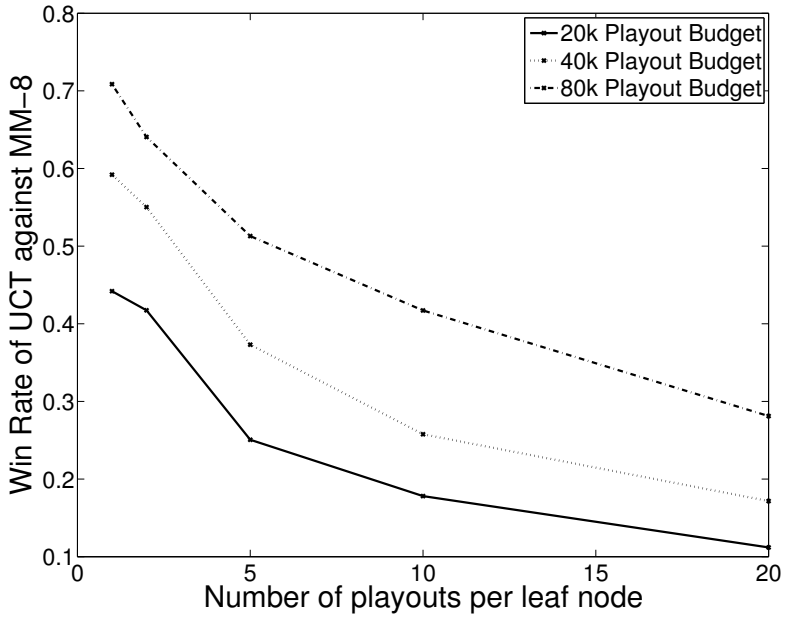

Figure 3.9: Win Rate of UCT against MM-8, with fixed total playout budget, while varying number of playouts per leaf node

3.2.5 Averaging vs Minimaxing

UCT's tree-building and information-propagation steps are interleaved. The building process is guided by a Minimaxing descent through the existing tree, that determines which section of the tree will be sampled next. The new information discovered on the current iteration is propagated up by an averaging operator, which informs the future growth of the tree. While the Minimaxing descent is justified by theoretical results for bandit-algorithms [9], it is not clear that averaging is the best way to propagate information up the tree. In the limit, the action utilities computed by UCT are guaranteed to converge to the true Minimax values [56], but the time to convergence may be super-exponential [30].

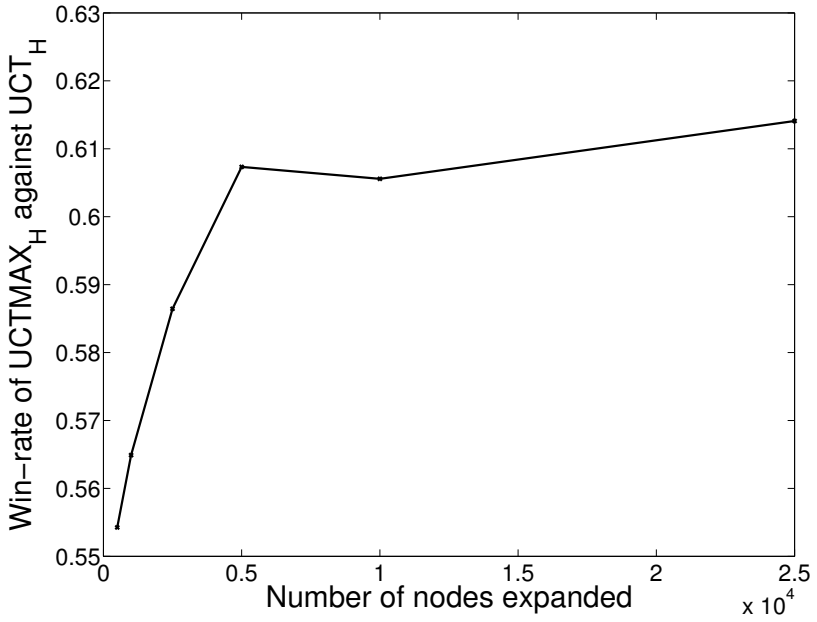

Figure 3.10: Win Rate of UCTMAX$_H$ against UCT$_H$ on complete Mancala games vs the number of iterations

Coulom empirically showed that when using random playouts with UCT,

averaging is the superior back-up operator when a node has been visited few times, whereas Minimaxing is better when node visit counts are higher [31]. Our results confirm that when using playouts, this phenomenon occurs in Mancala as well. However, while good heuristics are not known for Go, we *do* have a heuristic available for our domain. Thus, in Mancala, even a single visit to a node may yield a reasonably good estimate of its utility. Given this, can Minimaxing do a better job as a back-up operator, than the standard averaging approach?

In our first experiment, we play games between a UCT_H agent and a UCT agent that uses Minimax backups as defined below with the same heuristic (henceforth, UCTMAX_H).

$$n(s) \leftarrow n(s) + 1$$

$$Q(s) \leftarrow \begin{cases} n(s) \cdot \max\limits_{s' \in succ(s)} Q(s') & \text{if } s \text{ is a maximizing node} \\ n(s) \cdot \min\limits_{s' \in succ(s)} Q(s') & \text{if } s \text{ is a minimizing node} \end{cases}$$

Figure 3.10 shows the win rate of UCTMAX_H, as we vary the number of iterations that both players are given (all other parameters are fixed to the same values for both players). Note that even with just 500 iterations (nodes expanded), UCTMAX_H is already on-par with UCT_H. This suggests that in domains where good heuristics are available, UCT can be significantly boosted by replacing the averaging back-up operator with Minimaxing *even on small trees*. Increasing the number of iterations increases the performance gap between the two approaches.

One possible cause for why UCTMAX_H outperforms UCT_H could be that the former propagates information from terminal nodes much more efficiently than the latter. To test this hypothesis, we ran "partial"-games between UCTMAX_H

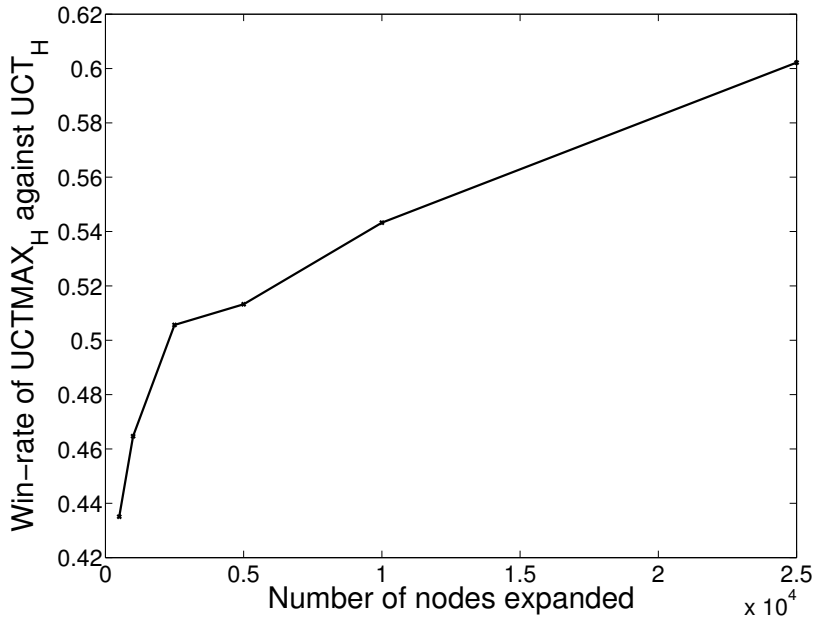

Figure 3.11: Win Rate of UCTMAX$_H$ against UCT$_H$ on partial Mancala games vs the number of iterations

and UCT$_H$ as follows: for the first 14 plys of a game (where terminal nodes are rare), we play UCTMAX$_H$ against UCT$_H$, and play two MM-16 players against each other thereafter. This lets us evaluate the relative strengths of the two algorithms in parts of the search space with few terminal nodes. Figure 3.11 plots the win rate of UCTMAX$_H$ in this set-up, as we vary the number of iterations. Once again, we observe an upward trend as the number of iterations is increased. With limited compute time, UCTMAX$_H$ makes decisions that are about as good (or marginally worse) than UCT$_H$ in regions of the search space where terminal nodes are absent. However, this is compensated by UCTMAX$_H$'s ability to efficiently handle terminal nodes when they do appear (as evidenced by its higher win rate in complete games, with the same parameter settings). With more compute time, UCTMAX$_H$ makes better decisions, even in regions that have few terminal nodes.

3.3 Discussion

The work presented in the first section of this chapter reexamined some of the commonly held beliefs surrounding the (non-)performance of UCT in domains such as Chess, where Minimax search has traditionally been very effective. Our results demonstrate that UCT consistently beats Minimax when used as a domain-independent planner, that it can be significantly boosted by incorporating a state evaluation function (though not to the same extent as Minimax), and that playouts can actually yield useful guidance.

In the second section, we studied UCT in the game of Mancala. To our knowledge, this is the first domain where it has been possible to make UCT competitive with Minimax with minimal amounts of knowledge-engineering. We carried out a dissection of UCT in this domain and showed that:

1. selective search *can* outperform full-width tree searches in some adversarial domains,
2. given a fixed computational budget, it is more prudent to build larger trees with just a few random playouts per leaf node, and
3. when a reasonable heuristic function is known for a domain, using a Minimaxing backup operator can offer a significant performance boost over the traditional averaging operator.

Finally, in a partial game setting, we demonstrated that the quality of decisions made by UCTMAX$_H$ were superior to those of UCT$_H$ in parts of the search space where terminal states proliferate. The effectiveness of the Minimaxing backup in these partial games, together with Minimax's superiority in Chess, suggests

that the density of terminal nodes in the search tree could have a significant impact on UCT's performance. We examine this hypothesis in greater detail in Chapter 4.

CHAPTER 4

ON THE EFFECTS OF TRAP STATES

In Chapter 3, we observed that the presence of early terminal nodes in the search tree appeared to have an impact on the performance of UCT. In this chapter, we examine this hypothesis further. We introduce and study the concept of *states at risk* and *shallow search traps*. Informally, a player is in a state that is at risk of falling into a search trap if there exists an unfortunate action m for him such that after executing m, the opponent has a guaranteed, short winning strategy (with optimal play). We will call the state after executing m a *trap*. We show that such traps, from where the opponent can win within as few as 3 to 7 plys, exist in interesting plays of Chess, at various depths of play.

In contrast, the winning condition in Go is typically only satisfied when the entire board is close to being filled. On average, games between expert Go players last about 200 moves; even the shortest recorded games last over 40 moves [5]. However, even in these cases, the end of the game is brought about by the resignation of one of the players. A computer search must instead proceed until terminal nodes can be used to conclusively establish the outcome of the game (or an endgame database can be consulted). Thus, computer Go players will typically have to play even longer games before they start to encounter terminal positions. We therefore notice a fundamental difference in the structure of the Go search space, in comparison to Chess: *it does not feature shallow search traps until deep into the game*.

When traps occur at all depths, the best way of detecting and avoiding them appears to be an iterative deepening Minimax-style search. Despite UCT's asymptotic convergence guarantee to the Minimax value function, it seems to

be incapable recognizing traps in Chess at depths 5 or greater. Specifically, our experiments reveal that the utility assigned by UCT to a trap move is often very close to the utility assigned by it to the best move. This suggests that even if UCT ranks a trap move somewhat lower than the best move, in general it has trouble distinguishing really good moves from really bad ones.

Traps sprinkled throughout the search space at various levels are, of course, extreme examples of bad moves, and it appears very likely that UCT will choose perhaps a less extreme bad move (a "soft trap") during the full length of the game. This, we believe, is a major reason why Minimax based strategies have been much more successful than UCT in games like Chess, even though UCT currently clearly dominates computer Go.

The remainder of this chapter is organized as follows: in Section 4.1, we formally define the concepts of states at risk and search traps. In Section 4.2, we demonstrate the existence of search traps in both randomly generated Chess positions, and those that arise in games between Grandmasters. Section 4.3 highlights convergence issues that plague UCT when confronting search traps, using examples drawn from the settings of Chess and Mancala. Section 4.4 uses the insights gleaned from the rest of the chapter to propose a novel hybrid search procedure, that outperforms both vanilla Minimax and UCT in Mancala, by combining elements of both. We then conclude with a brief discussion of our findings.

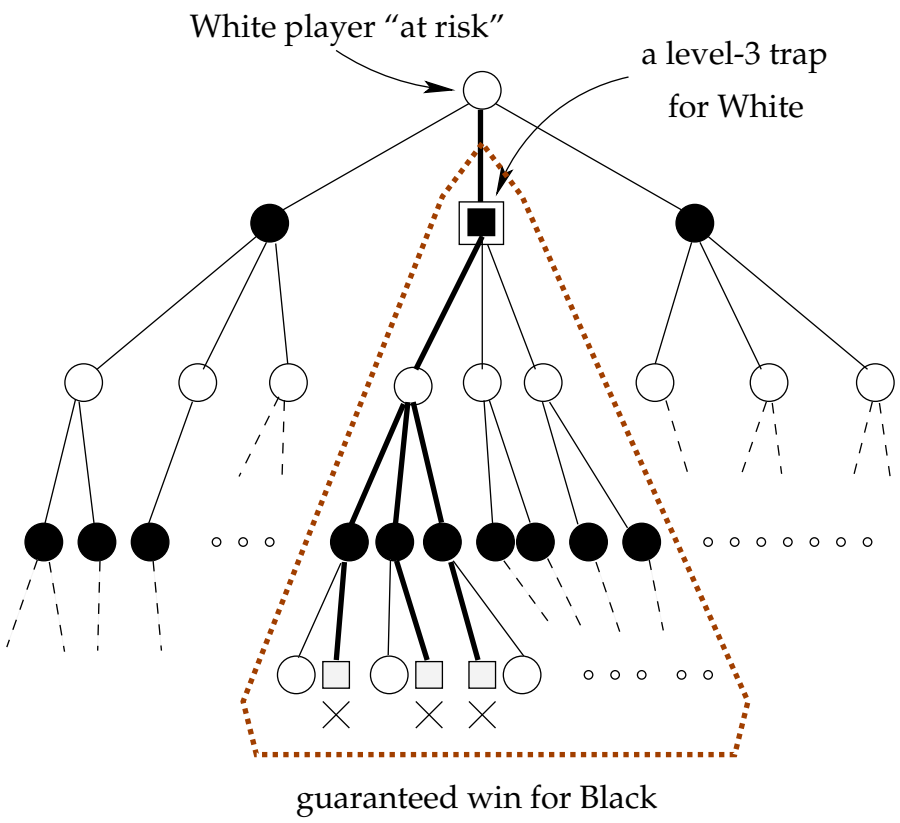

Figure 4.1: A level-3 search trap for the White player. The shaded square boxes at the leaves denote terminal nodes indicating a win for Black.

4.1 What is a Search Trap?

The question we seek to address is: *how often do adversarial search spaces exhibit traps and how likely is a player to fall into such a trap?* The notion of search traps and states at risk is illustrated in Figure 4.1 and a formal definition is given below.

Definition 1. *Let S be the state space of a two-player game, with the set of actions available to the players denoted by A. A k-*step winning strategy *for a player p is a policy function $\pi_k^p : S \mapsto A$ which, if implemented by p, guarantees him a win in k steps*

In other words, π_k^p prescribes an action for player p to follow at the initial

state, and then at every state thereafter that results from the counter-moves of his opponent, which if followed results in a win for p in exactly k plys. This occurs *regardless* of any move that the opposing player chooses to make.

Definition 2. *In a two-player game G, the current player p at state s is said to be* at risk *if there exists a move m from state s such that after executing m, the opponent of p has a k-step winning strategy. The state of the game after executing m is referred to as a* level-k search trap *for p.*

Clearly, a search trap is of concern only when the trap is computationally detectable by the opponent. In principle, *every* move leads to a position where either the current player has a winning strategy, or his opponent does (or the optimal line of play leads to a draw). However, if the winning strategy for the opponent is so complex that he cannot compute it with reasonable resource bounds, then making the move that leads to that state should not be viewed as falling into a dangerous trap. Therefore, what we are interested in is *traps that are identifiable with a reasonable amount of computation by the adversary*. This is quantified by the *level* associated with each trap, indicating the number of steps in the opponent's winning strategy.

We will be interested in relatively shallow search traps, typically those at levels 3, 5 or 7. Further, a trap will be of even more interest when avoiding the unfortunate move associated with the shallow trap can actually lead to a much deeper game play and perhaps even a winning strategy for the current player. If the underlying adversarial search space does have traps of this nature, then it becomes critical for the player to identify and avoid such traps. This is where the difference between exhaustive algorithms such as Minimax and sampling-based algorithms such as UCT comes in—we demonstrate that UCT-style algorithms

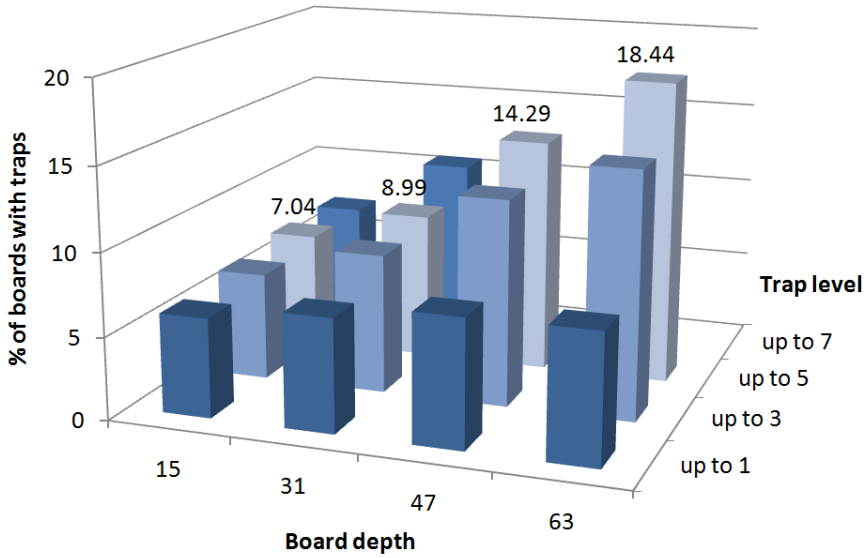

Figure 4.2: Percentage of Chess boards at various plys that have a shallow trap at level $\leq k$ on 200 semi-randomly generated boards.

have a good chance of missing even very shallow traps and thus losing the game.

4.2 Existence of Search Traps

Figures 4.2 and 4.3 show the percentage of Chess boards that were found to have traps at various levels. In Figure 4.2, we used 200 Chess games where each move was played randomly with probability 1/3 and selected by a GNU CHESS MM-8 search with probability 2/3. In Figure 4.3, we took 200 games that were actually played by Chess Grandmasters[1]. In each case, we played the game to 15, 31, 47, and 63 plys deep, and then computed whether the resulting board had a level-1, 3, or 5 trap (or 7 as well, in the case of semi-randomly generated 15-ply or 31-ply boards). The height of the bars in the plots indicate the percentage of

[1]Source: `http://www.chessgames.com`

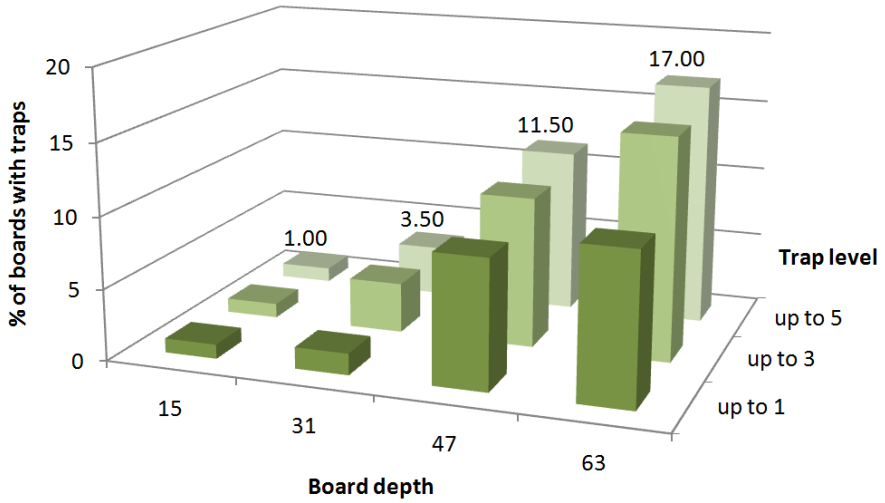

Figure 4.3: Percentage of Chess boards at various plys that have a shallow trap at level $\leq k$ on 200 positions sampled from games played between Chess Grandmasters.

the resulting boards that were found to have fairly shallow traps. For example, for semi-randomly generated boards that are 31-ply deep, roughly 7 – 10% of the games have a trap at levels 1 through 7. For games that are 47-ply deep, as many as 15% of the games have shallow traps. In games played by grandmasters, the percentage of traps is lower but still quite significant—roughly 12% for 47-ply games[2]. This shows that in games such as Chess, *surprisingly shallow traps are sprinkled throughout the search space, at essentially every depth of the game.* Hence, a good game playing strategy for such a search space must incorporate the capacity to recognize and avoid such traps.

Also, we observe that the proliferation of traps in a game is linked to the rules governing the game. In Chess, for instance, a win is defined through the capture of a certain key piece, the king. In Go, on the other hand, wins are

[2]Indeed, the difference is most pronounced in the early stages of the game where Chess theory and strategy have been extensively studied, and most games between expert humans follow safe, well-established lines

defined using the notion of territory. The former, where winning is defined by a function of some local features of the game, exhibits many more traps than the latter, where the winning criterion is a global function of the state. The game of Mancala falls into the latter camp, since the game only ends after all stones have been captured. Unsurprisingly, the Mancala game tree exhibits a trap distribution that mirrors the structure of Go, rather than Chess. In game positions generated by starting with a uniform random distribution of stones in pits, which are then played out to various depths by two MM-12 players, we discovered that the *first* level-5 or shallower trap states are encountered, on average, only 20-plys into the game. The proportion of level-5 or shallower traps in the overall set of boards reaches 5% only 35-plys into the game. Of course, given the reduced state space complexity of Mancala compared to Chess or Go, much deeper searches are possible here; nevertheless, we discovered that even the first level-11 trap in Mancala takes 12-plys to make an appearance, and one needs to consider positions 22-plys deep to reach a 5% density.

4.3 Identifying and Avoiding Traps

This section describes our empirical investigation of the question, *can UCT-style algorithms successfully identify and avoid shallow traps?* Clearly, a level-k trap can be identified by a Minimax search of depth $k + 1$ starting from the current state. We therefore give UCT an equivalent amount of search time (measured in terms of the number of nodes expanded) and study its behavior. In all these experiments, the exploration bias parameter for UCT is fixed at a value of 0.4, for this was empirically observed to produce the best game-playing performance.

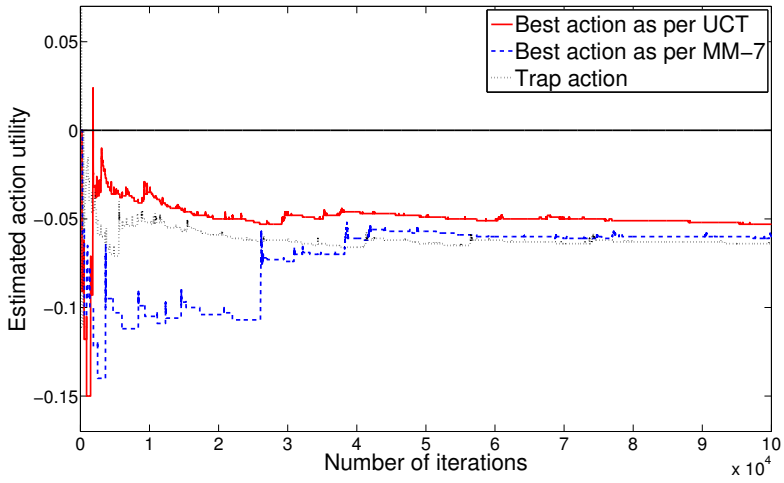

Figure 4.4: Utility estimates of UCT for the most promising actions (solid red at top and dashed blue) compared with that of a level-5 trap action (solid black initially in the middle).

For the first experiment, we consider the utility assigned by UCT to the action leading to the trap state. We compare this with the utility assigned by UCT to both the action it considers best, and the action MM-7 considers best. For Chess boards with White as the current player, the utilities are real numbers in the range $[-1, 1]$, with $+1$ indicating a guaranteed win for White (if played according to a winning strategy) and -1 indicating a guaranteed loss. Thus, the action a leading to the trap state should have a utility close to -1.

Figure 4.4 shows the evolution of the utility of these three actions as a function of the number of iterations of UCT, up to the first 100,000 iterations[3]. The board under consideration has a level-5 trap. Table 4.1 gives UCT utility values for three other boards as well, with traps at levels 1, 3, and 7, respectively. We make two observations from this experiment. First, the solid black line, corresponding to the utility assigned to the trap action, is typically far from being

[3]While the figure shows the outcome of a single run of UCT on a specific board, the observed behavior is fairly stable and reproducible across multiple runs and on different boards.

Table 4.1: UCT utility estimates for best action as per UCT, best action as per Minimax depth-7, and the trap action. Shown for varying trap depths.

	UCT-best	Minimax-best	Trap move
level-1 trap	−0.083	−0.092	−0.250
level-3 trap	+0.020	+0.013	−0.012
level-5 trap	−0.056	−0.063	−0.066
level-7 trap	+0.011	+0.009	+0.004

assigned the ideal utility of −1, which is also seen in the table except for the board with a level-1 trap. Second, for level-5 and level-7 traps, the UCT utilities for the trap action are very close to the utilities assigned to the *best* action (best as seen by either UCT or Minimax), to the point where it is not easy for the algorithm to distinguish between the two. As mentioned earlier, this suggests that UCT will have even more trouble distinguishing "soft traps" from good moves, and is likely to make moves that result in a significant material loss.

Figure 4.5 presents a utility evolution plot for a Mancala position and offers conclusive evidence of UCT's susceptibility to trap states[4]. In this example, the root node is a state at risk with a level-13 trap, and UCT is given access to a heuristic function rather than relying on playoutsm. While this may at first glance not fit the description of a "shallow" search trap, we note that the *maximum* branching factor in Mancala is only 6 and thus, a 13-ply deep search is

[4]We note that in this plot, the root node is actually a Min node and that the plot presents the *negated* utility estimates of the moves, to maintain consistency across plots and for improved readability

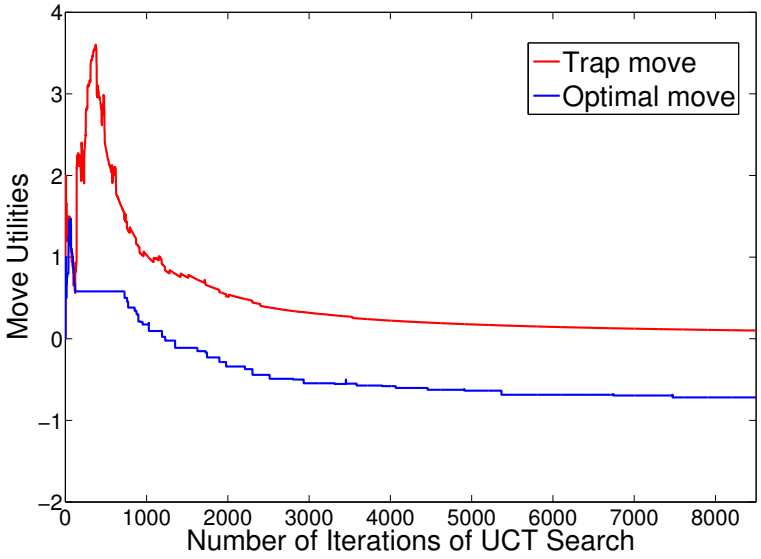

Figure 4.5: Trap state in Mancala that is erroneously picked by UCT to be the best move

actually a rather modest lookahead. With alpha-beta pruning, an MM-14 search at the root node establishes the value of the trap state after expanding only around 5000 nodes; UCT[5], on the other hand, mistakes the action leading to the trap state for the best action despite being allowed to use nearly twice the computational resources as that allotted to Minimax.

One reason that UCT assigns relatively high utilities to the trap states could be because it simply does not visit them frequently enough. *Could the estimate of UCT get better if it were allowed to have more samples from the trap state?* We again find that this is not the case. For example, in Chess, even if UCT is allowed to sample as many as 10 times the number of nodes Minimax needs to visit in order to identify the trap, the utility assigned by UCT to the trap state remains high — over −0.15 — as long as the trap is at level 5 or higher.[6] This is depicted in

[5]Technically, UCT_H, since we supply UCT with the same heuristic as the MM-14 player, rather than relying on playouts

[6]For a few level-7 traps, UCT did uncover the losing move in some runs after expending

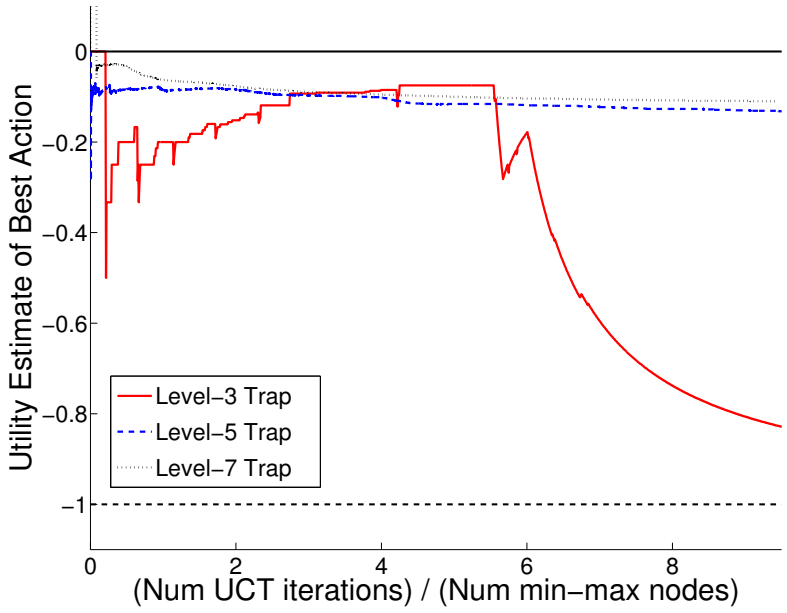

Figure 4.6: UCT estimate of the trap state stays incorrectly high, except for the level-3 trap (solid red curve that drops), even if the trap state is visited 10x times the number of nodes visited by the Minimax search identifying the trap.

Figure 4.6, which shows the evolution of the utility assigned by UCT to the trap state if UCT is started at the trap state itself, and thus forced to visit this state in every iteration. In the level-3 trap state (the solid red curve), UCT does identify the winning strategy and subsequently converges quickly to -1 as in the ideal case. However, for level-5 and level-7 traps, its utility remains inflated.

Finally, in order to better understand why UCT does not assign a utility even remotely close to -1 to trap states at levels 5 and higher even with 10 times the effort of Minimax search (and in some cases, even 50 times the effort), we took a deeper look at one of the main aspects of UCT: namely, that it searches relatively deeper in regions that it believes are promising. For a level-k trap, once Mini-

about 5 times the effort of Minimax search; nevertheless, it still failed on many runs.

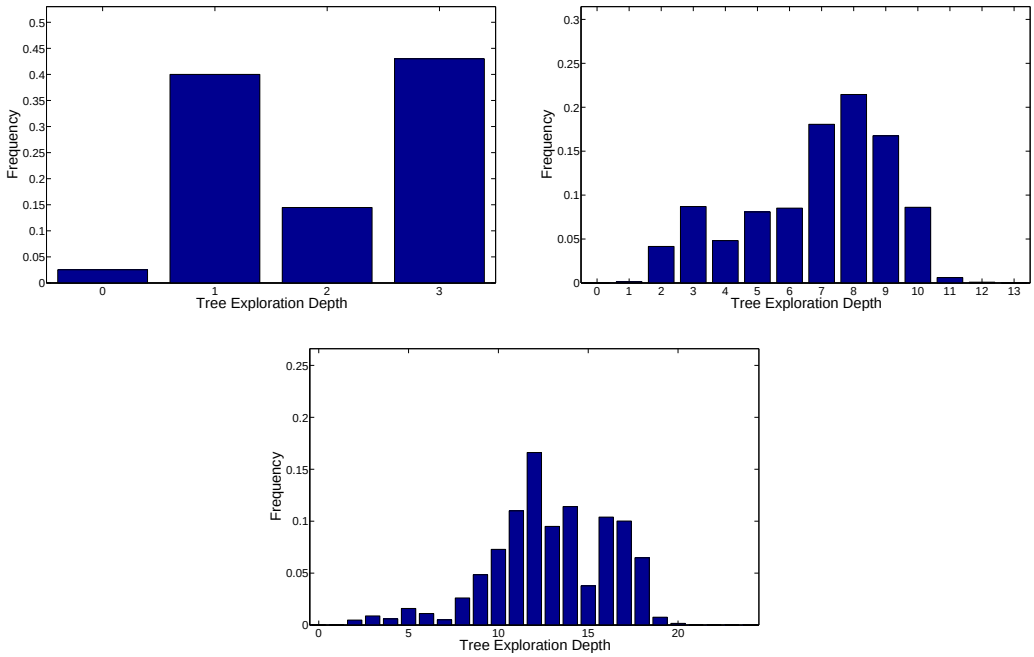

Figure 4.7: Histogram of search depths that iterations of UCT explored when in a trap state of depth 3, 5, and 7, respectively.

max search identifies the k-step winning strategy, it would never explore nodes deeper than depth k. On the other hand, as the histograms in the top-right and bottom panels of Figure 4.7 show, UCT spends nearly 70% of its time exploring nodes 5 plys or deeper in the presence of a level-5 trap, and nearly 95% of its time exploring nodes 7 plys or deeper (as high as level 20) in the presence of a level-7 trap. This is in stark contrast to the histogram in the top-left panel, which shows that UCT explores nodes only up to 3 plys deep in the presence of a level-3 trap, which it correctly identifies. In general, the amount of effort UCT spends beyond what it needs to (had it identified the winning strategy) appears to increase exponentially with trap depth. This demonstrates that UCT fails to identify the (shallow) winning strategy during these search experiments.

4.4 A Hybrid Strategy

Having established the fragility of UCT's decision-making in the presence of search traps, we now consider the flip-side of the coin: *does UCT outperform Minimax in search spaces lacking trap states?* We immediately run into a number of prickly issues when trying to resolve this question. Firstly, investigating this question using either Chess or Go as our testbed is hard: almost no part of the Chess search space appears to be trap-free, while the standard of play reached by Minimax in Go is so poor that it would not be possible to draw meaningful conclusions from a play-off against UCT. Moreover, while we had access to the ground-truth (i.e., the true Minimax values) when studying the estimated utilities of trap states, we would no longer have this facility and must therefore devise other measures of success. We circumvent the first problem by once again using Mancala as our testbed, where trap states are either completely absent, or appear fairly routinely, depending on the phase of the game. Moreover, both Minimax and UCT perform rather well in Mancala without the need for any extensive enhancements. To tackle the second problem, we use actual game-playing strength as our measure of success, but within the setting of the "partial" game that we introduced in Section 3.2.5. We now describe our experimental approach in greater detail.

Our objective is to establish whether the reasons for UCT's dominance in games like Go are due to its ability to outperform complete search methods like Minimax in regions of the search space that do not contain too many terminal nodes. If this hypothesis is indeed true, then since the majority of the Go search space lacks terminal nodes, it is plausible that UCT builds up a large positional advantage in the early stages of the game (where methods like Minimax have

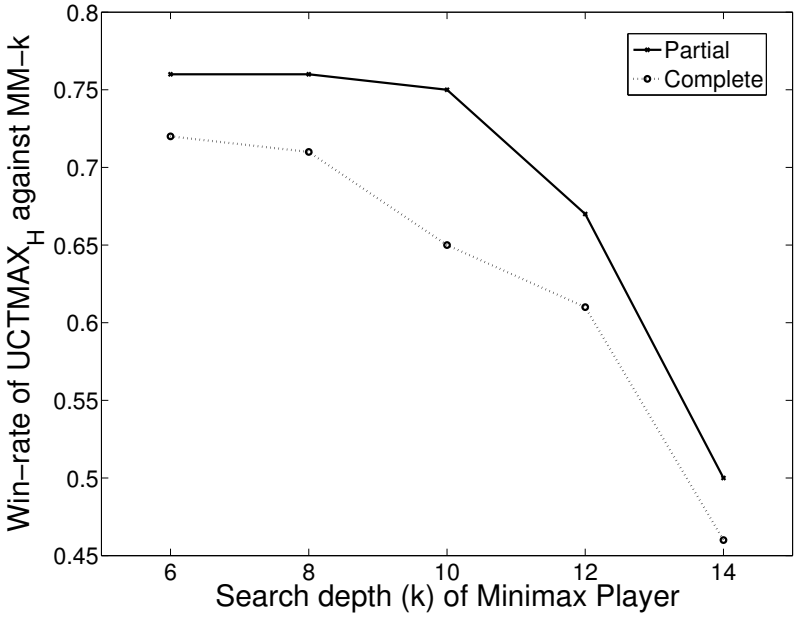

Figure 4.8: Win-rate of UCTMAX$_H$ against MM-k on partial and complete games of size $(6, 4)$

poor visibility) that its opponents cannot overcome later. We tested this theory by running partial games pitting UCTMAX$_H$, our best performing variant of UCT in this domain, against Minimax in which the two algorithms are played against each other until the ratio of terminal to non-terminal nodes in the search tree expanded by the Minimax player first exceeds 0.003. Thereafter, the game is completed by two MM-16 players. We do this for a range of Minimax opponents with different lookahead depths. We also repeated the experiment on a larger Mancala board, with 8 pits and 6 stones per pit (denoted $(8, 6)$). We randomize the trials, control for the amount of search effort and compute the win-rates of the algorithms in the fashion that was described in Section 3.2.2. The results are presented in Figures 4.8 and 4.9. For comparison, we also include win-rate data for UCTMAX$_H$ on complete games between the two players.

The key observation here is that, in both board sizes, and independent of

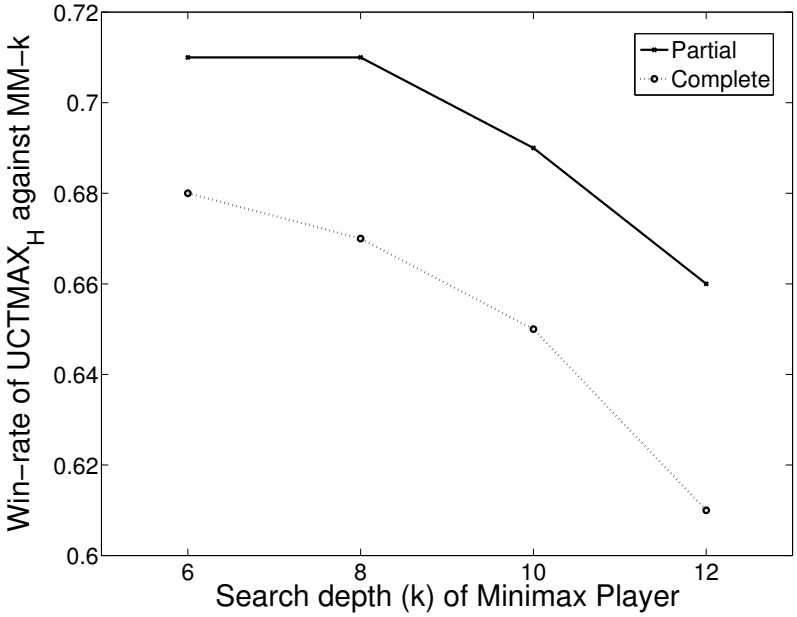

Figure 4.9: Win-rate of UCTMAX$_H$ against MM-k on partial and complete games of size $(8, 6)$

the lookahead depth of the Minimax player, UCTMAX$_H$ always does better in the partial games than the complete games. We have found that this trend holds (though not as pronounced) even when the standard UCT implementation, with averaging back-ups, is used. This supports our hypothesis that UCT is a better predictor of good moves in regions of the search space in which terminal nodes are largely absent.

Another trend that is apparent from Figures 4.8 and 4.9 is that the win-rate of UCTMAX$_H$ declines in the partial games with increasing k. There are two reasons for this. As we increase k, the terminal node ratio threshold of 0.003 is reached faster by the Minimax player. For example, in $(6, 4)$-size games between UCTMAX$_H$ and MM-6, the switch-over to the MM-16 playout phase occurs 21 plys into the game on average. In contrast, against MM-14, the threshold is exceeded after only 8 plys, which hardly gives UCT sufficient opportunity to steer

the game decisively in its favor. Secondly, even when terminal nodes are not directly visible, they make their presence felt more strongly to deeper searching players by influencing the heuristic estimates of nearby positions. This effect is diluted when the depth of the lookahead is smaller.

Finally, we note that there is an alternative interpretation of these partial game results — they can be viewed as *complete* games of Mancala being played by a hybrid UCT-Minimax agent against a standard Minimax opponent. The hybrid agent uses UCT for guidance in phases of the game where traps are absent, and switches to Minimax once traps become widespread. The results demonstrate that this hybrid approach outperforms either algorithm on its own, and highlights the potential for success of such hybrid approaches in other challenging application domains.

4.5 Discussion

In this chapter, we provided new insights into the nature of adversarial search spaces, specifically in terms of the existence of shallow search traps — positions which, if reached, guarantee the opponent victory within a few short steps. We demonstrated that the performance of UCT was sensitive to the presence of these traps. In Chess and Mancala, UCT's computed utility estimates of the actions leading to trap states were often so close to that of the best moves, that it was easy to confuse the two. Indeed, we presented a Mancala example where UCT picked an action that led to a level-13 trap. Then, using a novel partial game setting, we demonstrated the strength of UCT-style approaches in decision making tasks in search spaces that have a low density of terminal positions.

Taken together, these results offer an explanation for the success of UCT in domains such as Go and expose its limitations in domains such as Chess. They also offer up the intriguing possibility of hybrid adversarial search algorithms that can combine the strengths of both UCT and Minimax to produce stronger game-playing agents.

CHAPTER 5

A COMPARATIVE STUDY OF UCT AND MINIMAX IN SYNTHETIC GAMES

One of the goals of this thesis has been to provide a characterization of the structural properties of adversarial search domains that make them more or less suited to MCTS based planning approaches. Our tack in previous chapters has been to study this problem in specific real-world domains such as Chess and Mancala. While those results have been useful in building an intuitive understanding of the strengths and weaknesses of UCT, the domains themselves are too complicated for any mathematical analysis or more controlled experimentation. In this chapter, we address this issue by studying the *convergence time* and *decision accuracy* of UCT and Minimax (i.e., how frequently these algorithms make the optimal decision at the root node of the search tree) on two families of synthetic games. In doing so, we provide fresh insights into the processes that aid and hinder the performance of sampling-based planning techniques like UCT. While prior theoretical analyses of the behavior of UCT have focused on proving asymptotic convergence properties [56] or runtimes in some pathological examples [30], we focus on simplified versions of "typical" use cases for UCT.

We begin in Section 5.1 with a brief survey of synthetic tree models that have been studied by researchers in the past. In Section 5.2, we present our first new family of synthetic games that feature *critical levels*. These trees model common scenarios in tactical games like Chess where a player's winning strategy is constrained at some crucial points in the game, but is more forgiving elsewhere. We demonstrate empirically, and analytically, that the convergence time of UCT in

these kinds of games scales exponentially with the depth of the critical choice levels. Moreover, we also show that the runtime of UCT in this domain can be decomposed additively into the time spent between critical levels. Finally, we demonstrate how the averaging backup operator that is utilized in most UCT implementations can create problems for UCT by prematurely "freezing" onto sub-optimal lines of play.

In Section 5.3, we present a different model for generating synthetic games, that we term *Bounded Prefix Value (BPV)* trees. Inspired by a recent model proposed by Furtak and Buro [39], BPV trees correct some shortcomings that are common in many previously studied game tree models. We compare the decision accuracy of Minimax and UCT in the BPV games, while experimenting with various heuristic models and means for introducing trap states. We demonstrate how UCT is robust with respect to heuristics that are built around the commonly assumed Gaussian noise model, while Minimax is not. However, Minimax comes into its own when a heuristic exhibits *dispersion lag*, a form of systematic bias, while UCT performs poorly. Finally, we note that randomly placed trap states do not hamper UCT, but when sown carefully (in conjunction with a biased heuristic), they can hurt the decision accuracy of UCT. We conclude the section with a case study from the domain of Mancala that justifies many of our model's assumptions.

5.1 Related work

Numerous synthetic tree models have been proposed and studied in the adversarial search literature. They may be broadly classified into two categories: bottom-up and top-down models. In the bottom-up approach to tree construc-

tion, the values of the leaves of the tree are carefully specified, either as win/loss values [76] (known as P-games) or as real numbers [61]. The properties of the tree arise organically from the distributions used to set the leaf node values. P-games were the subject of much interest in the early 80s [72, 77] as researchers sought to understand why deeper Minimax searches worked in practice. While the model's relative simplicity allows for rigorous mathematical analysis, it also suffers from a number of drawbacks. Firstly, computing the value of the game and the Minimax value of the internal nodes requires search, and that all the terminal nodes of the game tree be retained in memory. This restricts the size of the games that may be studied. Secondly, the model makes the strong assumption that the value of any leaf node is independent of the value of any other leaf node — an assumption that does not hold in many real games, where sibling values tend to be closely correlated. Finally, P-game trees exhibit *lookahead pathology*, a situation where deeper Minimax searches lead to worse decision making [72, 77]. While this is an intriguing phenomenon, it is not a feature of real-world games where deeper searches consistently lead to better performance and have been key in attaining human-level performance in many domains [65, 24].

In the top-down approach to tree construction, each internal node of the tree maintains some state information that is incrementally updated and passed down the tree. The value of a leaf node is then determined by a function of the path that was taken to reach it. For example, in the models studied by Nau [73], and Scheucher and Kaindl [93], values are assigned to the edges in the game tree and the value of a leaf node is determined by the sum of the edge values on the path from the root node to the leaf. By introducing correlations among sibling nodes, these models mirror the structure of real world games more closely. These correlations also eliminate the lookahead pathology. However, search is

still required to determine the true value of internal nodes, thereby only allowing for the study of small games.

More recently, Furtak and Buro introduced the *prefix value* tree model [39], which extends the top-down model proposed in [93]. The key insight exploited by this new model is the observation that the Minimax value of a node can never *increase* for the player on move. Setting the values of nodes while obeying this constraint obviates the need for search, allowing us to analyze and simulate arbitrarily large games. However, this elegant model suffers from a surprising drawback — naïve algorithms that combine limited lookahead with purely random playouts perform extremely well on these trees. We explore this issue in greater detail in Section 5.3.

A final notable model is that proposed by Long et al. [59], which was used to study the performance of Monte Carlo methods in imperfect information games. While their model itself isn't suited to our task (since we are interested in deterministic, perfect information games), our work shares the same spirit in that it seeks to understand the factors that affect the success of UCT vis-a-vis Minimax search in controlled adversarial settings.

5.2 Trees with Critical Levels

UCT's behavior is rich and complex in adversarial search spaces such as those of Chess and Go. In order to better understand its behavior, we first consider a synthetic adversarial search space where we vary, in a controlled manner, key properties that affect the performance of UCT.

We consider game trees with implanted winning strategies for the maximizing player (denoted Max), who is assumed to be on move at the root node. The winning strategies are parametrized by the number of *critical decision levels* and their depths. If Max makes the correct action choice at every critical node, then regardless of the actions chosen by either player at all other nodes, the payoff at the end of the game is +1. If Max chooses an incorrect action at any of the critical nodes, then the payoff at the end of the game is drawn uniformly from $\{-1, 0, +1\}$. This simple model captures the notion of winning plans that exist in many tactical games like Chess, where from a given state, a player can force a win by executing a sequence of a few clever moves.

In these experiments, we are interested in the time UCT takes to "discover" the winning strategy for Max, which we define in terms of the utility assigned by UCT to the root node. Once UCT has settled on a winning sequence of moves for Max (i.e., a principal variation), it will exploit it on subsequent iterations and this will force the utility of the root node to approach +1. A subtle point here is that the minimizing player (denoted Min) might continuously force Max to explore different lines of play; nonetheless, the optimal paths for Max will all be equally good and the value of the root will still approach +1.

Formally, let $v(t)$ be the utility assigned to the root node of the search tree after t iterations of UCT. For a single UCT search, we define the *τ-convergence point* t^* as the smallest t such that $v(t) \geq \tau$ for all $t \geq t^*$. We say that UCT has *τ-converged* if the current iteration number is at least t^*. Unless otherwise specified, we will simply use the term converged to imply τ-convergence at the root, with $\tau = 0.7$.

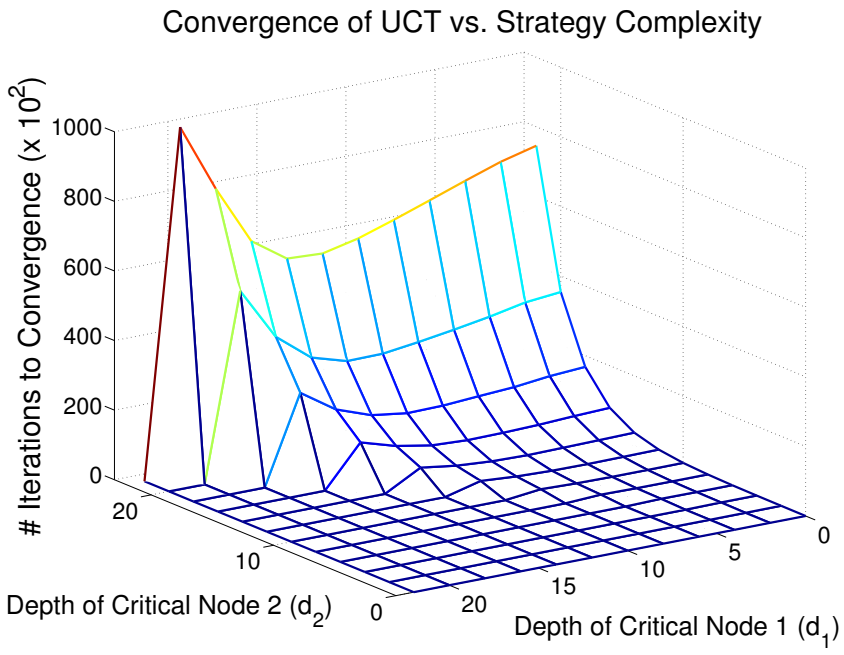

Figure 5.1: UCT convergence time as a function of the depths of the critical levels

5.2.1 Empirical Observations

Figure 5.1 illustrates how the time UCT takes to converge in the presence of two-step winning strategies (i.e., strategies with 2 critical levels) in a 24-level binary tree varies as a function of the depths of the two critical nodes (hereafter referred to as d_1 and d_2, with $d_2 > d_1$). The convergence times are an average computed over many hundreds of trees. Note that in the mesh plot, the area of interest lies beyond the $d_1 = d_2$ line, towards the back of the plot. The panels in Figure 5.2 depict slices of this surface obtained by fixing d_1 and d_2, respectively.

As seen in the left panel of Figure 5.2, for a fixed d_1, the convergence time of UCT is essentially exponential in d_2. The dependence of the convergence time on d_1 is more intriguing—with a fixed d_2, UCT appears to perform best when d_1 is slightly more than half of d_2. This "dip" in the curve is captured by the

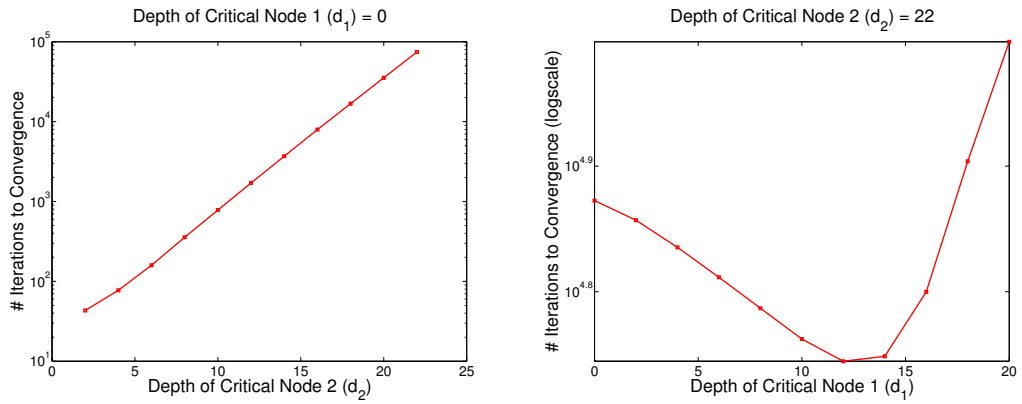

Figure 5.2: Slice of surface in Figure 5.1 with a fixed depth of critical level 1, in logscale (left), and slice with a fixed depth of critical node 2, in logscale (right).

following expression for the runtime of UCT, which we explain below:

$$UCT(d_1, d_2) = a \cdot N^{d_1/2} + b \cdot 2^{d_1/2} \cdot N^{(d_2-d_1)/2} \tag{5.1}$$

Here, $2 < N < 3$ (empirically estimated to be 2.37) and $a, b > 0$ are small constants. This expression fits the mesh plot in Figure 5.1 very closely and highlights a key property of UCT in the presence of multi-step winning strategies: *The runtime of UCT can be decomposed additively into the time spent between consecutive critical levels*. Specifically, UCT first explores roughly $2^{d_1/2}$ "active" nodes at level d_1 in time $O(N^{d_1/2})$, then explores each of the roughly $2^{d_1/2}$ subtrees below these active nodes at level d_1 in time $O(N^{(d_2-d_1)/2})$ each to identify roughly $2^{(d_2-d_1)/2}$ active nodes at level d_2 in each subtree, and so on down to other critical decision levels. The quantity $2^{d_1/2}$ (in general, $2^{(d_i-d_{i-1})/2}$ per subtree) representing "active" nodes is simply the minimum number of nodes that Min can continually force Max to explore, until Max has determined a winning sequence from all of these nodes. In general, we can extend this reasoning to k critical decision

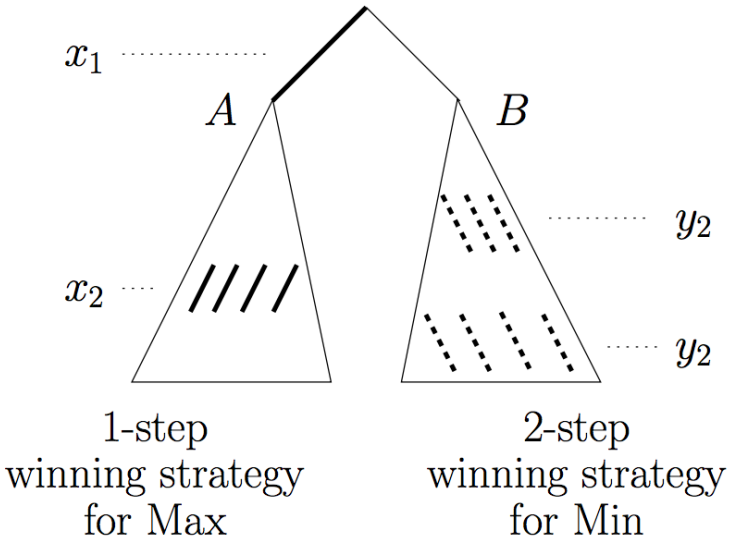

Figure 5.3: Synthetic binary tree with implanted winning strategies for both Max and Min.

levels, suggesting that the runtime of UCT is captured by:

$$UCT(d_1, d_2, \ldots, d_k) = \\ a \cdot N^{d_1/2} + b \cdot 2^{d_1/2} \cdot UCT(d_2, \ldots, d_k) \tag{5.2}$$

Note that Max takes $N^{d_1/2}$ iterations, and not $2^{d_1/2}$, to identify the $2^{d_1/2}$ active nodes at level d_1 that Min can force it to. This is because, although Max ideally has the choice to "freeze" on to any one of its equally good children, the exploration constant forces Max to explore to some extent the other child as well, especially during the initial few visits to that node. Nevertheless, the overall time is much less than the size of the full search tree till this level, which is 2^{d_1} or $4^{d_1/2}$.

In our second experiment, we study a more complex scenario where both Max and Min have implanted strategies and a few initial samples provide in-

Max's Strategy Depth	Min's Strategy Depth					
	Shallow		Mid-level		Deep	
	F	U	F	U	F	U
Shallow	36	148	77	610	560	3800
Mid-level	950	1000	1500	1900	7900	13k
Deep	16k	16k	17k	17k	30k	33k

Table 5.1: Effects of the depths of Max and Min's strategies on UCT's convergence time. 'F' and 'U' denote instances with favorable and unfavorable initial estimates, respectively.

correct guidance at the root (see Figure 5.3). In particular, we study binary trees of depth 20 where Max has critical nodes at depths (x_1, x_2) where $x_1 = 0$, $2 \leq x_2 \leq 18$, and x_2 is even, and Min has critical nodes at levels (y_1, y_2), where $1 \leq y_1, y_2 \leq 19$ and y_1, y_2 are odd. In order to win, Max must move left at the root and again at level x_2; if Max goes right at the root, then Min can force a win by going left at levels y_1 and y_2 (i.e., the right child of the root is a trap state for Max). Let A and B be the subtrees rooted at the left and right children of the root node respectively. We bias the values of the leaves that are not on a winning path for either player such that the average of the values of the leaves in A is 0, while the average of the values of the leaves in B is 0.5. Thus, the B subtree, though ultimately a losing proposition for Max (assuming optimal play by Min) will look more promising with limited sampling[1]. We now ask the question, *how do the depths of the strategies for the two players influence UCT's convergence time?*

Table 5.1 presents our findings based on an average of 100 UCT runs on a fixed tree. On its first few iterations, UCT receives extremely noisy estimates of the utilities of its two children at the root. In the best or "favorable" case,

[1]Though this misdirection may sound like an artificial construct, we will show in Section 5.3.4 that these scenarios *do* arise in real games, particularly in the vicinity of trap states

Max's Strategy Depth	Min's Strategy Depth					
	Shallow		Mid-level		Deep	
	F	U	F	U	F	U
Shallow	33%	21%	34%	30%	40%	36%
Mid-level	3%	4%	14%	16%	31%	31%
Deep	0.2%	0.3%	1.8%	2%	16%	16%

Table 5.2: Effects of the depths of Max and Min's strategies on the distribution of visits to the right subtree.

these initial estimates correlate correctly with the true utilities of the children and Max chooses to explore subtree A first. In the unfavorable case, the child rankings are reversed and Max chooses to explore subtree B first. Note that any ties will eventually resolve one way or the other, and at that point, we fall back on one of these two cases.

There are a number of interesting trends in Table 5.1. First, when estimates are unfavorable at the root, the time to convergence is greater as UCT initially "wastes" time in subtree B until it (at least partially) uncovers Min's winning strategy. Second, this gap in convergence time is most pronounced when either Max has a shallow winning strategy or Min has a deep winning strategy. This too makes sense; in the former case, UCT can uncover Max's strategy very quickly if given the chance, and hence the time "wasted" in subtree B counts relatively much more; in the latter case, UCT simply needs to work harder to uncover Min's winning strategy and switch to subtree A.

Finally, we note that increasing the depth of Min's strategy slows down UCT's convergence *even in the favorable instances*. The data in Table 5.2, which shows the average percentage of time UCT spends in subtree B during the runs

presented in Table 5.1, helps explain this phenomenon. As Min's strategy is implanted deeper down in the tree, UCT spends more time exploring subtree B. A by-product of this repeated sampling from B is that the estimated utility of the root node is now heavily biased by the samples drawn from B; when UCT eventually switches to subtree A and discovers Max's winning strategy, it needs to work extra hard to overcome this bias and reinforce the true utility of the root (we will formalize this in Section 5.2.2, equation (5.3)). This is illustrated by the fact that when both Max and Min have shallow strategies, when UCT converges, the root node of subtree A has a typical utility estimate of 0.720; when both have deep strategies, the root node of subtree A needs to reach a much higher target value of 0.850.

This highlights an important shortcoming of UCT, namely that it can be overly optimistic in its estimates of state utilities, that lead it on wild goose chases. By the time it discovers that an action it has been exploring is sub-optimal, nodes higher up the tree have been reinforced with so many samples that it faces an uphill task in changing these estimates. In the face of computational constraints (for example, in a timed game-playing setting such as Blitz Chess), this is particularly troublesome for it means that UCT could easily have spent its time exploring sub-optimal moves and thus faces a very real risk of falling into a trap state.

5.2.2 Analytical Insights

While a few attempts have been made to analyze MCTS methods in general and UCT in particular [9, 56, 30], these analyses are based on the worst case sce-

nario and, in essence, boil down to showing that an exponential (or even super-exponential [30]) number of iterations are necessary and sufficient for UCT to converge to true Minimax values. These exponential time convergence results, while intricate and interesting, do not explain the success of UCT in practice in domains such as Go, with a practically limited number of iterations available during game play. In contrast, our goal in this section is to provide a methodology for analyzing some simple scenarios where UCT *does* work, and obtain insights into its runtime behavior. Specifically, we will consider two-step winning strategies implanted in binary trees.

We highlight three take-away messages, some of which have previously been observed empirically and are derived here analytically:

1. the averaging backups of UCT can make recovering from poor early choices very costly,
2. UCT in two-player settings scales exponentially with the depth of the critical choice points, whereas in single-player settings, all that matters is the number of critical choice points, not their depth, and
3. the constant tension between exploration and exploitation as controlled by the exploration constant.

In order to simplify the analysis while still retaining the essential aspects of UCT, we work with a modified version of the algorithm in this section. Instead of implementing the UCB1 exploration-exploitation strategy, we will use an ϵ-*greedy* version of the algorithm, where $\epsilon \in [0, 1]$ is a constant determining how often sub-optimal moves are explored. Specifically, when exploring a node for the first few times, UCT simply visits all children once (a "round"), as usual.

However, after this round, it selects an optimal branch (breaking ties at random) with probability $1 - \epsilon$ and a sub-optimal branch (breaking ties at random) with probability ϵ. Auer et al. showed that this simpler bandit algorithm has similar asymptotic regret bounds as UCB1, so long as ϵ decreases linearly with the number of times the node is visited [9].

We make one further modification, where instead of dealing with tie-breaking, we assume that rounds similar to the first round are repeated (i.e., all children explored in each round) until ties are broken. For binary trees, which will be the main focus of this section, this modification does not make a significant difference.

Scenario A

For ease of illustration, we start with the simplest case and build upon it. Consider a binary game tree T with Max on play at the top node. Let T^L and T^R denote the left and right subtrees, respectively, of T. Suppose that all leaves of T^L are labeled +1, i.e., Max has a sure win if he makes the left move. Suppose also that a p fraction of the leaves, where $p \in [0, 1)$, of T^R are labeled +1 and the rest are labeled −1. This tree is depicted in Figure 5.4(a), with bold edges corresponding to winning strategy moves. *How long does it take for UCT to identify the left branch as the winning move?*

In a given round at the root node of T, a playout from the left child always leads to +1 while a playout from the right child leads to +1 with probability p. Therefore, we have a tie with probability p and it follows that the expected number of rounds needed to break the tie is $1/(1 - p)$. Hence, the total number

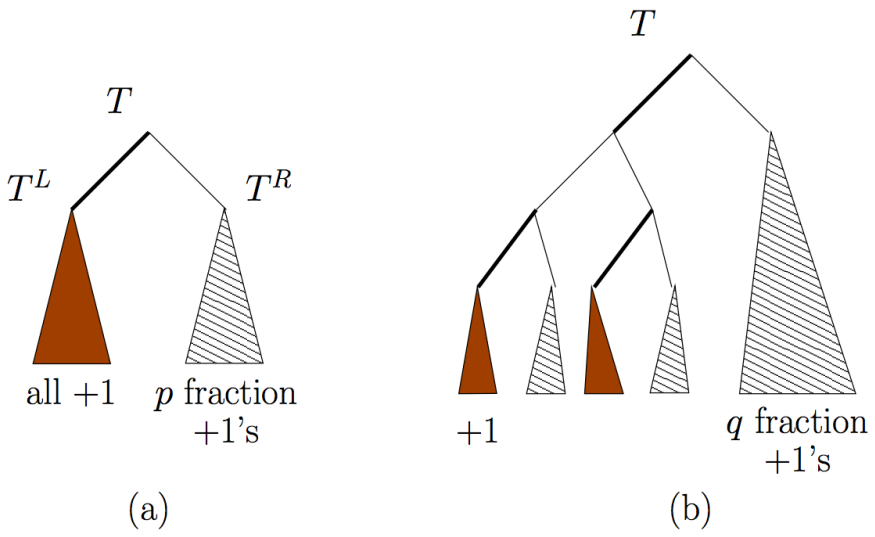

Figure 5.4: Synthetic binary trees with implanted winning strategies for Max. (a) 1-step winning strategy. (b) 2-step winning strategy.

of visits needed to the root node of T in expectation equals $2/(1-p)$ (as there are 2 visits per round) plus the time it takes for UCT to converge at the left child after the tie is broken. Note that the only way for the tie to be broken in this tree is to have all +1 playouts on the left and exactly one −1 playout on the right, implying that the left move will necessarily be identified as the optimal move when the tie is broken. (This will not be the case in general, as we discuss later.) From this point on, T^L will be visited a $1-\epsilon$ fraction of the times the root node of T is visited. Let $C(\tau, \mathit{value}, \mathit{iter})$ denote the number of visits needed to the winning strategy node (in this case the root node of T^L) for UCT to τ-converge at the root node of T, where *value* denotes the current value of the node and *iter* denotes the number of visits already made to the node; due to the "averaging" backups of UCT, the current state of the node significantly affects the time to convergence even after a winning strategy has been identified, and we will quantify this shortly. The number of visits needed to the root of T is

therefore roughly $2/(1-p) + C(\tau, value, iter)/(1-\epsilon)$.

How do we determine $C(\tau, value, iter)$? In the unlikely case that the current value, *value*, is already at least as good as τ (i.e., $value \geq \tau$ for Max), this quantity is 0. Otherwise, assuming subsequent visits explore the identified winning strategy, resulting in all +1 playout values, the averaging nature of backups dictates that:

$$\frac{(C(\tau, value, iter) \times 1) + (iter \times value)}{C(\tau, value, iter) + iter} = \tau$$

$$\implies \quad C(\tau, value, iter) = iter \times \frac{\tau - value}{1 - \tau} \tag{5.3}$$

In our case, $iter \approx 1/(1-p)$ and $value = 1 - 2/iter$ as all but the very last round should result in playout values of +1. Plugging these values in, the number of visits to the root node of T till convergence is roughly:

$$\frac{2}{1-p} + \frac{1 + \tau - 2p}{(1-\epsilon)(1-p)(1-\tau)}$$

Remark 1. *Equation (5.3) points out an interesting limitation of UCT that we have already encountered near the end of Section 5.2.1, namely, that the* averaging backups of UCT can make recovering from poor early choices very expensive. *In particular, if iter is high and value is too low (for Max), then UCT will take a long time to make up for its mistakes before it reaches τ. This suggests there might be other backup strategies, although finding an effective alternative backup strategy requires further study because natural choices such as simple Minimaxing tend to be very brittle.*

Scenario B

We now explore the "tension" between having a small value for the exploration constant, ϵ, and a large value. This example will also illustrate that the depth of the critical nodes of a winning strategy *exponentially influences* the number of

iterations needed for convergence. This is in stark contrast to k-step winning strategies in *single-player settings*, where it is easy to argue that the depth of the critical choice points is immaterial, and all that matters is the number of critical choice points. Intuitively, the difference between the single player and two players settings is that in the former case, since all choices look equally good (or bad) at non-critical points, the player can arbitrarily "freeze" on one of them and keep exploiting it, while in the two player setting, the opponent prevents this freezing by continually forcing the winning player to different areas of the search space in the hope of avoiding defeat. For example, for a depth d winning strategy, the losing player can force the other player to explore precisely $2^{d/2}$ paths.

Suppose that T is modified so that the strategy embodied by T^L in Scenario A is actually hidden deeper and that Max needs to make one good move to get to this strategy. Specifically, we now have a 2-step winning strategy for Max, with critical moves at levels 0 and 2, with the subtrees at level 2 being identical to the ones in Scenario A. Also, let us suppose that the right subtree of the root node has a fraction q of +1 leaves, which will affect tie breaking at the root. This is depicted in Figure 5.4(b).

Given the expression derived above for Figure 5.4(a) for the number of times we need to visit each of these subtrees at level 2 in order to identify the winning strategy from there on, how many times do we need to visit the root node of the tree to achieve this? First, consider a node X one level above a winning strategy at level 2. Min is on move at X, which means that as soon as Max begins to identify the winning strategy on the left branch of X, Min has an incentive to switch to the right branch of X (i.e., what's good for Max is bad for Min). In other

words, *Min will keep switching* between the two choices until Max has figured out the winning strategy under both choices of Min. This means that the number of visits to X that we need is *twice* the number of visits to each of T^L; in general, when Max's winning strategy is at depth d, the number of visits needed will be 2^d times the number of visits to any single "winning" subtree at level d—hence the exponential scaling with the depth of the winning strategy.

Further, the tie at the root node of T may now be broken in favor of the right child as well, as there are leaves labeled -1 on both sides. If the tie breaks in favor of the left child (the "favorable" case), then the number of iterations needed after breaking the tie is:

$$D(favorable) \approx \frac{2}{1-\epsilon} \times \left(\frac{2}{1-p} + \frac{C(\tau, value, iter)}{1-\epsilon} \right)$$

the latter part of which is similar to Scenario A, multiplied by 2 for twice the work that needs to be done due to Min's choice at level 2, and divided by $(1-\epsilon)$ since in the favorable case we will visit the left subtree of T this fraction of the times we visit the root node of T.

More interestingly, when the tie at the root is incorrectly broken in favor of the right hand side child at the root (the "unfavorable" case), the left subtree is visited only an ϵ fraction of the time, implying that many more visits to the root node are needed in order to achieve the same number of visits as before to the strategy nodes at level 2. Specifically, the number of iterations needed after breaking the tie is:

$$D(unfavorable) = \frac{2}{\epsilon} \times \left(\frac{2}{1-p} + \frac{C(\tau, value, iter)}{1-\epsilon} \right)$$

This illustrates, in a concrete fashion, the tension between small and large values of ϵ, when the goal is to minimize the number of visits to the root node

to achieve convergence; see Figure 5.5 for an illustration where $p = 0.5$ and the C value is taken to be 10.

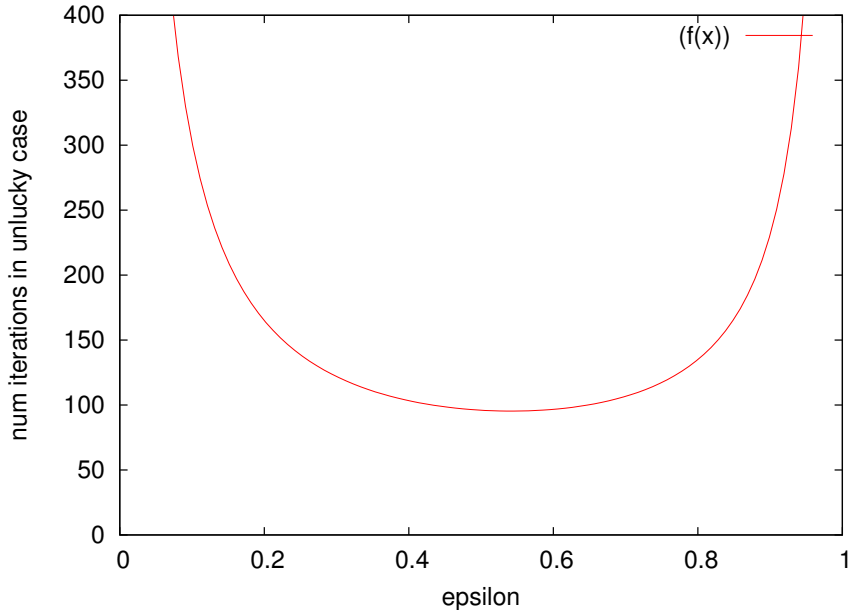

Figure 5.5: The effect of varying ϵ on convergence time.

Additionally, we must consider the time to break the tie at the root node, which is slightly more complex than in Scenario A. The fraction of +1 labeled leaves on the right is q and on the left is $p' = 3p/4$. Therefore, the probability of a tie is $p'q$ (when both playouts yield +1) plus $(1-p')(1-q)$ (when both playouts yield −1), giving $p'+q-2p'q$. Thus, the expected number of visits before the tie is broken is $2/(p'+q-2p'q)$. Further, when this happens, the tie is broken in favor of the left subtree with probability $p'(1-q)$ and in favor of the right subtree with probability $(1-p')q$. Putting all this together, we have the following expression for the rough number of visits needed to the root node:

$$\frac{2}{p'+q-2p'q} + \frac{p'(1-q) \times D(favorable)}{p'+q-2p'q} + \frac{(1-p')q \times D(unfavorable)}{p'+q-2p'q}$$

5.3 Bounded Prefix Value (BPV) Trees

While trees with implanted critical levels offer a useful means to model tactical situations in many games, they still suffer from a number of drawbacks. Firstly, like other top-down models, the size of the games that can be studied are limited by the fact that every leaf node in the tree needs to be stored. Moreover, while the lack of any structure in the game, beyond the critical levels themselves, enables us to carry out some interesting analysis, it is reasonable to question how well this model matches up to the intricate structure of real games. In this section, we instead propose and study a new top-down tree model that we dub *Bounded Prefix Value (BPV)* trees.

5.3.1 Prefix Value (PV) Trees

In 2009, Furtak and Buro introduced their Prefix Value tree model [39]. In a PV tree, the values of nodes are drawn from the set of integers, with positive values representing wins for Max and the rest indicating wins for Min. For the sake of simplicity, we disallow draws. Let $m(v)$ represent the minimax value of a node v. We grow the subtree rooted at v as follows:

- Let $V = \{v_1, v_2, \ldots, v_b\}$ represent the set of children of v, corresponding to action choices $A = \{a_1, a_2, \ldots, a_b\}$.
- Pick an $a_i \in A$ uniformly at random — this is designated to be the optimal action choice at v.
- Assign $m(v_i) = m(v)$. If Max is on move at v, then $m(v_j) = m(v) - k, \forall j \neq i$. If Min is on move v, then $m(v_j) = m(v) + k, \forall j \neq i$.

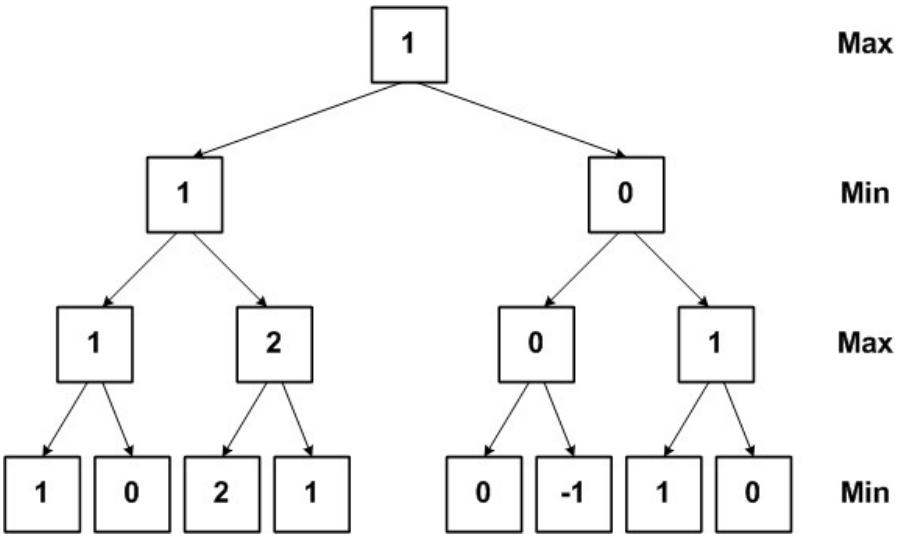

Figure 5.6: A sample PV tree of depth 3 and branching factor 2 where sub-optimal moves are assessed a constant penalty of 1.

Here, k is a constant that represents the cost incurred by the player on move for taking a sub-optimal action (though in general, it may be a random variable — indeed, we address this scenario later). The depth of the tree is controlled by the parameter d_{max} and a uniform branching factor of b is assumed. Figure 5.6 shows an example PV tree.

The PV tree model is an attractive object of study as, despite its relative simplicity, it captures a very rich class of games. However, it has one major drawback: a simple 1-ply lookahead search, using the average outcome of a large number (we use 1000) of random playout trajectories as a heuristic (i.e., MM-1-1000R), achieves very high decision accuracies. We now explore this phenomenon a little deeper.

Without loss of generality, we restrict our study from hereon to trees where Max is on move at the root node n. Moreover, we require that $m(n) = 1$ and that n has exactly one optimal child — this ensures that our search algorithms are

faced with a non-trivial decision at the root node. While we focus on the case where $b = 2$ in what follows, extending our results to higher branching factors is straightforward. We denote the left and right children of n by l and r respectively and assume that l is the optimal move. Define $S_d(v)$ to be the sum of the Minimax values of the leaf nodes in the subtree of depth d rooted at node v.

Proposition 1. *$S_d(l) - S_d(r) = 2^d$ for all $d \geq 0$, for PV trees.*

Proof. We proceed by induction on d. For the base case, $S_0(l) - S_0(r) = 1 - 0 = 2^0$ by definition. Assume the claim holds for $d = t, t \geq 0$, where t is a Max level. Then, $S_{t+1}(l) - S_{t+1}(r) = (S_t(l) + (S_t(l) - k)) - (S_t(r) + (S_t(r) - k)) = 2(S_t(l) - S_t(r)) = 2 \cdot 2^d = 2^{d+1}$. A symmetric argument can be made for the case where t is a Min level. □

Define $P(v)$ to be the average outcome of random playouts performed from the node v. If the subtree rooted at v has uniform depth d, then $\mathbb{E}[P(v)] = S_d(v)/2^d$. An immediate consequence of Proposition 1 is that $\mathbb{E}[P(l)] - \mathbb{E}[P(r)] = (S_d(l) - S_d(r))/2^d = 1$, for any depth d, i.e., with sufficient playout samples, the estimated utility of the optimal move l at the root will always be greater than that of r. In other words, the decision accuracy of MM-1-kR approaches 100% as we increase the number of playouts k, independent of the depth of the tree — a fact confirmed by our simulation results. Moreover, this effect persists even when k is drawn uniformly at random from the set $\{1, \ldots, k_{max}\}$ for each node in the tree. In simulations with $k_{max} = 8$, MM-1-1000R achieves a decision accuracy of $91.4\% \pm 1.7$ over a sample of 1000 randomly generated trees. Since such naïve planning approaches are seldom successful in real games, the PV tree model

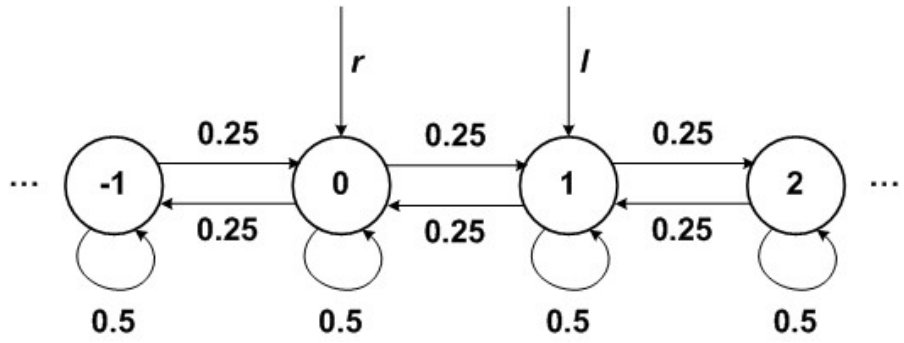

Figure 5.7: Markov Chain corresponding to performing a random playout on a PV tree with $k = 1$.

is clearly not realistic — but can we modify it in a simple way to correct this anomaly?

To answer this question, it is useful to visualize the possible two-step transformations that the value of a node in a PV tree can undergo as we descend it, in the form of a graph (as in Figure 5.7). For example, if we begin at a state with value 1, then after descending two plys in the tree, we could find ourselves at a node with a value in the set $\{0, 1, 2\}$ (assuming $k = 1$). A random playout on a depth d PV tree therefore translates to a lazy random walk of length d on this graph (i.e., the set of integers $\mathbb{Z}$), with transition probabilities as noted in Figure 5.7 (assuming $b = 2$). This offers an alternate way of understanding Proposition 1. It is easy to show that the average of the end-points of a large number of d-step random walks that begin at a value v is v; therefore, the average of random playouts that begin at l (the optimal child of the root node) will converge to 1, while those starting at r will converge to 0.

Thus, the chief problem with the PV tree model is that random walks retain *memory* of their initial conditions. This is because the Markov Chain corresponding to a finite random walk on an infinite graph is not irreducible, i.e., not every

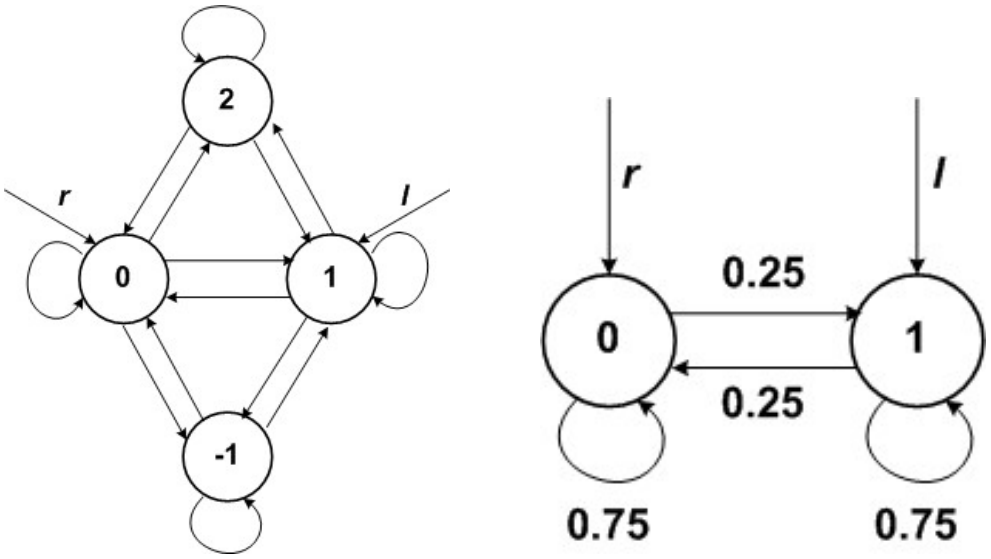

Figure 5.8: Markov Chain corresponding to performing a random playout on a PV tree capped to assume Minimax values from the set $\{-1, 0, +1, +2\}$ (left) and from the set $\{0, 1\}$ (right)

state in the graph is accessible from any other state within a finite number of steps, and therefore, the probability distribution over states after a d-step walk is not independent of the starting point (i.e., not *stationary*). The most straightforward way to fix this problem is to make the chain a finite size. In the PV tree model, this corresponds to *capping* the Minimax values that nodes can assume and not allowing them to grow arbitrarily large/small. Moreover, it is desirable that in this finite graph, random walks quickly "forget" where they started (in Markov Chain parlance, this corresponds to a fast mixing rate). It is known that the complete graph on s vertices attains the minimum mean mixing time among all graphs on s vertices [3]. The extreme case — a complete graph with self-loops on 2 nodes (Figure 5.8) — is instructive: here, the stationary distribution $(0.5, 0.5)$ can be approached to within 1% in as few as 10 steps on the chain. This graph corresponds to a top-down tree, where node values are only assigned win/loss values, and any mistake by a player leads to a winning po-

sition for his opponent. Random playouts are practically memoryless in even *short* games in this family, and MM-1-1000R performs no better than random guessing. We argue this mathematically, below. The quantities l, r, d, and $S_d(v)$ are all defined as in Proposition 1, but are used here in the context of top-down win/loss trees.

Proposition 2. $S_d(l) - S_d(r) = 1$ *for all* $d \geq 0$, *for win/loss trees.*

Proof. We proceed by induction on d. For the base case, $S_0(l) - S_0(r) = 1 - 0 = 1$ by definition. Assume the claim holds for $d = t, t \geq 0$. If t is a Max level, then every node with Minimax value 1 at level t spawns one child each with values 1 and 0, while the children of a node with Minimax value 0 both have value 0. Thus, $S_{t+1}(l) - S_{t+1}(r) = S_t(l) - S_t(r) = 1$. If t is a Min level, then each node with value 0 at level t spawns one child each with values 0 and 1, while the children of a node with Minimax value 1 both have value 1. Thus, $S_{t+1}(l) - S_{t+1}(r) = (2 \cdot S_{t+1}(l) + 2^t - S_{t+1}(l)) - (2 \cdot S_{t+1}(r) + 2^t - S_{t+1}(r)) = S_{t+1}(l) - S_{t+1}(r) = 1$. □

As before, we denote the average outcome of playouts from v to be $P(v)$. If the subtree rooted at v has uniform depth d, then similar to Proposition 2, $\mathbb{E}[P(l)] - \mathbb{E}[P(r)] = (S_d(l) - S_d(r))/2^d = 1/2^d$. Since $1/2^d \to 0$ for increasing d, we conclude that for games of non-trivial duration, playout samples become increasingly ineffective at predicting the optimal move.

Of course, there is a choice to be made about where we "cap" our Markov Chain. We use these insights to propose a new synthetic tree model in the following section.

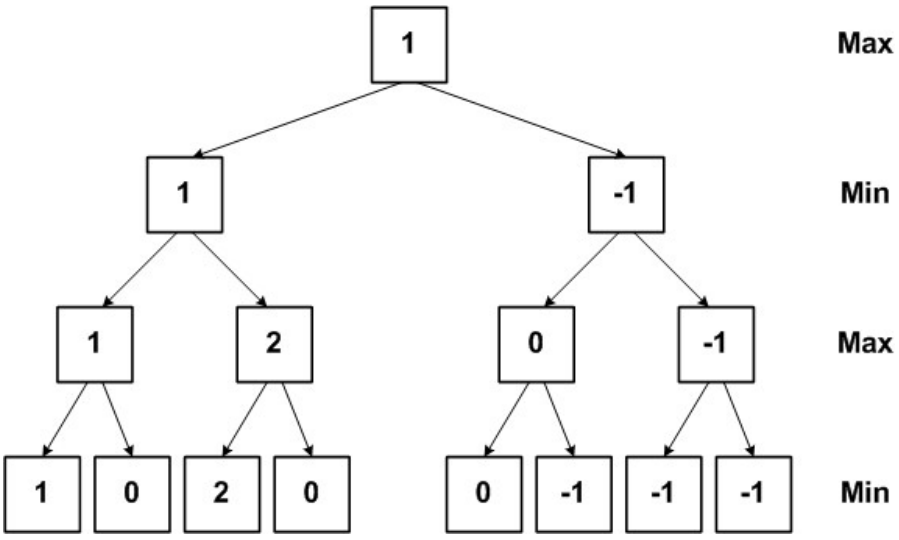

Figure 5.9: A sample BPV tree of depth 3, branching factor 2, Minimax values capped in the range $\{-1, \ldots, 2\}$ and where sub-optimal moves are assessed a penalty drawn from the set $\{1, 2\}$.

5.3.2 A New Model

A *Bounded Prefix Value (BPV)* tree is a PV tree with two modifications:

- An additional parameter v_{max}, which constrains the Minimax values of the nodes to stay within the range $\{-v_{max} + 1, -v_{max} + 2, \ldots, 0, \ldots, v_{max} - 1, v_{max}\}$.
- The sub-optimal penalty value k is drawn uniformly at random from the set $\{1, \ldots, k_{max}\}$, rather than fixing it to a constant value.

Figure 5.9 shows a sample BPV tree with $v_{max} = 2$ and $k_{max} = 2$.

Heuristic model

In most real-world games, a complete search to the horizon of the game is infeasible. The typical workaround is to set a depth or time cut-off and to evaluate

the leaf nodes at the frontier of the search using a heuristic function. Following the approach of Luštrek et al. [62], we generate heuristic values for nodes by adding Gaussian noise to their true Minimax values. In particular, the heuristic value $h(v)$ of a node v is defined as follows:

$$h(v) = m(v) + X$$

Here, $X \sim \mathcal{N}(0, \sigma)$, i.e., X is a Gaussian random variable with mean 0 and standard deviation σ, which is a parameter of the model. The noise sampled to compute the heuristic value of a node is independent of the noise sampled at other nodes. Moreover, once a node is assigned a particular heuristic value, it retains that value for the duration of the game — this ensures that repeated visits to a given node yield consistent heuristic estimates.

Balancing v_{max} and k_{max}

In Section 5.3.1, we saw how setting $v_{max} = \infty$ yielded unreasonably good performance from MM-1-1000R. We subsequently saw that when v_{max} is small and k_{max} proportionally large (i.e., large enough to guarantee a complete transition graph), we obtain a fast mixing Markov Chain and MM-1-1000R performs no better than random guessing. We now investigate the impact of these parameter settings on the behavior of Minimax search. In particular, we want to avoid parameter settings that elicit pathological behavior from Minimax. In what follows, we fix the branching factor b at 4 and set the maximum tree depth $d_{max} = 100$.

First, we define the pathology index *PI* of a given family of BPV trees as

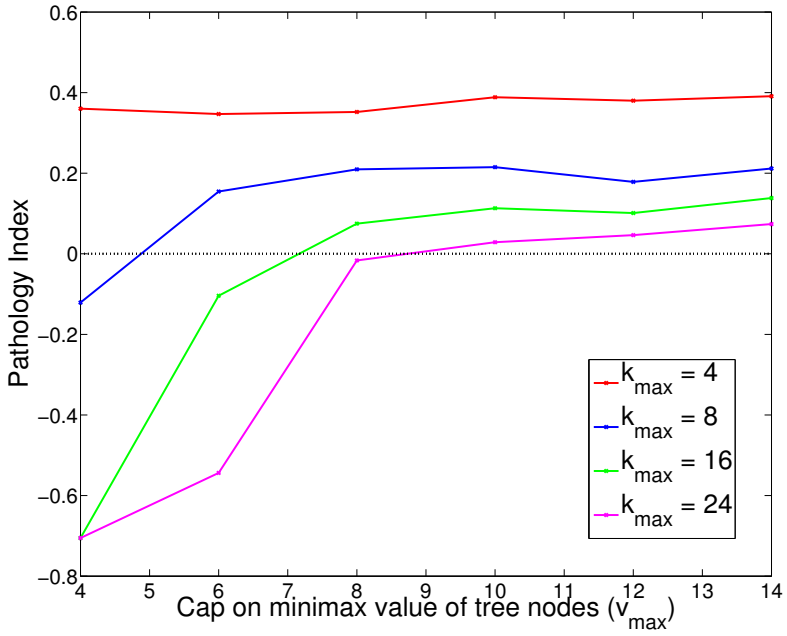

Figure 5.10: A plot of the pathology index as a function of v_{max}, for various values of k_{max}.

follows:

$$PI = 1 - \frac{decAcc_2}{decAcc_6}$$

$decAcc_i$ represents the decision accuracy of Minimax search to depth i on the given BPV tree family. Thus, $PI < 0$ indicates the presence of lookahead pathology, while $PI > 0$ indicates that search is beneficial. In figure 5.10, we plot PI against v_{max} for various values of k_{max}. Each data point is the result of simulations on 1000 trees generated as per the BPV model ($\sigma = 4$). We see that pathology occurs whenever k_{max} is large in relation to v_{max}. This makes sense: in these families of trees, since the Minimax values of moves "jump" around a lot, long-term correlations between parent-child values are rapidly lost, resulting in the return of pathology. However, in figure 5.11 which illustrates how the decision accuracy of MM-1-1000R varies with v_{max} for difference value of k_{max}, we see that MM-1-1000R performs poorly exactly in these circumstances, where node correlations

are small. Thus, large regions of the parameter space of the BPV model yield unrealistic game trees. In the rest of this chapter, we fix $v_{max} = 10$ and $k_{max} = 8$ to strike a balance between these extreme outcomes and produce realistic games (though our results hold for other balanced settings of v_{max} and k_{max} as well).

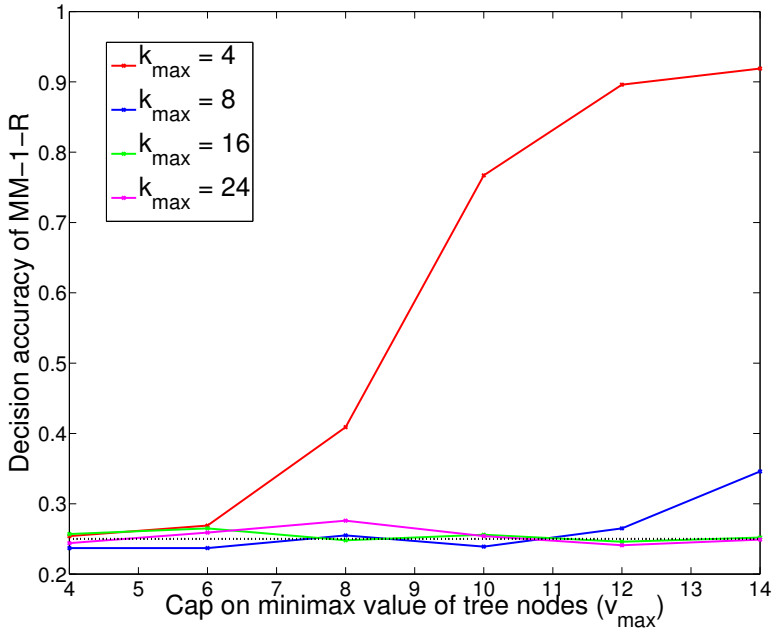

Figure 5.11: A plot of the decision accuracy of MM-1-R as a function of v_{max}, for various values of k_{max}

5.3.3 Comparing Minimax and UCT

Having established the range of parameters for which the BPV model produces "reasonable" trees, we now focus our attention on studying how Minimax and UCT perform on this family of trees. In particular, we are interested in discovering if there is a simple "knob" in our model which can be turned to make Minimax outperform UCT and vice versa. To keep the comparisons fair in these experiments, we allow UCT to expand at most the same number of nodes as

the Minimax search it is competing against. Moreover, both algorithms use the heuristic for evaluating leaf nodes. We fix $b = 4$ and $d_{max} = 100$ in these experiments, though the results shown here carry over to other reasonable choices for b and d_{max} as well.

Domination

In practice, UCT's performance is heavily dependent on the value of the exploration bias term c that is used, as we saw in Section 3.2.3. Typically, c is tuned via careful experimentation, to find the single value that leads to the best overall performance. However, the structure of real games seldom shows any uniformity: many games have distinct opening, mid-game and end-game phases, with very different characteristics. This inability to adapt its exploration/exploitation scheme to the local search space is one of the key reasons why UCT suffers in many games (indeed, this is also a promising avenue for future research). Thus, rather than comparing Minimax to an optimally tuned UCT search, we focus our attention on whether UCT *dominates* Minimax as defined below.

Definition 3. *We say that UCT* ***dominates*** *Minimax with respect to a BPV parameter vector* Φ *if there is a single constant* c^* *such that UCT with* $c = c^*$ *achieves a higher decision accuracy than Minimax for any assignment to* Φ*. For convenience, we will refer to* Φ *as a* ***knob****.*

This definition captures the intuition that if UCT scores higher than Minimax at some fixed setting of c, *independent* of the tree parameters being adjusted, then it is outperforming Minimax in a very robust sense. On the other hand, if UCT's

performance at a fixed c setting can be made arbitrarily worse by adjusting some tree parameters, then it is likely to fare poorly in a game where those parameters fluctuate locally. The remainder of this section is dedicated to introducing and experimenting with a variety of knobs in the BPV model.

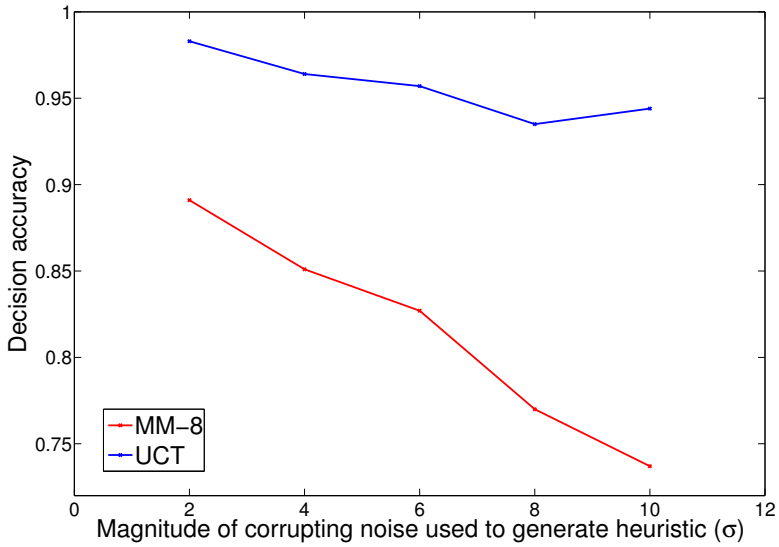

Figure 5.12: Decision accuracies of MM-8 and UCT, with σ as the controlling knob.

The Effect of σ

The first knob we consider is the single parameter σ, the magnitude of the corrupting noise that is used to generate the heuristic estimates of nodes. Figure 5.12 plots the decision accuracies of an MM-8 search (i.e., a Minimax search to 8-ply depth) and a UCT search ($c = 0.2$) as a function of σ. Firstly, we observe that UCT does indeed dominate Minimax with respect to σ; in other words, UCT is an unarguably superior planning algorithm for this family of trees. Secondly, we note that the performance of both algorithms deteriorates as we increase the noise in the heuristic estimates — however, UCT seems far more ro-

bust to the increased noise. This is a result of the averaging backups that are performed by UCT. As more samples accumulate at the nodes near the top of the UCT search tree, the i.i.d. noise we introduce via the heuristic undergoes perfect cancellation, and UCT's utility estimates near the root node converge quickly.

5.3.4 Augmenting BPV trees

In this section, we extend the basic BPV tree model with additional parameters that allows us to generate a richer class of trees. This also equips us with new knobs with which to study the UCT-Minimax trade-off.

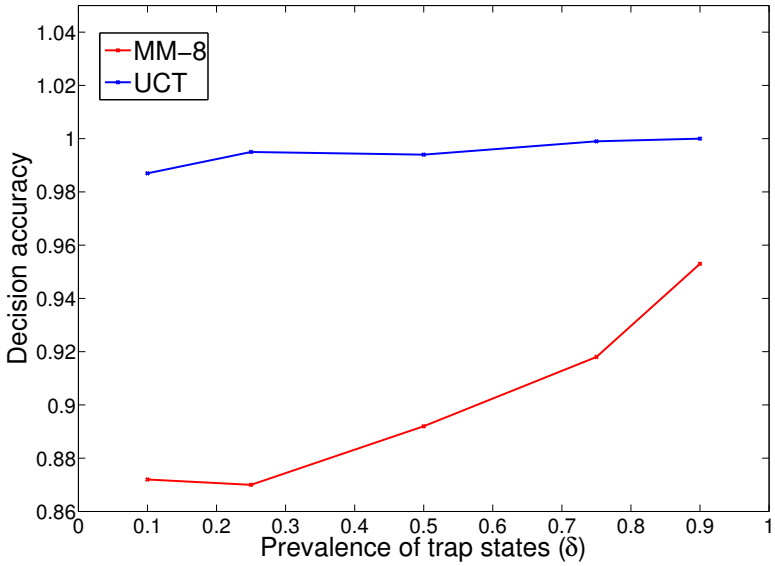

Figure 5.13: Decision accuracies of MM-8 and UCT, with δ as the controlling knob.

Search traps

One obvious augmentation to the BPV tree model is a mechanism for adding search traps. The presence of search traps was demonstrated to affect the performance of UCT in games like Chess and Mancala in Chapter 4. We follow the approach of Pearl [77], and introduce a parameter δ which represents the probability that a given node might be marked as a terminal (in particular, as a loss for the previous player). Figure 5.13 plots the decision accuracies of UCT and MM-8, as a function of δ (once again with $c = 0.2$). Surprisingly, we discover that traps states do *not* hurt the performance of UCT. Indeed, these results hold even when we use more sophisticated schemes — such as planting level-3 traps with probability δ only when there is a transition in the Minimax value across the win/loss threshold — to implant trap states. This suggests that search traps planted in such a random fashion may not be be able to affect the performance of UCT in the same manner as seen in real games, and that more intricate design may be necessary.

Systematic heuristic biases

The second modification to the basic BPV model we consider is what we term the *dispersion lag* of the heuristic. In many tactical games, a major component of the heuristic function is a material count; however, in many game situations, the material count may not change much from move to move, while the underlying strategic advantage shifts dramatically (the Chess end-game is a good example). Thus, deceiving feedback from the heuristic may persist for several plys in the tree, before getting corrected. We model this phenomenon by introducing two new parameters: *bias* and *lag*. This works as follows: for the first

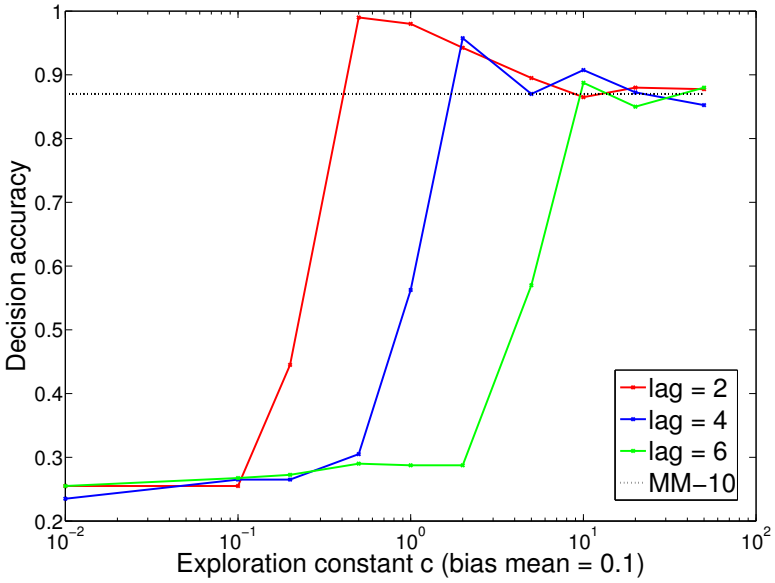

Figure 5.14: Decision accuracies of UCT and MM-10, with c on a logarithmic x-axis, with the controlling knob being the tuple $< bias, lag >$.

lag plys from the root node, we decouple the heuristic estimate entirely from the underlying true Minimax values of the nodes; the heuristic values are instead sampled uniformly at random from the interval $[bias - 0.1, bias + 0.1]$. Thereafter, we revert to generating heuristic estimates for nodes as per our original scheme (see section 5.3.2). Figures 5.14 and 5.15 plot the decision accuracy of UCT for various settings of the parameter c, while varying the knob $< lag, bias >$. We also record the decision accuracy of MM-10, which is unaffected by this specific knob and is therefore just a single line. We notice something intriguing: one may intuitively expect that the *bias* parameter should have no bearing on the performance of UCT, since the information content of a purely random heuristic is 0. However, this is clearly not the case: a bias of 0.5 allows UCT to dominate MM-10, whereas a bias of 0.1 does not. Indeed, interpolating the trend visible in figure 5.14, it seems that UCT needs to build more complete trees to stay com-

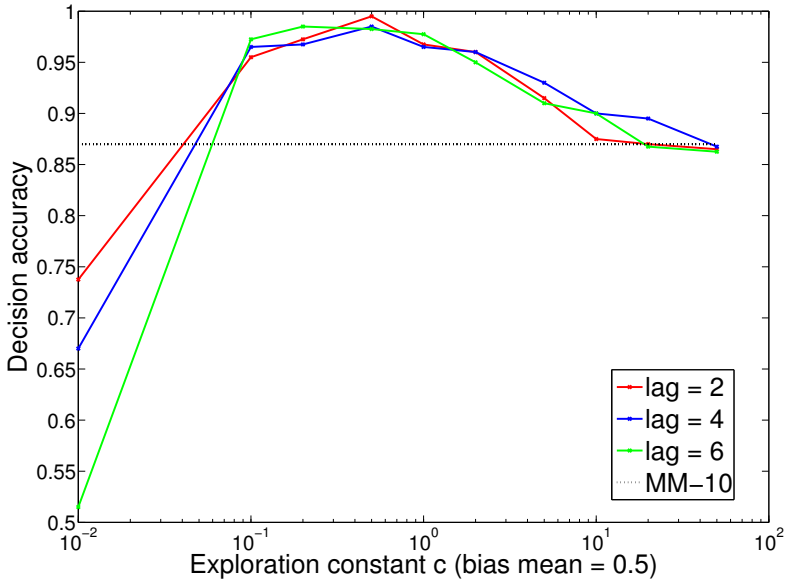

Figure 5.15: Decision accuracies of UCT and MM-10, with c on a logarithmic x-axis, with the controlling knob being the tuple $< bias, lag >$.

petitive with MM-10 as we increase *lag* — in other words, this defines a family of trees where Minimax is the superior algorithm. A pertinent question at this point is why this happens. For this, we need to look at what the dispersion lag implies for the estimates UCT computes for nodes close to the root. In the case where $bias = 0.1$, after b^{lag} iterations, the children of the root node all have the same expected utility and each has been visited b^{lag-1} times. However, the next UCT iteration is going to return a sample from a much wider range — and since our current estimates for the children of the root node are all so low, with high likelihood, this new sample will be significantly greater than 0.1. Indeed, so will most subsequent samples. Thus, immediately, UCT begins to exploit this sliver of information and "freezes" onto this variation. In the unlucky event that this is in fact the incorrect move at the root, UCT will end up overcommitting to the wrong decision. In the case where the $bias = 0.5$, the sample outcome from iter-

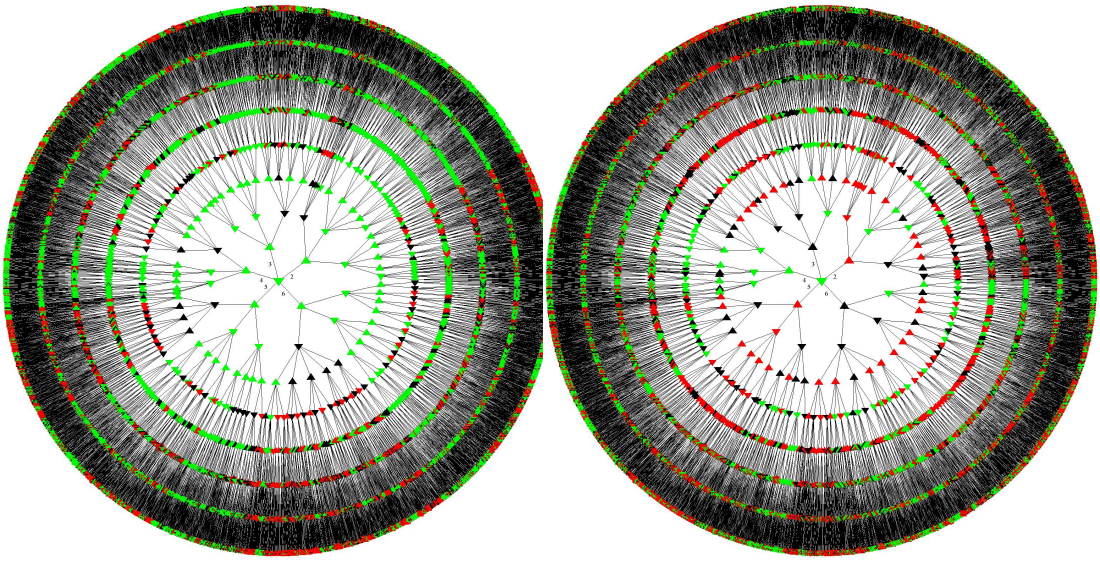

Figure 5.16: State at risk in Mancala where UCT blunders and picks the trap move instead of the optimal one. The shading in the left panel is based on static heuristic evaluation of states, while in the right panel, it based on the true Minimax value of the positions.

ation $b^{lag} + 1$ has just as likely a chance of being higher than 0.5 as lower. Indeed, this is true for all subsequent samples as well, with the result that UCT never overcommits to the wrong variation. Indeed, this observation neatly mirrors our findings in Section 5.2.2.

A pertinent question at this point is: *is such dispersion lag observed in practice?* The answer is in the affirmative — in Figure 5.16, we present a visualization of a *complete* expansion of the game tree of a Mancala position, up to 8 plys. Nodes are shaded one of three colors based on their evaluation: red corresponds to a win for Max, green corresponds to a win for Min, and black corresponds to drawn positions. While the shading in the left panel is based upon *static* evaluations of the states (i.e., a win-loss-draw discretization of the heuristic estimate), the right panel uses the true Minimax values of the nodes to shade them. We

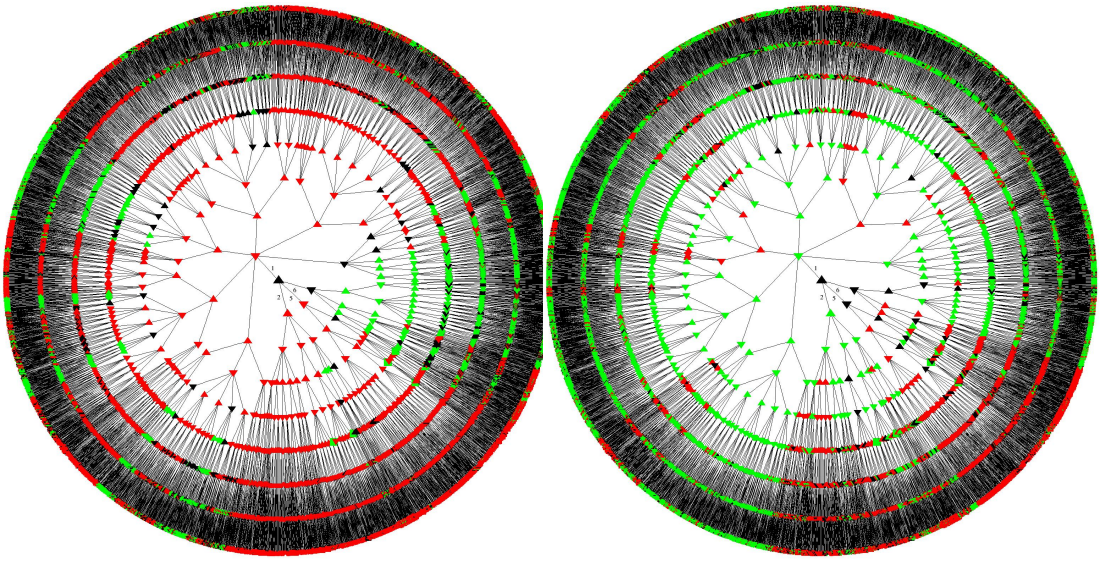

Figure 5.17: Another state at risk in Mancala where UCT makes a mistake

make two observations: first, the *classification* accuracy of the heuristic, i.e. its ability to correctly predict a win as a win, a loss and a loss and a draw as a draw, is quite low. This is visually apparent from comparing the contrasting shading of the two trees in Figure 5.16. We provide a second example in Figure 5.17 that tells a similar story. Secondly, the figures also provide evidence of the occurrence of dispersion lag; in both, we see erroneous evaluations that propagate fairly deep down subtrees. This can be observed as near-uniformly shaded portions of the trees in the left panel of both figures, which disagree with the shading of the right tree — evidence of a material misevaluation that is not easily corrected without the state of the game undergoing a significant evolution through additional play. Both these board positions are, in fact, states at risk: UCT incorrectly picks the trap move in both instances, when there are other options that do not lead to a premature loss.

5.4 Discussion

The main thrust of this chapter was to characterize aspects of adversarial search domains that make them more or less favorable to UCT-style search algorithms. Our focus was on synthetic games, that allow analysis and principled experimentation to gain deeper insights than is possible by studying real-world games like Chess and Go. We introduced two novel synthetic game settings and examined the performance of Minimax and UCT in them. The first family of game trees, named trees with implanted critical levels, captures a key feature that is present in many tactical games such as Chess — namely, the presence of a few critical moves that need to be played exactly right, to guarantee victory. In this setting, we analyzed the convergence properties of UCT, which was shown to be additively decomposable into time spent between critical levels. Later, we presented a calculation that demonstrated a "freezing" phenomenon in UCT, and explained how misleading heuristic feedback could severely impair the performance of UCT.

Next, we introduced the Bounded Prefix Value (BPV) tree model, which corrects many of the shortcomings of other models studied in the literature. In particular, we showed that the Prefix Value (PV) tree model has the undesirable property of being solved by the naïve MM-1-kR algorithm, for sufficiently large values of k. We presented our extension of the PV tree model, namely the BPV model, which while retaining the richness of the former also generates trees that are more faithful to real-world games. Finally, we introduced the notion of domination for comparing the relative strengths of UCT and Minimax. We were able to produce a relatively clean characterization of the family of trees for which UCT is superior to Minimax, and vice versa — in particular, we showed

that for a vast class of trees with a simple additive Gaussian noise heuristic model, UCT handsomely outperforms Minimax. Heuristic dispersion lag, on the other hand, can single-handedly wreck UCT by leading it into blind alleys from which recovery is difficult.

CHAPTER 6

CONCLUSIONS

This thesis was motivated by the puzzling complementarity of today's two most popular approaches to planning in adversarial domains. Depth-limited complete search techniques based on the Minimax algorithm were the tool of choice for tackling the problem of decision making in hard combinatorial domains like Chess and Checkers for many decades. However, attempts to adapt these approaches to certain other domains — most notably Go — were unsuccessful. In the last few years, planners based on the Monte-Carlo Tree Search framework such as UCT have met with great success in Go and several other domains, where previous Minimax-style approaches had failed. Indeed, there is now a real possibility that computer programs will exceed human-level performance in Go within the next decade — a proposition that would have been unthinkable five years ago. However, this success of UCT-style search procedures has not been transferred back to domains where Minimax approaches have traditionally done well. This thesis presented a body of work that offers explanations for this performance mismatch.

In Chapter 3, we carried out a rigorous comparison of UCT and Minimax in the domain of Chess. By stripping down both algorithms to their "purest" form, we experimentally confirmed what had been heretofore informal and anecdotal knowledge — that UCT can be a powerful knowledge-free planning method, that it indeed does not compare well with Chess when some knowledge (in the form of heuristics) is made available and that random playouts, surprisingly, yield useful information even in games with highly unstable dynamics like Chess. We followed this up with a dissection of UCT in the game of Man-

cala. The choice of this domain was motivated by the fact that *both* Minimax and UCT perform well here "out-of-the-box". This allows for comparative experiments between the two that are not possible in other domains such as Chess and Go. We concluded this chapter by demonstrating a number of trade-offs in UCT — how the setting of the exploration bias parameter c indeed produces very differently shaped trees, how Minimax backups can offer improvements over averaging backups when good heuristic guidance is available, and how it is more prudent to spend time building larger trees, rather than on more simulations.

In Chapter 4, we identified one key aspect in which the domain of Chess differs from that of Go and Mancala. In the former, shallow search traps occur throughout the search space. Thus, the planning agent is constantly on thin ice, attempting to avoid blunders that place it in positions from where its opponent has a short, guaranteed winning strategy. In Mancala, like Go, traps only appear in the latter stages of the game. We empirically demonstrated how the prevalence of traps states can impact the performance of UCT, by presenting cases where UCT picked (provably) sub-optimal moves, despite being given the same computational resources as an equivalent Minimax search. We further strengthened this claim by demonstrating how a "hybrid" planning strategy, that switches from UCT to Minimax for guidance based on the estimated prevalence of trap states in the search neighborhood, outperformed either strategy by itself.

Finally, in Chapter 5, we complemented our investigation of UCT and Minimax in real games, with studies in synthetic game trees. First, we presented games with implanted critical levels, that mimic the kind of situation that often

arises in tactical games like Chess. In this setting, we were able to show that the convergence of time of UCT could be additively decomposed into times spent between critical levels. Under some simplifying assumptions, we also showed how unlucky sampling outcomes could cause UCT to "freeze" and waste time exploring sub-optimal lines of play. Our second game tree model, dubbed the Bounded Prefix Value (BPV) model, was motivated by the fact that existing synthetic games tend to exhibit undesirable or unrealistic properties, such as the lookahead pathology or susceptibility to naïve sampling. By drawing connections to Markov Chain theory, we were able to show that the games generated by our model are not trivially solved using simple sampling-based search, and nor do they suffer from the lookahead pathology. We showed that under the assumption of independent Gaussian heuristic noise, UCT was a very robust planning algorithm. We then introduced the notion of value dispersion lag, and showed how systematic biases in heuristics create problems for UCT-style planning approaches.

This work opens up several interesting research avenues worthy of further investigation. The results highlighting how search traps and heuristic biases entrap UCT are an important first step in recognizing the factors that lead UCT to fail in practice. The next step would entail applying these insights to develop new algorithms that are more robust to such effects — these could potentially be of great use in designing agents for the AAAI General Game Playing contest [36]. The hybrid approach presented in Section 4.4 is one such attempt, but other, more attractive alternatives may exist. In particular, one could consider alternative bandit strategies that take into account the fact that we are operating on a limited computational budget [63]. Though our initial experiments with this approach have not been successful, more investigation along these lines

is needed. Another interesting line of research involves applying UCT-style approaches to other difficult combinatorial reasoning and optimization problems. There has been some success, for example, in using UCT for satisfiability testing [78] and in guiding Mixed-Integer Programming (MIP) solvers [86]. However, these domains have been the subject of intense research for many decades and it will be hard to make UCT competitive with these highly optimized solvers. Problems like Maximum Satisfiability (MaxSAT) and finding satisfying assignments to Quantified Boolean Formulas (QBF), where current state-of-the-art techniques do not scale well, are perhaps far more ripe for attacking with sampling-based search methods. The possibility that UCT could do to these domains what it has done for Go remains a tantalizing scenario.

BIBLIOGRAPHY

[1] Bruce Abramson. Control strategies for two-player games. *ACM Comput. Surv.*, 21(2):137–161, June 1989.

[2] Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(2):182–193, February 1990.

[3] David Aldous and Jim Fill. Reversible markov chains and random walks on graphs. Manuscript available at `http://www.stat.berkeley.edu/~aldous/RWG/book.html`, 1999.

[4] Victor L. Allis. A knowledge-based approach to connect-four. The game is solved: White wins. Master's thesis, Vrije Universiteit, October 1988.

[5] Victor L. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, 1994.

[6] Thomas Anantharaman, Murray S. Campbell, and Feng hsiung Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99 – 109, 1990.

[7] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo tree search in hex. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):251–258, December 2010.

[8] L. Atkin and D. Slate. Chess 4.5-the northwestern university chess program. In David Levy, editor, *Computer chess compendium*, pages 80–103. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[9] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[10] Radha-Krishna Balla and Alan Fern. Uct for tactical assault planning in real-time strategy games. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 40–45, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[11] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Knightcap: A chess programm that learns by combining td(lambda) with game-tree search. In *Proceedings of the Fifteenth International Conference on Machine Learning*,

ICML '98, pages 28–36, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[12] D.F. Beal. Experiments with the null move. In *Advances in Computer Chess 5*, pages 65–79. Elsevier, 1989.

[13] A. G. Bell. Kalah on atlas. *Machine Intelligence*, 3:181–194, 1968.

[14] Hans J. Berliner. Some necessary conditions for a master chess program. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 77–85, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[15] Hans J. Berliner. The B^* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1):23 – 40, 1979.

[16] Hans J. Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, 14(2):205 – 220, 1980.

[17] Alan W. Biermann. Theoretical issues related to computer game playing programs. *Personal Computing*, pages 86–88, 1978.

[18] Darse Billings. *Algorithms and assessment in computer poker*. PhD thesis, University of Alberta, Edmonton, Alta., Canada, 2006. AAINR22991.

[19] Bruno Bouzy and Tristan Cazenave. Computer go: An ai oriented survey. *Artificial Intelligence*, 132(1):39 – 103, 2001.

[20] Bernd Brügmann. Monte carlo go. `http://www.ideanest.com/vegos/MonteCarloGo.pdf`, 1993.

[21] Michael Buro. The evolution of strong othello programs. In Ryohei Nakatsu and Jun'ichi Hoshino, editors, *IWEC*, volume 240 of *IFIP Conference Proceedings*, pages 81–88. Kluwer, 2002.

[22] Michael Buro. Improving heuristic mini-max search by supervised learning. *Artif. Intell.*, 134(1-2):85–99, 2002.

[23] Michael Buro, Jeffrey R. Long, Timothy Furtak, and Nathan Sturtevant. Improving state evaluation, inference, and search in trick-based card

games. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 1407–1413, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[24] Murray Campbell, A. Joseph Hoane, Jr., and Feng-hsiung Hsu. Deep Blue. *Artif. Intell.*, 134:57–83, January 2002.

[25] Tristan Cazenave and Nicolas Jouandeau. A parallel monte-carlo tree search algorithm. In van den Herik et al. [110], pages 72–80.

[26] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In van den Herik et al. [110], pages 60–71.

[27] Guillaume M. J-B. Chaslot, Mark H. M. Winands, H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4:343–357, 2008.

[28] Jonathan Chevelu, Thomas Lavergne, Yves Lepage, and Thierry Moudenc. Introduction of a new paraphrase generation tool based on monte-carlo sampling. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, ACLShort '09, pages 249–252, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[29] Paolo Ciancarini and Gian Piero Favini. Monte carlo tree search in kriegspiel. *Artif. Intell.*, 174(11):670–684, 2010.

[30] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *CoRR*, abs/cs/0703062, 2007.

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

[32] Rémi Coulom. Computing Elo ratings of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, December 2007.

[33] Chessgames Online Chess Database and Community. Alan turing vs alick glennie. `http://www.chessgames.com/perl/chessgame?gid=1356927`, 1952.

[34] Chrilly Donninger. Null move and deep search: Selective-search heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143, 1993.

[35] Roger Eckhardt. Stan Ulam, John von Neumann, and the Monte Carlo method. *Los Alamos Science, Special Issue*, (15):131–137, 1987.

[36] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, AAAI'08, pages 259–264. AAAI Press, 2008.

[37] David B. Fogel, Timothy J. Hays, Sarah L. Hahn, and James Quon. A self-learning evolutionary chess program. In *Proceedings of the IEEE*, pages 1947–1954, 2004.

[38] B. Franklin. *The Morals of Chess*. W. Bent, 1787.

[39] Timothy Furtak and Michael Buro. Minimum proof graphs and fastest-cut-first search heuristics. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 492–498, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[40] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000.

[41] Romaric Gaudel and Michèle Sebag. Feature selection as a one-player game. In Johannes Fürnkranz and Thorsten Joachims, editors, *ICML*, pages 359–366. Omnipress, 2010.

[42] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pages 273–280, New York, NY, USA, 2007. ACM.

[43] Sylvain Gelly and David Silver. Achieving master level play in 9 x 9 computer go. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 1537–1540. AAAI Press, 2008.

[44] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.*, 175(11):1856–1875, 2011.

[45] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for

Monte-Carlo Go. In *NIPS-06 Workshop on Online Trading between Exploration and Exploitation*, Whistler, BC, December 2006.

[46] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France, November 2006.

[47] Chris Gifford, James Bley, Dayo Ajayi, and Zach Thompson. Searching and game playing: An artificial intelligence approach to Mancala. Technical Report ITTC-FY2009-TR-03050-03, Information Telecommunication and Technology Center, University of Kansas, Lawrence, KS, July 2008.

[48] James J. Gillogly. The technology chess program. *Artif. Intell.*, 3(1-3):145–163, 1972.

[49] Matthew L. Ginsberg. Gib: Imperfect information in a computationally challenging game. *J. Artif. Intell. Res. (JAIR)*, 14:303–358, 2001.

[50] Peter Grünwald and Peter Spirtes, editors. *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*. AUAI Press, 2010.

[51] Geoffrey Irving, Jeroen Donkers, and Jos Uiterwijk. Solving Kalah. *ICGA Journal*, 23(3):139–148, 2000.

[52] Michael B. Johanson. Robust Strategies and Counter-Strategies: Building a Champion Level Computer Poker Player. Master's thesis, University of Alberta, 2007.

[53] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Mach. Learn.*, 49(2-3):193–208, November 2002.

[54] Thomas Keller and Patrick Eyerich. Prost: Probabilistic planning based on uct. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *ICAPS*. AAAI, 2012.

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

[56] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo plan-

ning. In *Proceedings of the 17th European conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[57] Kai-Fu Lee and Sanjoy Mahajan. The development of a world class othello program. *Artificial Intelligence*, 43(1):21 – 36, 1990.

[58] David Levy, David Broughton, and Mark Taylor. The SEX algorithm in computer chess. *ICCA Journal*, 12(1):10–21, 1989.

[59] Jeffrey Richard Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[60] Richard J. Lorentz. Amazons discover Monte-Carlo. In *Computers and Games*, pages 13–24, 2008.

[61] Mitja Luštrek, Matjaž Gams, and Ivan Bratko. Why minimax works: an alternative explanation. In *Proceedings of the 19th international joint conference on Artificial intelligence*, IJCAI'05, pages 212–217, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[62] Mitja Luštrek, Matjaž Gams, and Ivan Bratko. Is real-valued minimax pathological? *Artificial Intelligence*, 170(6-7):620–642, 2006.

[63] Omid Madani, Daniel J. Lizotte, and Russell Greiner. The budgeted multi-armed bandit problem. In John Shawe-Taylor and Yoram Singer, editors, *COLT*, volume 3120 of *Lecture Notes in Computer Science*, pages 643–645. Springer, 2004.

[64] T. A. Marsland. Computer chess and search. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley, 1991.

[65] T. A. Marsland and Yngvi Björnsson. From minimax to manhattan. In *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*, pages 31–36. AAAI Press, 1997.

[66] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Comput. Surv.*, 14(4):533–551, December 1982.

[67] David A. McAllester. Conspiracy numbers for min-max search. *Artif. Intell.*, 35(3):287–310, July 1988.

[68] David A. McAllester and Deniz Yuret. Alpha-beta-conspiracy search. *ICGA Journal*, 25(1):16–35, 2002.

[69] John W. Milnor. Games against nature. Technical Report RM-69-PR, Rand Corporation, 1951.

[70] Martin Müller. Computer go. *Artif. Intell.*, 134(1-2):145–179, 2002.

[71] John Nash. Non-Cooperative Games. *The Annals of Mathematics*, 54(2):286–295, September 1951.

[72] Dana S. Nau. An investigation of the causes of pathology in games. *Artif. Intell.*, 19:257–278, 1982.

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

[74] Allen Newell, J. C. Shaw, and Herbert A. Simon. Chess-playing programs and the problem of complexity. In *Computers & Thought*, pages 39–70. MIT Press, Cambridge, MA, USA, 1995.

[75] BBC News Online. Garry Kasparov plays 1950 chess program by computing founder Alan Turing. `http://www.bbc.co.uk/news/uk-england-manchester-18584312`, June 2012.

[76] Judea Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14(2):113 – 138, 1980.

[77] Judea Pearl. On the nature of pathology in game searching. *Artif. Intell.*, 20(4):427–453, 1983.

[78] Alessandro Previti, Raghuram Ramanujan, Marco Schaerf, and Bart Selman. Applying UCT to boolean satisfiability. In Karem A. Sakallah and Laurent Simon, editors, *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 373–374. Springer, 2011.

[79] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. On adversarial search spaces and sampling-based planning. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *ICAPS*, pages 242–245. AAAI, 2010.

[80] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. Understanding sampling style adversarial search methods. In Grünwald and Spirtes [50], pages 474–483.

[81] Raghuram Ramanujan and Bart Selman. Trade-offs in sampling-based adversarial planning. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *ICAPS*. AAAI, 2011.

[82] Alexander Reinefeld. An Improvement to the Scout Tree-Search Algorithm. *International Computer Chess Association Journal*, 6(4):4–14, December 1983.

[83] D.J. Richards and T.P. Hart. The alpha-beta heuristic. Technical Report AIM-030, Massachussetts Institute of Technology, December 1961.

[84] Igor Roizen and Judea Pearl. A minimax algorithm better than alpha-beta? yes and no. *Artif. Intell.*, 21(1-2):199–220, March 1983.

[85] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[86] Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with uct. In Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson, editors, *CPAIOR*, volume 7298 of *Lecture Notes in Computer Science*, pages 356–361. Springer, 2012.

[87] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959.

[88] Jonathan Schaeffer. *Experiments in search and knowledge*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1986.

[89] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(11):1203–1212, November 1989.

[90] Jonathan Schaeffer. Computer (and human) perfection at checkers. In *Proceedings of the 22nd Canadian Conference on Artificial Intelligence: Advances in Artificial Intelligence*, Canadian AI '09, pages 3–3, Berlin, Heidelberg, 2009. Springer-Verlag.

[91] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[92] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artif. Intell.*, 53:273–289, February 1992.

[93] Anton Scheucher and Hermann Kaindl. Benefits of using multivalued functions for minimaxing. *Artif. Intell.*, 99(2):187 – 208, 1998.

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

[95] Brian Sheppard. World-championship-caliber Scrabble. *Artif. Intell.*, 134(1-2):241–275, 2002.

[96] David Silver and Gerald Tesauro. Monte-carlo simulation balancing. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman, editors, *ICML*, volume 382 of *ACM International Conference Proceeding Series*, page 119. ACM, 2009.

[97] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta, editors, *NIPS*, pages 2164–2172. Curran Associates, Inc., 2010.

[98] James R. Slagle and John K. Dixon. Experiments with the m & n tree-searching program. *Commun. ACM*, 13(3):147–154, March 1970.

[99] Stephen J. J. Smith, Dana S. Nau, and Thomas A. Throop. Total-order multi-agent task-network planning for contract bridge. In William J. Clancey and Daniel S. Weld, editors, *AAAI/IAAI, Vol. 1*, pages 108–113. AAAI Press / The MIT Press, 1996.

[100] George C. Stockman. A Minimax Algorithm Better than Alpha-Beta? *Artificial Intelligence*, 12(2):179–196, August 1979.

[101] Nathan R. Sturtevant. An analysis of uct in multi-player games. In van den Herik et al. [110], pages 37–49.

[102] Nathan R. Sturtevant. An analysis of uct in multi-player games. In van den Herik et al. [110], pages 37–49.

[103] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3(1):9–44, August 1988.

[104] Gerald Tesauro. Practical issues in temporal difference learning. *Mach. Learn.*, 8(3-4):257–277, May 1992.

[105] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[106] Gerald Tesauro, V. T. Rajan, and Richard Segal. Bayesian inference in monte-carlo tree search. In Grünwald and Spirtes [50], pages 580–588.

[107] John Tromp. Connect-4 data set. `http://archive.ics.uci.edu/ml/datasets/Connect-4`, 1995.

[108] T. R. Truscott. Techniques used in minimax game-playing programs. Master's thesis, Duke University, Durham, NC, 1981.

[109] Alan M. Turing. Digital computers applied to games. In B. V. Bowden, editor, *Faster Than Thought*, pages 286–295. Pitman Publishing, 1953.

[110] H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors. *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*, volume 5131 of *Lecture Notes in Computer Science*. Springer, 2008.

[111] H.Jaap van den Herik, Jos W.H.M. Uiterwijk, and Jack van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(12):277 – 311, 2002.

[112] Joel Veness, David Silver, William T. B. Uther, and Alan Blair. Bootstrapping from game tree search. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *NIPS*, pages 1937–1945. Curran Associates, Inc., 2009.

[113] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928. 10.1007/BF01448847.

[114] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[115] Mark H. M. Winands and Yngvi Björnsson. Evaluation function based Monte-Carlo LOA. In *ACG*, pages 33–44, 2009.

[116] Albert L. Zobrist. A new hashing method with application for game playing. Technical Report TR-88, Department of Computer Sciences, University of Wisconsin-Madison, April 1970.