A Survey of Monte Carlo Tree Search Methods

Cameron Browne, *Member, IEEE,* Edward Powley, *Member, IEEE,* Daniel Whitehouse, *Member, IEEE,* Simon Lucas, *Senior Member, IEEE,* Peter I. Cowling, *Member, IEEE,* Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton

Abstract—Monte Carlo Tree Search (MCTS) is a recently proposed search method that combines the precision of tree search with the generality of random sampling. It has received considerable interest due to its spectacular success in the difficult problem of computer Go, but has also proved beneficial in a range of other domains. This paper is a survey of the literature to date, intended to provide a snapshot of the state of the art after the first five years of MCTS research. We outline the core algorithm's derivation, impart some structure on the many variations and enhancements that have been proposed, and summarise the results from the key game and non-game domains to which MCTS methods have been applied. A number of open research questions indicate that the field is ripe for future work.

Index Terms—Monte Carlo Tree Search (MCTS), Upper Confidence Bounds (UCB), Upper Confidence Bounds for Trees (UCT), Bandit-based methods, Artificial Intelligence (AI), Game search, Computer Go.

1 Introduction

Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. It has already had a profound impact on Artificial Intelligence (AI) approaches for domains that can be represented as trees of sequential decisions, particularly games and planning problems.

In the five years since MCTS was first described, it has become the focus of much AI research. Spurred on by some prolific achievements in the challenging task of computer Go, researchers are now in the process of attaining a better understanding of when and why MCTS succeeds and fails, and of extending and refining the basic algorithm. These developments are greatly increasing the range of games and other decision applications for which MCTS is a tool of choice, and pushing its performance to ever higher levels. MCTS has many attractions: it is a statistical anytime algorithm for which more computing power generally leads to better performance. It can be used with little or no domain knowledge, and has succeeded on difficult problems where other techniques have failed. Here we survey the range of published work on MCTS, to provide the reader with the tools to solve new problems using MCTS and to investigate this powerful approach to searching trees and directed graphs.

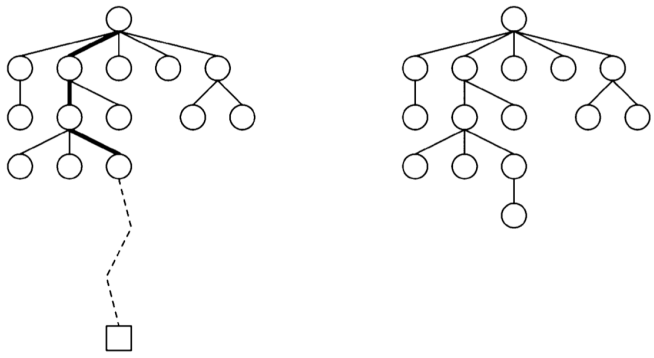


Fig. 1. The basic MCTS process [17].

1.1 Overview

The basic MCTS process is conceptually very simple, as shown in Figure 1 (from [17]). A tree[1] is built in an incremental and asymmetric manner. For each iteration of the algorithm, a *tree policy* is used to find the most urgent node of the current tree. The tree policy attempts to balance considerations of exploration (look in areas that have not been well sampled yet) and exploitation (look in areas which appear to be promising). A *simulation*[2] is then run from the selected node and the search tree updated according to the result. This involves the addition of a child node corresponding to the action taken from the selected node, and an update of the statistics of its ancestors. Moves are made during this simulation

- *C. Browne, S. Tavener and S. Colton are with the Department of Computing, Imperial College London, UK. E-mail: camb,sct110,sgc@doc.ic.ac.uk*
- *S. Lucas, P. Rohlfshagen, D. Perez and S. Samothrakis are with the School of Computer Science and Electronic Engineering, University of Essex, UK. E-mail: sml,prohlf,dperez,ssamot@essex.ac.uk*
- *E. Powley, D. Whitehouse and P.I. Cowling are with the School of Computing, Informatics and Media, University of Bradford, UK. E-mail: e.powley,d.whitehouse1,p.i.cowling@bradford.ac.uk*

Manuscript received October 22, 2011; revised January 12, 2012; accepted January 30, 2012. Digital Object Identifier 10.1109/TCIAIG.2012.2186810

1. Typically a game tree.
2. A random or statistically biased sequence of actions applied to the given state until a terminal condition is reached.

according to some *default policy*, which in the simplest case is to make uniform random moves. A great benefit of MCTS is that the values of intermediate states do not have to be evaluated, as for depth-limited minimax search, which greatly reduces the amount of domain knowledge required. Only the value of the terminal state at the end of each simulation is required.

While the basic algorithm (3.1) has proved effective for a wide range of problems, the full benefit of MCTS is typically not realised until this basic algorithm is adapted to suit the domain at hand. The thrust of a good deal of MCTS research is to determine those variations and enhancements best suited to each given situation, and to understand how enhancements from one domain may be used more widely.

1.2 Importance

Monte Carlo methods have a long history within numerical algorithms and have also had significant success in various AI game playing algorithms, particularly imperfect information games such as Scrabble and Bridge. However, it is really the success in computer Go, through the recursive application of Monte Carlo methods during the tree-building process, which has been responsible for much of the interest in MCTS. This is because Go is one of the few classic games for which human players are so far ahead of computer players. MCTS has had a dramatic effect on narrowing this gap, and is now competitive with the very best human players on small boards, though MCTS falls far short of their level on the standard 19×19 board. Go is a hard game for computers to play: it has a high branching factor, a deep tree, and lacks any known reliable heuristic value function for non-terminal board positions.

Over the last few years, MCTS has also achieved great success with many specific games, general games, and complex real-world planning, optimisation and control problems, and looks set to become an important part of the AI researcher's toolkit. It can provide an agent with some decision making capacity with very little domain-specific knowledge, and its selective sampling approach may provide insights into how other algorithms could be hybridised and potentially improved. Over the next decade we expect to see MCTS become a greater focus for increasing numbers of researchers, and to see it adopted as part of the solution to a great many problems in a variety of domains.

1.3 Aim

This paper is a comprehensive survey of known MCTS research at the time of writing (October 2011). This includes the underlying mathematics behind MCTS, the algorithm itself, its variations and enhancements, and its performance in a variety of domains. We attempt to convey the depth and breadth of MCTS research and its exciting potential for future development, and bring together common themes that have emerged.

This paper supplements the previous major survey in the field [170] by looking beyond MCTS for computer Go to the full range of domains to which it has now been applied. Hence we aim to improve the reader's understanding of how MCTS can be applied to new research questions and problem domains.

1.4 Structure

The remainder of this paper is organised as follows. In Section 2, we present central concepts of AI and games, introducing notation and terminology that set the stage for MCTS. In Section 3, the MCTS algorithm and its key components are described in detail. Section 4 summarises the main variations that have been proposed. Section 5 considers enhancements to the tree policy, used to navigate and construct the search tree. Section 6 considers other enhancements, particularly to simulation and backpropagation steps. Section 7 surveys the key applications to which MCTS has been applied, both in games and in other domains. In Section 8, we summarise the paper to give a snapshot of the state of the art in MCTS research, the strengths and weaknesses of the approach, and open questions for future research. The paper concludes with two tables that summarise the many variations and enhancements of MCTS and the domains to which they have been applied.

The References section contains a list of known MCTS-related publications, including book chapters, journal papers, conference and workshop proceedings, technical reports and theses. We do not guarantee that all cited works have been peer-reviewed or professionally recognised, but have erred on the side of inclusion so that the coverage of material is as comprehensive as possible. We identify almost 250 publications from the last five years of MCTS research.[3]

We present a brief Table of Contents due to the breadth of material covered:

3. One paper per week indicates the high level of research interest.

2 Background

This section outlines the background theory that led to the development of MCTS techniques. This includes decision theory, game theory, and Monte Carlo and bandit-based methods. We emphasise the importance of game theory, as this is the domain to which MCTS is most applied.

2.1 Decision Theory

Decision theory combines probability theory with utility theory to provide a formal and complete framework for decisions made under uncertainty [178, Ch.13].[4] Problems whose utility is defined by sequences of decisions were pursued in operations research and the study of Markov decision processes.

2.1.1 Markov Decision Processes (MDPs)

A *Markov decision process* (MDP) models sequential decision problems in fully observable environments using four components [178, Ch.17]:

- S: A set of states, with s_0 being the initial state.
- A: A set of actions.
- $T(s, a, s')$: A transition model that determines the probability of reaching state s' if action a is applied to state s.
- $R(s)$: A reward function.

Overall decisions are modelled as sequences of ($state$, $action$) pairs, in which each next state s' is decided by a probability distribution which depends on the current state s and the chosen action a. A *policy* is a mapping from states to actions, specifying which action will be chosen from each state in S. The aim is to find the policy π that yields the highest expected reward.

2.1.2 Partially Observable Markov Decision Processes

If each state is not fully observable, then a *Partially Observable Markov Decision Process* (POMDP) model must be used instead. This is a more complex formulation and requires the addition of:

- $O(s, o)$: An observation model that specifies the probability of perceiving observation o in state s.

The many MDP and POMDP approaches are beyond the scope of this review, but in all cases the optimal policy π is deterministic, in that each state is mapped to a single action rather than a probability distribution over actions.

2.2 Game Theory

Game theory extends decision theory to situations in which multiple agents interact. A game can be defined as a set of established rules that allows the interaction of one[5] or more players to produce specified outcomes.

4. We cite Russell and Norvig [178] as a standard AI reference, to reduce the number of non-MCTS references.
5. Single-player games constitute solitaire puzzles.

A game may be described by the following components:

- S: The set of states, where s_0 is the initial state.
- $S_T \subseteq S$: The set of terminal states.
- $n \in \mathbb{N}$: The number of players.
- A: The set of actions.
- $f : S \times A \to S$: The state transition function.
- $R : S \to \mathbb{R}^k$: The utility function.
- $\rho : S \to (0, 1, \ldots, n)$: Player about to act in each state.

Each game starts in state s_0 and progresses over time $t = 1, 2, \ldots$ until some terminal state is reached. Each player k_i takes an action (i.e. makes a move) that leads, via f, to the next state s_{t+1}. Each player receives a reward (defined by the utility function R) that assigns a value to their performance. These values may be arbitrary (e.g. positive values for numbers of points accumulated or monetary gains, negative values for costs incurred) but in many games it is typical to assign non-terminal states a reward of 0 and terminal states a value of $+1$, 0 or -1 (or $+1$, $+\frac{1}{2}$ and 0) for a win, draw or loss, respectively. These are the *game-theoretic* values of a terminal state.

Each player's *strategy* (policy) determines the probability of selecting action a given state s. The combination of players' strategies forms a *Nash equilibrium* if no player can benefit by unilaterally switching strategies [178, Ch.17]. Such an equilibrium always exists, but computing it for real games is generally intractable.

2.2.1 Combinatorial Games

Games are classified by the following properties:

- *Zero-sum*: Whether the reward to all players sums to zero (in the two-player case, whether players are in strict competition with each other).
- *Information*: Whether the state of the game is fully or partially observable to the players.
- *Determinism*: Whether chance factors play a part (also known as completeness, i.e. uncertainty over rewards).
- *Sequential*: Whether actions are applied sequentially or simultaneously.
- *Discrete*: Whether actions are discrete or applied in real-time.

Games with two players that are zero-sum, perfect information, deterministic, discrete and sequential are described as *combinatorial games*. These include games such as Go, Chess and Tic Tac Toe, as well as many others. Solitaire puzzles may also be described as combinatorial games played between the puzzle designer and the puzzle solver, although games with more than two players are not considered combinatorial due to the social aspect of coalitions that may arise during play. Combinatorial games make excellent test beds for AI experiments as they are controlled environments defined by simple rules, but which typically exhibit deep and complex play that can present significant research challenges, as amply demonstrated by Go.

2.2.2 AI in Real Games

Real-world games typically involve a delayed reward structure in which only those rewards achieved in the terminal states of the game accurately describe how well each player is performing. Games are therefore typically modelled as trees of decisions as follows:

- *Minimax* attempts to minimise the opponent's maximum reward at each state, and is the traditional search approach for two-player combinatorial games. The search is typically stopped prematurely and a value function used to estimate the outcome of the game, and the α-β heuristic is typically used to prune the tree. The $\max^n$ algorithm is the analogue of minimax for non-zero-sum games and/or games with more than two players.
- *Expectimax* generalises minimax to stochastic games in which the transitions from state to state are probabilistic. The value of a chance node is the sum of its children weighted by their probabilities, otherwise the search is identical to $\max^n$. Pruning strategies are harder due to the effect of chance nodes.
- *Miximax* is similar to single-player expectimax and is used primarily in games of imperfect information. It uses a predefined opponent strategy to treat opponent decision nodes as chance nodes.

2.3 Monte Carlo Methods

Monte Carlo methods have their roots in statistical physics where they have been used to obtain approximations to intractable integrals, and have since been used in a wide array of domains including games research.

Abramson [1] demonstrated that this sampling might be useful to approximate the game-theoretic value of a move. Adopting the notation used by Gelly and Silver [94], the Q-value of an action is simply the expected reward of that action:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s, a) z_i$$

where $N(s, a)$ is the number of times action a has been selected from state s, $N(s)$ is the number of times a game has been played out through state s, z_i is the result of the ith simulation played out from s, and $\mathbb{I}_i(s, a)$ is 1 if action a was selected from state s on the ith play-out from state s or 0 otherwise.

Monte Carlo approaches in which the actions of a given state are uniformly sampled are described as *flat Monte Carlo*. The power of flat Monte Carlo is demonstrated by Ginsberg [97] and Sheppard [199], who use such approaches to achieve world champion level play in Bridge and Scrabble respectively. However it is simple to construct degenerate cases in which flat Monte Carlo fails, as it does not allow for an opponent model [29].

Althöfer describes the *laziness* of flat Monte Carlo in non-tight situations [5]. He also describes unexpected *basin* behaviour that can occur [6], which might be used

to help find the optimal UCT search parameters for a given problem.

It is possible to improve the reliability of game-theoretic estimates by biasing action selection based on past experience. Using the estimates gathered so far, it is sensible to bias move selection towards those moves that have a higher intermediate reward.

2.4 Bandit-Based Methods

Bandit problems are a well-known class of sequential decision problems, in which one needs to choose amongst K actions (e.g. the K arms of a multi-armed bandit slot machine) in order to maximise the cumulative reward by consistently taking the optimal action. The choice of action is difficult as the underlying reward distributions are unknown, and potential rewards must be estimated based on past observations. This leads to the *exploitation-exploration dilemma*: one needs to balance the *exploitation* of the action currently believed to be optimal with the *exploration* of other actions that currently appear sub-optimal but may turn out to be superior in the long run.

A K-armed bandit is defined by random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, where i indicates the arm of the bandit [13], [119], [120]. Successive plays of bandit i yield $X_{i,1}, X_{i,2}, \ldots$ which are independently and identically distributed according to an unknown law with unknown expectation μ_i. The K-armed bandit problem may be approached using a *policy* that determines which bandit to play, based on past rewards.

2.4.1 Regret

The policy should aim to minimise the player's *regret*, which is defined after n plays as:

$$R_N = \mu^\star n - \mu_j \sum_{j=1}^{K} \mathbb{E}[T_j(n)]$$

where $\mu^\star$ is the best possible expected reward and $\mathbb{E}[T_j(n)]$ denotes the expected number of plays for arm j in the first n trials. In other words, the regret is the expected loss due to not playing the best bandit. It is important to highlight the necessity of attaching non-zero probabilities to all arms at all times, in order to ensure that the optimal arm is not missed due to temporarily promising rewards from a sub-optimal arm. It is thus important to place an upper confidence bound on the rewards observed so far that ensures this.

In a seminal paper, Lai and Robbins [124] showed there exists no policy with a regret that grows slower than $O(\ln n)$ for a large class of reward distributions. A policy is subsequently deemed to resolve the exploration-exploitation problem if the growth of regret is within a constant factor of this rate. The policies proposed by Lai and Robbins made use of *upper confidence indices*, which allow the policy to estimate the expected reward of a specific bandit once its index is computed. However, these indices were difficult to compute and Agrawal [2] introduced policies where the index could be expressed as a simple function of the total reward obtained so far by the bandit. Auer et al. [13] subsequently proposed a variant of Agrawal's index-based policy that has a finite-time regret logarithmically bound for arbitrary reward distributions with bounded support. One of these variants, UCB1, is introduced next.

2.4.2 Upper Confidence Bounds (UCB)

For bandit problems, it is useful to know the *upper confidence bound (UCB)* that any given arm will be optimal. The simplest UCB policy proposed by Auer et al. [13] is called *UCB1*, which has an expected logarithmic growth of regret uniformly over n (not just asymptotically) without any prior knowledge regarding the reward distributions (which have to have their support in $[0, 1]$). The policy dictates to play arm j that maximises:

$$\text{UCB1} = \overline{X}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

where $\overline{X}_j$ is the average reward from arm j, n_j is the number of times arm j was played and n is the overall number of plays so far. The reward term $\overline{X}_j$ encourages the *exploitation* of higher-reward choices, while the right hand term $\sqrt{\frac{2 \ln n}{n_j}}$ encourages the *exploration* of less-visited choices. The exploration term is related to the size of the one-sided confidence interval for the average reward within which the true expected reward falls with overwhelming probability [13, p 237].

3 Monte Carlo Tree Search

This section introduces the family of algorithms known as *Monte Carlo Tree Search* (MCTS). MCTS rests on two fundamental concepts: that the true value of an action may be approximated using random simulation; and that these values may be used efficiently to adjust the policy towards a best-first strategy. The algorithm progressively builds a partial game tree, guided by the results of previous exploration of that tree. The tree is used to estimate the values of moves, with these estimates (particularly those for the most promising moves) becoming more accurate as the tree is built.

3.1 Algorithm

The basic algorithm involves iteratively building a search tree until some predefined *computational budget* – typically a time, memory or iteration constraint – is reached, at which point the search is halted and the best-performing root action returned. Each node in the search tree represents a state of the domain, and directed links to child nodes represent actions leading to subsequent states.

Four steps are applied per search iteration [52]:

1) *Selection*: Starting at the root node, a child selection policy is recursively applied to descend through

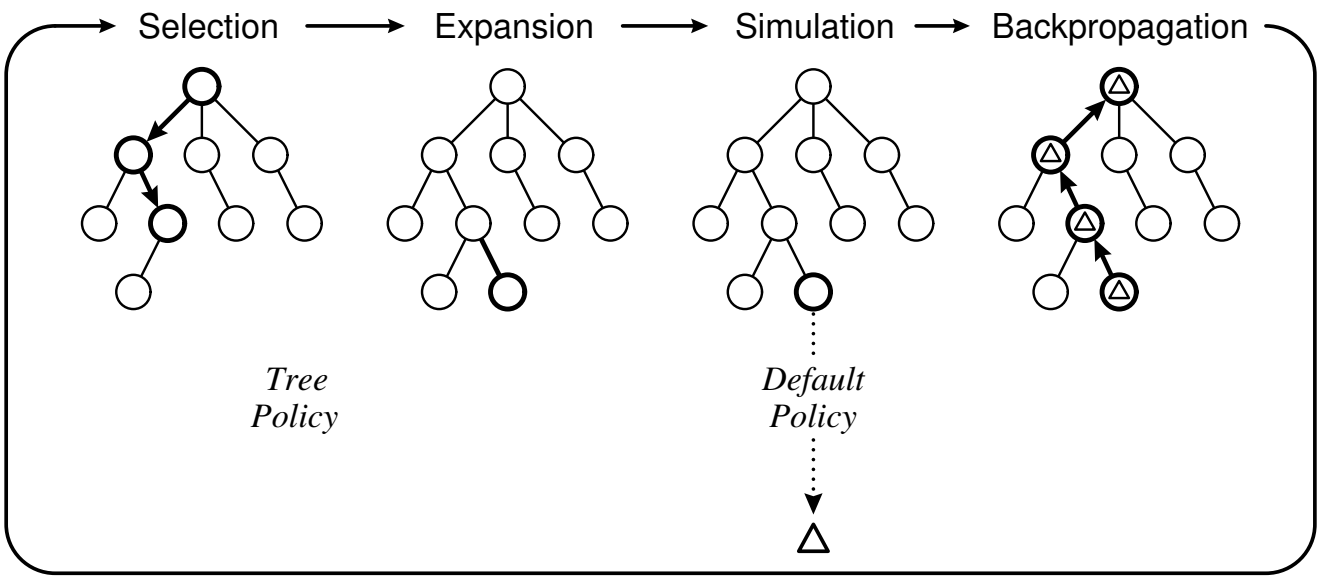


Fig. 2. One iteration of the general MCTS approach.

Algorithm 1 General MCTS approach.

```
function MCTSSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0))
```

the tree until the most urgent expandable node is reached. A node is *expandable* if it represents a non-terminal state and has unvisited (i.e. unexpanded) children.

2) *Expansion*: One (or more) child nodes are added to expand the tree, according to the available actions.
3) *Simulation*: A simulation is run from the new node(s) according to the default policy to produce an outcome.
4) *Backpropagation*: The simulation result is "backed up" (i.e. backpropagated) through the selected nodes to update their statistics.

These may be grouped into two distinct policies:

1) *Tree Policy*: Select or create a leaf node from the nodes already contained within the search tree (selection and expansion).
2) *Default Policy*: Play out the domain from a given non-terminal state to produce a value estimate (simulation).

The backpropagation step does not use a policy itself, but updates node statistics that inform future tree policy decisions.

These steps are summarised in pseudocode in Algorithm 1.[6] Here v_0 is the root node corresponding to state s_0, v_l is the last node reached during the tree policy stage and corresponds to state s_l, and Δ is the reward for the terminal state reached by running the default policy from state s_l. The result of the overall search $a(\text{BESTCHILD}(v_0))$ is the action a that leads to the best child of the root node v_0, where the exact definition of "best" is defined by the implementation.

Note that alternative interpretations of the term "simulation" exist in the literature. Some authors take it to mean the complete sequence of actions chosen per iteration during both the tree and default policies (see for example [93], [204], [94]) while most take it to mean the sequence of actions chosen using the default policy only. In this paper we shall understand the terms *playout* and *simulation* to mean "playing out the task to completion according to the default policy", i.e. the sequence of actions chosen after the tree policy steps of selection and expansion have been completed.

Figure 2 shows one iteration of the basic MCTS algorithm. Starting at the root node[7] t_0, child nodes are recursively selected according to some utility function until a node t_n is reached that either describes a terminal state or is not fully expanded (note that this is not necessarily a leaf node of the tree). An unvisited action a from this state s is selected and a new leaf node t_l is added to the tree, which describes the state s' reached from applying action a to state s. This completes the tree policy component for this iteration.

A simulation is then run from the newly expanded leaf node t_l to produce a reward value Δ, which is then

6. The simulation and expansion steps are often described and/or implemented in the reverse order in practice [52], [67].

7. Each node contains statistics describing at least a reward value and number of visits.

backpropagated up the sequence of nodes selected for this iteration to update the node statistics; each node's visit count is incremented and its average reward or Q value updated according to Δ. The reward value Δ may be a discrete (win/draw/loss) result or continuous reward value for simpler domains, or a vector of reward values relative to each agent p for more complex multi-agent domains.

As soon as the search is interrupted or the computation budget is reached, the search terminates and an action a of the root node t_0 is selected by some mechanism. Schadd [188] describes four criteria for selecting the winning action, based on the work of Chaslot et al [60]:

1) *Max child*: Select the root child with the highest reward.
2) *Robust child*: Select the most visited root child.
3) *Max-Robust child*: Select the root child with both the highest visit count and the highest reward. If none exist, then continue searching until an acceptable visit count is achieved [70].
4) *Secure child*: Select the child which maximises a lower confidence bound.

3.2 Development

Monte Carlo methods have been used extensively in games with randomness and partial observability [70] but they may be applied equally to deterministic games of perfect information. Following a large number of simulated games, starting at the current state and played until the end of the game, the initial move with the highest win-rate is selected to advance the game. In the majority of cases, actions were sampled uniformly at random (or with some game-specific heuristic bias) with no game-theoretic guarantees [119]. In other words, even if the iterative process is executed for an extended period of time, the move selected in the end may not be optimal [120].

Despite the lack of game-theoretic guarantees, the accuracy of the Monte Carlo simulations may often be improved by selecting actions according to the cumulative reward of the game episodes they were part of. This may be achieved by keeping track of the states visited in a tree. In 2006 Coulom [70] proposed a novel approach that combined Monte Carlo evaluations with tree search. His proposed algorithm iteratively runs random simulations from the current state to the end of the game: nodes close to the root are added to an incrementally growing tree, revealing structural information from the random sampling episodes. In particular, nodes in the tree are selected according to the estimated probability that they are better than the current best move.

The breakthrough for MCTS also came in 2006 and is primarily due to the selectivity mechanism proposed by Kocsis and Szepesvári, whose aim was to design a Monte Carlo search algorithm that had a small error probability if stopped prematurely and that converged to the game-theoretic optimum given sufficient time [120]. This may be achieved by reducing the estimation error of the nodes' values as quickly as possible. In order to do so, the algorithm must balance exploitation of the currently most promising action with exploration of alternatives which may later turn out to be superior. This exploitation-exploration dilemma can be captured by multi-armed bandit problems (2.4), and UCB1 [13] is an obvious choice for node selection. [8]

Table 1 summarises the milestones that led to the conception and popularisation of MCTS. It is interesting to note that the development of MCTS is the coming together of numerous different results in related fields of research in AI.

3.3 Upper Confidence Bounds for Trees (UCT)

This section describes the most popular algorithm in the MCTS family, the *Upper Confidence Bound for Trees* (UCT) algorithm. We provide a detailed description of the algorithm, and briefly outline the proof of convergence.

3.3.1 The UCT algorithm

The goal of MCTS is to approximate the (true) game-theoretic value of the actions that may be taken from the current state (3.1). This is achieved by iteratively building a partial search tree, as illustrated in Figure 2. How the tree is built depends on how nodes in the tree are selected. The success of MCTS, especially in Go, is primarily due to this tree policy. In particular, Kocsis and Szepesvári [119], [120] proposed the use of UCB1 (2.4.2) as tree policy. In treating the choice of child node as a multi-armed bandit problem, the value of a child node is the expected reward approximated by the Monte Carlo simulations, and hence these rewards correspond to random variables with unknown distributions.

UCB1 has some promising properties: it is very simple and efficient and guaranteed to be within a constant factor of the best possible bound on the growth of regret. It is thus a promising candidate to address the exploration-exploitation dilemma in MCTS: every time a node (action) is to be selected within the existing tree, the choice may be modelled as an independent multi-armed bandit problem. A child node j is selected to maximise:

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

where n is the number of times the current (parent) node has been visited, n_j the number of times child j has been visited and $C_p > 0$ is a constant. If more than one child node has the same maximal value, the tie is usually broken randomly [120]. The values of $X_{i,t}$ and thus of $\overline{X}_j$ are understood to be within $[0, 1]$ (this holds true for both the UCB1 and the UCT proofs). It is generally understood that $n_j = 0$ yields a UCT value of ∞, so that

8. Coulom [70] points out that the Boltzmann distribution often used in n-armed bandit problems is not suitable as a selection mechanism, as the underlying reward distributions in the tree are non-stationary.

1990	Abramson demonstrates that Monte Carlo simulations can be used to evaluate value of state [1].
1993	Brügmann [31] applies Monte Carlo methods to the field of computer Go.
1998	Ginsberg's GIB program competes with expert Bridge players.
1998	MAVEN defeats the world scrabble champion [199].
2002	Auer et al. [13] propose UCB1 for multi-armed bandit, laying the theoretical foundation for UCT.
2006	Coulom [70] describes Monte Carlo evaluations for tree-based search, coining the term Monte Carlo tree search.
2006	Kocsis and Szepesvari [119] associate UCB with tree-based search to give the UCT algorithm.
2006	Gelly et al. [96] apply UCT to computer Go with remarkable success, with their program MOGO.
2006	Chaslot et al. describe MCTS as a broader framework for game AI [52] and general domains [54].
2007	CADIAPLAYER becomes world champion General Game Player [83].
2008	MOGO achieves *dan* (master) level at 9×9 Go [128].
2009	FUEGO beats top human professional at 9×9 Go [81].
2009	MOHEX becomes world champion Hex player [7].

TABLE 1
Timeline of events leading to the widespread popularity of MCTS.

previously unvisited children are assigned the largest possible value, to ensure that all children of a node are considered at least once before any child is expanded further. This results in a powerful form of *iterated local search*.

There is an essential balance between the first (exploitation) and second (exploration) terms of the UCB equation. As each node is visited, the denominator of the exploration term increases, which decreases its contribution. On the other hand, if another child of the parent node is visited, the numerator increases and hence the exploration values of unvisited siblings increase. The exploration term ensures that each child has a nonzero probability of selection, which is essential given the random nature of the playouts. This also imparts an inherent *restart* property to the algorithm, as even low-reward children are guaranteed to be chosen eventually (given sufficient time), and hence different lines of play explored.

The constant in the exploration term C_p can be adjusted to lower or increase the amount of exploration performed. The value $C_p = 1/\sqrt{2}$ was shown by Kocsis and Szepesvári [120] to satisfy the Hoeffding ineqality with rewards in the range $[0, 1]$. With rewards outside this range, a different value of C_p may be needed and also certain enhancements[9] work better with a different value for C_p (7.1.3).

The rest of the algorithm proceeds as described in Section 3.1: if the node selected by UCB descent has children that are not yet part of the tree, one of those is chosen randomly and added to the tree. The default policy is then used until a terminal state has been reached. In the simplest case, this default policy is uniformly random. The value Δ of the terminal state s_T is then backpropagated to all nodes visited during this iteration, from the newly added node to the root.

Each node holds two values, the number $N(v)$ of times it has been visited and a value $Q(v)$ that corresponds to the total reward of all playouts that passed through this state (so that $Q(v)/N(v)$ is an approximation of the node's game-theoretic value). Every time a node is part of a playout from the root, its values are updated. Once some computational budget has been reached, the algorithm terminates and returns the best move found, corresponding to the child of the root with the highest visit count.

Algorithm 2 shows the UCT algorithm in pseudocode. This code is a summary of UCT descriptions from several sources, notably [94], but adapted to remove the two-player, zero-sum and turn order constraints typically found in the existing literature.

Each node v has four pieces of data associated with it: the associated state $s(v)$, the incoming action $a(v)$, the total simulation reward $Q(v)$ (a vector of real values), and the visit count $N(v)$ (a nonnegative integer). Instead of storing $s(v)$ for each node, it is often more efficient in terms of memory usage to recalculate it as TREEPOLICY descends the tree. The term $\Delta(v, p)$ denotes the component of the reward vector Δ associated with the current player p at node v.

The return value of the overall search in this case is $a(\text{BESTCHILD}(v_0, 0))$ which will give the action a that leads to the child with the highest reward,[10] since the exploration parameter c is set to 0 for this final call on the root node v_0. The algorithm could instead return the action that leads to the most visited child; these two options will usually – but not always! – describe the same action. This potential discrepancy is addressed in the Go program ERICA by continuing the search if the most visited root action is not also the one with the highest reward. This improved ERICA's winning rate against GNU GO from 47% to 55% [107].

Algorithm 3 shows an alternative and more efficient backup method for two-player, zero-sum games with alternating moves, that is typically found in the literature. This is analogous to the negamax variant of minimax search, in which scalar reward values are negated at each level in the tree to obtain the other player's reward. Note that while Δ is treated as a vector of rewards with an entry for each agent in Algorithm 2,[11] it is a single scalar value representing the reward to the agent running the

9. Such as RAVE (5.3.5).

10. The *max child* in Schadd's [188] terminology.

11. $\Delta(v, p)$ denotes the reward for p the player to move at node v.

Algorithm 2 The UCT algorithm.

```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))

function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v

function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and  a(v') = a
    return v'

function BESTCHILD(v, c)
    return argmax_{v' ∈ children of v}  Q(v')/N(v') + c sqrt(2 ln N(v) / N(v'))

function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s

function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

Algorithm 3 UCT backup for two players.

```
function BACKUPNEGAMAX(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ
        Δ ← −Δ
        v ← parent of v
```

search in Algorithm 3. Similarly, the node reward value $Q(v)$ may be treated as a vector of values for each player $Q(v, p)$ should circumstances dictate.

3.3.2 Convergence to Minimax

The key contributions of Kocsis and Szepesvári [119], [120] were to show that the bound on the regret of UCB1 still holds in the case of non-stationary reward distributions, and to empirically demonstrate the workings of MCTS with UCT on a variety of domains. Kocsis and Szepesvári then show that the failure probability at the root of the tree (i.e. the probability of selecting a suboptimal action) converges to zero at a polynomial rate as the number of games simulated grows to infinity. This proof implies that, given enough time (and memory), UCT allows MCTS to converge to the minimax tree and is thus optimal.

3.4 Characteristics

This section describes some of the characteristics that make MCTS a popular choice of algorithm for a variety of domains, often with notable success.

3.4.1 Aheuristic

One of the most significant benefits of MCTS is the lack of need for domain-specific knowledge, making it readily applicable to any domain that may be modelled using a tree. Although full-depth minimax is optimal in the game-theoretic sense, the quality of play for depth-limited minimax depends significantly on the heuristic used to evaluate leaf nodes. In games such as Chess, where reliable heuristics have emerged after decades of research, minimax performs admirably well. In cases such as Go, however, where branching factors are orders of magnitude larger and useful heuristics are much more difficult to formulate, the performance of minimax degrades significantly.

Although MCTS can be applied in its absence, significant improvements in performance may often be achieved using domain-specific knowledge. All top-performing MCTS-based Go programs now use game-specific information, often in the form of patterns (6.1.9). Such knowledge need not be complete as long as it is able to bias move selection in a favourable fashion.

There are trade-offs to consider when biasing move selection using domain-specific knowledge: one of the advantages of uniform random move selection is speed, allowing one to perform many simulations in a given time. Domain-specific knowledge usually drastically reduces the number of simulations possible, but may also reduce the variance of simulation results. The degree to which game-specific knowledge should be included, with respect to performance versus generality as well as speed trade-offs, is discussed in [77].

3.4.2 Anytime

MCTS backpropagates the outcome of each game immediately (the tree is built using playouts as opposed to stages [119]) which ensures all values are always up-to-date following every iteration of the algorithm. This allows the algorithm to return an action from the root at

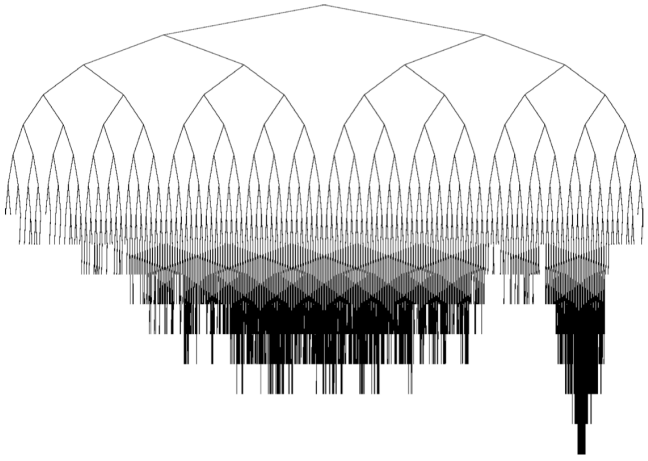


Fig. 3. Asymmetric tree growth [68].

any moment in time; allowing the algorithm to run for additional iterations often improves the result.

It is possible to approximate an anytime version of minimax using iterative deepening. However, the granularity of progress is much coarser as an entire ply is added to the tree on each iteration.

3.4.3 Asymmetric

The tree selection allows the algorithm to favour more promising nodes (without allowing the selection probability of the other nodes to converge to zero), leading to an asymmetric tree over time. In other words, the building of the partial tree is skewed towards more promising and thus more important regions. Figure 3 from [68] shows asymmetric tree growth using the BAST variation of MCTS (4.2).

The tree shape that emerges can even be used to gain a better understanding about the game itself. For instance, Williams [231] demonstrates that shape analysis applied to trees generated during UCT search can be used to distinguish between playable and unplayable games.

3.5 Comparison with Other Algorithms

When faced with a problem, the a priori choice between MCTS and minimax may be difficult. If the game tree is of nontrivial size and no reliable heuristic exists for the game of interest, minimax is unsuitable but MCTS is applicable (3.4.1). If domain-specific knowledge is readily available, on the other hand, both algorithms may be viable approaches.

However, as pointed out by Ramanujan et al. [164], MCTS approaches to games such as Chess are not as successful as for games such as Go. They consider a class of synthetic spaces in which UCT significantly outperforms minimax. In particular, the model produces bounded trees where there is exactly one optimal action per state; sub-optimal choices are penalised with a fixed additive cost. The systematic construction of the tree ensures that the true minimax values are known.[12] In this domain, UCT clearly outperforms minimax and the gap in performance increases with tree depth.

Ramanujan et al. [162] argue that UCT performs poorly in domains with many *trap states* (states that lead to losses within a small number of moves), whereas iterative deepening minimax performs relatively well. Trap states are common in Chess but relatively uncommon in Go, which may go some way towards explaining the algorithms' relative performance in those games.

3.6 Terminology

The terms MCTS and UCT are used in a variety of ways in the literature, sometimes inconsistently, potentially leading to confusion regarding the specifics of the algorithm referred to. For the remainder of this survey, we adhere to the following meanings:

- ***Flat Monte Carlo***: A Monte Carlo method with uniform move selection and no tree growth.
- ***Flat UCB***: A Monte Carlo method with bandit-based move selection (2.4) but no tree growth.
- ***MCTS***: A Monte Carlo method that builds a tree to inform its policy online.
- ***UCT***: MCTS with any UCB tree selection policy.
- ***Plain UCT***: MCTS with UCB1 as proposed by Kocsis and Szepesvári [119], [120].

In other words, "plain UCT" refers to the specific algorithm proposed by Kocsis and Szepesvári, whereas the other terms refer more broadly to families of algorithms.

4 Variations

Traditional game AI research focusses on zero-sum games with two players, alternating turns, discrete action spaces, deterministic state transitions and perfect information. While MCTS has been applied extensively to such games, it has also been applied to other domain types such as single-player games and planning problems, multi-player games, real-time games, and games with uncertainty or simultaneous moves. This section describes the ways in which MCTS has been adapted to these domains, in addition to algorithms that adopt ideas from MCTS without adhering strictly to its outline.

4.1 Flat UCB

Coquelin and Munos [68] propose *flat UCB* which effectively treats the leaves of the search tree as a single multi-armed bandit problem. This is distinct from *flat Monte Carlo* search (2.3) in which the actions for a given state are uniformly sampled and no tree is built. Coquelin and Munos [68] demonstrate that flat UCB retains the adaptivity of standard UCT while improving its regret bounds in certain worst cases where UCT is overly optimistic.

12. This is related to P-game trees (7.3).

4.2 Bandit Algorithm for Smooth Trees (BAST)

Coquelin and Munos [68] extend the flat UCB model to suggest a *Bandit Algorithm for Smooth Trees* (BAST), which uses assumptions on the *smoothness* of rewards to identify and ignore branches that are suboptimal with high confidence. They applied BAST to Lipschitz function approximation and showed that when allowed to run for infinite time, the only branches that are expanded indefinitely are the optimal branches. This is in contrast to plain UCT, which expands all branches indefinitely.

4.3 Learning in MCTS

MCTS can be seen as a type of *Reinforcement Learning* (RL) algorithm, so it is interesting to consider its relationship with temporal difference learning (arguably the canonical RL algorithm).

4.3.1 Temporal Difference Learning (TDL)

Both *temporal difference learning* (TDL) and MCTS learn to take actions based on the values of states, or of state-action pairs. Under certain circumstances the algorithms may even be equivalent [201], but TDL algorithms do not usually build trees, and the equivalence only holds when all the state values can be stored directly in a table. MCTS estimates temporary state values in order to decide the next move, whereas TDL learns the long-term value of each state that then guides future behaviour. Silver et al. [202] present an algorithm that combines MCTS with TDL using the notion of permanent and transient memories to distinguish the two types of state value estimation. TDL can learn heuristic value functions to inform the tree policy or the simulation (playout) policy.

4.3.2 Temporal Difference with Monte Carlo (TDMC(λ))

Osaki et al. describe the *Temporal Difference with Monte Carlo* (TDMC(λ)) algorithm as "a new method of reinforcement learning using winning probability as substitute rewards in non-terminal positions" [157] and report superior performance over standard TD learning for the board game Othello (7.3).

4.3.3 Bandit-Based Active Learner (BAAL)

Rolet et al. [175], [173], [174] propose the *Bandit-based Active Learner* (BAAL) method to address the issue of small training sets in applications where data is sparse. The notion of *active learning* is formalised under bounded resources as a finite horizon reinforcement learning problem with the goal of minimising the generalisation error. Viewing active learning as a single-player game, the optimal policy is approximated by a combination of UCT and billiard algorithms [173]. Progressive widening (5.5.1) is employed to limit the degree of exploration by UCB1 to give promising empirical results.

4.4 Single-Player MCTS (SP-MCTS)

Schadd et al. [191], [189] introduce a variant of MCTS for single-player games, called *Single-Player Monte Carlo Tree Search* (SP-MCTS), which adds a third term to the standard UCB formula that represents the "possible deviation" of the node. This term can be written

$$\sqrt{\sigma^2 + \frac{D}{n_i}},$$

where σ^2 is the variance of the node's simulation results, n_i is the number of visits to the node, and D is a constant. The $\frac{D}{n_i}$ term can be seen as artificially inflating the standard deviation for infrequently visited nodes, so that the rewards for such nodes are considered to be less certain. The other main difference between SP-MCTS and plain UCT is the use of a heuristically guided default policy for simulations.

Schadd et al. [191] point to the need for *Meta-Search* (a higher-level search method that uses other search processes to arrive at an answer) in some cases where SP-MCTS on its own gets caught in local maxima. They found that periodically restarting the search with a different random seed and storing the best solution over all runs considerably increased the performance of their SameGame player (7.4).

Björnsson and Finnsson [21] discuss the application of standard UCT to single-player games. They point out that averaging simulation results can hide a strong line of play if its siblings are weak, instead favouring regions where all lines of play are of medium strength. To counter this, they suggest tracking maximum simulation results at each node in addition to average results; the averages are still used during search.

Another modification suggested by Björnsson and Finnsson [21] is that when simulation finds a strong line of play, it is stored in the tree in its entirety. This would be detrimental in games of more than one player since such a strong line would probably rely on the unrealistic assumption that the opponent plays weak moves, but for single-player games this is not an issue.

4.4.1 Feature UCT Selection (FUSE)

Gaudel and Sebag introduce *Feature UCT Selection* (FUSE), an adaptation of UCT to the combinatorial optimisation problem of feature selection [89]. Here, the problem of choosing a subset of the available features is cast as a single-player game whose states are all possible subsets of features and whose actions consist of choosing a feature and adding it to the subset.

To deal with the large branching factor of this game, FUSE uses UCB1-Tuned (5.1.1) and RAVE (5.3.5). FUSE also uses a game-specific approximation of the reward function, and adjusts the probability of choosing the stopping feature during simulation according to the depth in the tree. Gaudel and Sebag [89] apply FUSE to three benchmark data sets from the NIPS 2003 FS Challenge competition (7.8.4).

4.5 Multi-player MCTS

The central assumption of minimax search (2.2.2) is that the searching player seeks to maximise their reward while the opponent seeks to minimise it. In a two-player zero-sum game, this is equivalent to saying that each player seeks to maximise their own reward; however, in games of more than two players, this equivalence does not necessarily hold.

The simplest way to apply MCTS to multi-player games is to adopt the $\max^n$ idea: each node stores a vector of rewards, and the selection procedure seeks to maximise the UCB value calculated using the appropriate component of the reward vector. Sturtevant [207] shows that this variant of UCT converges to an optimal equilibrium strategy, although this strategy is not precisely the $\max^n$ strategy as it may be mixed.

Cazenave [40] applies several variants of UCT to the game of Multi-player Go (7.1.5) and considers the possibility of players acting in coalitions. The search itself uses the $\max^n$ approach described above, but a rule is added to the simulations to avoid playing moves that adversely affect fellow coalition members, and a different scoring system is used that counts coalition members' stones as if they were the player's own.

There are several ways in which such coalitions can be handled. In *Paranoid UCT*, the player considers that all other players are in coalition against him. In *UCT with Alliances*, the coalitions are provided explicitly to the algorithm. In *Confident UCT*, independent searches are conducted for each possible coalition of the searching player with one other player, and the move chosen according to whichever of these coalitions appears most favourable. Cazenave [40] finds that Confident UCT performs worse than Paranoid UCT in general, but the performance of the former is better when the algorithms of the other players (i.e. whether they themselves use Confident UCT) are taken into account. Nijssen and Winands [155] describe the *Multi-Player Monte-Carlo Tree Search Solver* (MP-MCTS-Solver) version of their MCTS Solver enhancement (5.4.1).

4.5.1 Coalition Reduction

Winands and Nijssen describe the *coalition reduction* method [156] for games such as Scotland Yard (7.7) in which multiple cooperative opponents can be reduced to a single effective opponent. Note that rewards for those opponents who are not the root of the search must be biased to stop them getting lazy [156].

4.6 Multi-agent MCTS

Marcolino and Matsubara [139] describe the simulation phase of UCT as a single agent playing against itself, and instead consider the effect of having multiple agents (i.e. multiple simulation policies). Specifically, the different agents in this case are obtained by assigning different priorities to the heuristics used in Go program FUEGO's simulations [81]. If the right subset of agents is chosen (or learned, as in [139]), using multiple agents improves playing strength. Marcolino and Matsubara [139] argue that the emergent properties of interactions between different agent types lead to increased exploration of the search space. However, finding the set of agents with the correct properties (i.e. those that increase playing strength) is computationally intensive.

4.6.1 Ensemble UCT

Fern and Lewis [82] investigate an *Ensemble UCT* approach, in which multiple instances of UCT are run independently and their root statistics combined to yield the final result. This approach is closely related to root parallelisation (6.3.2) and also to determinization (4.8.1).

Chaslot et al. [59] provide some evidence that, for Go, Ensemble UCT with n instances of m iterations each outperforms plain UCT with mn iterations, i.e. that Ensemble UCT outperforms plain UCT given the same total number of iterations. However, Fern and Lewis [82] are not able to reproduce this result on other experimental domains.

4.7 Real-time MCTS

Traditional board games are turn-based, often allowing each player considerable time for each move (e.g. several minutes for Go). However, real-time games tend to progress constantly even if the player attempts no move, so it is vital for an agent to act quickly. The largest class of real-time games are video games, which – in addition to the real-time element – are usually also characterised by uncertainty (4.8), massive branching factors, simultaneous moves (4.8.10) and open-endedness. Developing strong artificial players for such games is thus particularly challenging and so far has been limited in success.

Simulation-based (anytime) algorithms such as MCTS are well suited to domains in which time per move is strictly limited. Furthermore, the asymmetry of the trees produced by MCTS may allow a better exploration of the state space in the time available. Indeed, MCTS has been applied to a diverse range of real-time games of increasing complexity, ranging from Tron and Ms. Pac-Man to a variety of real-time strategy games akin to Starcraft. In order to make the complexity of real-time video games tractable, approximations may be used to increase the efficiency of the forward model.

4.8 Nondeterministic MCTS

Traditional game AI research also typically focusses on deterministic games with perfect information, i.e. games without chance events in which the state of the game is fully observable to all players (2.2). We now consider games with *stochasticity* (chance events) and/or *imperfect information* (partial observability of states).

Opponent modelling (i.e. determining the opponent's policy) is much more important in games of imperfect information than games of perfect information, as the opponent's policy generally depends on their hidden

information, hence guessing the former allows the latter to be inferred. Section 4.8.9 discusses this in more detail.

4.8.1 Determinization

A stochastic game with imperfect information can be transformed into a deterministic game with perfect information, by fixing the outcomes of all chance events and making states fully observable. For example, a card game can be played with all cards face up, and a game with dice can be played with a predetermined sequence of dice rolls known to all players. *Determinization*[13] is the process of sampling several such instances of the deterministic game with perfect information, analysing each with standard AI techniques, and combining those analyses to obtain a decision for the full game.

Cazenave [36] applies depth-1 search with Monte Carlo evaluation to the game of Phantom Go (7.1.5). At the beginning of each iteration, the game is determinized by randomly placing the opponent's hidden stones. The evaluation and search then continues as normal.

4.8.2 Hindsight optimisation (HOP)

Hindsight optimisation (HOP) provides a more formal basis to determinization for single-player stochastic games of perfect information. The idea is to obtain an upper bound on the expected reward for each move by assuming the ability to optimise one's subsequent strategy with "hindsight" knowledge of future chance outcomes. This upper bound can easily be approximated by determinization. The bound is not particularly tight, but is sufficient for comparing moves.

Bjarnason et al. [20] apply a combination of HOP and UCT to the single-player stochastic game of Klondike solitaire (7.7). Specifically, UCT is used to solve the determinized games sampled independently by HOP.

4.8.3 Sparse UCT

Sparse UCT is a generalisation of this HOP-UCT procedure also described by Bjarnason et al. [20]. In Sparse UCT, a node may have several children corresponding to the same move, each child corresponding to a different stochastic outcome of that move. Moves are selected as normal by UCB, but the traversal to child nodes is stochastic, as is the addition of child nodes during expansion. Bjarnason et al. [20] also define an *ensemble* version of Sparse UCT, whereby several search trees are constructed independently and their results (the expected rewards of actions from the root) are averaged, which is similar to Ensemble UCT (4.6.1).

Borsboom et al. [23] suggest ways of combining UCT with HOP-like ideas, namely *early probabilistic guessing* and *late random guessing*. These construct a single UCT tree, and determinize the game at different points in each iteration (at the beginning of the selection and simulation phases, respectively). Late random guessing significantly outperforms early probabilistic guessing.

13. For consistency with the existing literature, we use the Americanised spelling "determinization".

4.8.4 Information Set UCT (ISUCT)

Strategy fusion is a problem with determinization techniques, which involves the incorrect assumption that different moves can be chosen from different states in the same information set. Long et al. [130] describe how this can be measured using synthetic game trees.

To address the problem of strategy fusion in determinized UCT, Whitehouse et al. [230] propose *information set UCT* (ISUCT), a variant of MCTS that operates directly on trees of information sets. All information sets are from the point of view of the root player. Each iteration samples a determinization (a state from the root information set) and restricts selection, expansion and simulation to those parts of the tree compatible with the determinization. The UCB formula is modified to replace the "parent visit" count with the number of parent visits in which the child was compatible.

For the experimental domain in [230], ISUCT fails to outperform determinized UCT overall. However, ISUCT is shown to perform well in precisely those situations where access to hidden information would have the greatest effect on the outcome of the game.

4.8.5 Multiple MCTS

Auger [16] proposes a variant of MCTS for games of imperfect information, called *Multiple Monte Carlo Tree Search* (MMCTS), in which multiple trees are searched simultaneously. Specifically, there is a tree for each player, and the search descends and updates all of these trees simultaneously, using statistics in the tree for the relevant player at each stage of selection. This more accurately models the differences in information available to each player than searching a single tree. MMCTS uses EXP3 (5.1.3) for selection.

Auger [16] circumvents the difficulty of computing the correct belief distribution at non-initial points in the game by using MMCTS in an offline manner. MMCTS is run for a large number of simulations (e.g. 50 million) to construct a partial game tree rooted at the initial state of the game, and the player's policy is read directly from this pre-constructed tree during gameplay.

4.8.6 UCT+

Van den Broeck et al. [223] describe a variant of MCTS for miximax trees (2.2.2) in which opponent decision nodes are treated as chance nodes with probabilities determined by an opponent model. The algorithm is called *UCT+*, although it does not use UCB: instead, actions are selected to maximise

$$\overline{X}_j + c\sigma_{\overline{X}_j},$$

where $\overline{X}_j$ is the average reward from action j, $\sigma_{\overline{X}_j}$ is the standard error on $\overline{X}_j$, and c is a constant. During backpropagation, each visited node's $\overline{X}_j$ and $\sigma_{\overline{X}_j}$ values are updated according to their children; at opponent nodes and chance nodes, the calculations are weighted by the probabilities of the actions leading to each child.

4.8.7 Monte Carlo α-β (MC$_{\alpha\beta}$)

Monte Carlo α-β (MC$_{\alpha\beta}$) combines MCTS with traditional tree search by replacing the default policy with a shallow α-β search. For example, Winands and Björnnsson [232] apply a selective two-ply α-β search in lieu of a default policy for their program MC$_{\alpha\beta}$, which is currently the strongest known computer player for the game Lines of Action (7.2). An obvious consideration in choosing MC$_{\alpha\beta}$ for a domain is that a reliable heuristic function must be known in order to drive the α-β component of the search, which ties the implementation closely with the domain.

4.8.8 Monte Carlo Counterfactual Regret (MCCFR)

Counterfactual regret (CFR) is an algorithm for computing approximate Nash equilibria for games of imperfect information. Specifically, at time $t+1$ the policy plays actions with probability proportional to their positive counterfactual regret at time t, or with uniform probability if no actions have positive counterfactual regret. A simple example of how CFR operates is given in [177].

CFR is impractical for large games, as it requires traversal of the entire game tree. Lanctot et al. [125] propose a modification called *Monte Carlo counterfactual regret* (MCCFR). MCCFR works by sampling *blocks* of terminal histories (paths through the game tree from root to leaf), and computing immediate counterfactual regrets over those blocks. MCCFR can be used to minimise, with high probability, the overall regret in the same way as CFR. CFR has also been used to create agents capable of exploiting the non-Nash strategies used by UCT agents [196].

4.8.9 Inference and Opponent Modelling

In a game of imperfect information, it is often possible to *infer* hidden information from opponent actions, such as learning opponent policies directly using Bayesian inference and relational probability tree learning. The opponent model has two parts – a prior model of a general opponent, and a corrective function for the specific opponent – which are learnt from samples of previously played games. Ponsen et al. [159] integrate this scheme with MCTS to infer probabilities for hidden cards, which in turn are used to determinize the cards for each MCTS iteration. When the MCTS selection phase reaches an opponent decision node, it uses the mixed policy induced by the opponent model instead of bandit-based selection.

4.8.10 Simultaneous Moves

Simultaneous moves can be considered a special case of hidden information: one player chooses a move but conceals it, then the other player chooses a move and both are revealed.

Shafiei et al. [196] describe a simple variant of UCT for games with simultaneous moves. Shafiei et al. [196] argue that this method will not converge to the Nash equilibrium in general, and show that a UCT player can thus be exploited.

Teytaud and Flory [216] use a similar technique to Shafiei et al. [196], the main difference being that they use the EXP3 algorithm (5.1.3) for selection at simultaneous move nodes (UCB is still used at other nodes). EXP3 is explicitly probabilistic, so the tree policy at simultaneous move nodes is mixed. Teytaud and Flory [216] find that coupling EXP3 with UCT in this way performs much better than simply using the UCB formula at simultaneous move nodes, although performance of the latter does improve if random exploration with fixed probability is introduced.

Samothrakis et al. [184] apply UCT to the simultaneous move game Tron (7.6). However, they simply avoid the simultaneous aspect by transforming the game into one of alternating moves. The nature of the game is such that this simplification is not usually detrimental, although Den Teuling [74] identifies such a degenerate situation and suggests a game-specific modification to UCT to handle this.

4.9 Recursive Approaches

The following methods recursively apply a Monte Carlo technique to grow the search tree. These have typically had success with single-player puzzles and similar optimisation tasks.

4.9.1 Reflexive Monte Carlo Search

Reflexive Monte Carlo search [39] works by conducting several recursive layers of Monte Carlo simulations, each layer informed by the one below. At level 0, the simulations simply use random moves (so a level 0 reflexive Monte Carlo search is equivalent to a 1-ply search with Monte Carlo evaluation). At level $n > 0$, the simulation uses level $n-1$ searches to select each move.

4.9.2 Nested Monte Carlo Search

A related algorithm to reflexive Monte Carlo search is *nested Monte Carlo search* (NMCS) [42]. The key difference is that nested Monte Carlo search memorises the best sequence of moves found at each level of the search.

Memorising the best sequence so far and using this knowledge to inform future iterations can improve the performance of NMCS in many domains [45]. Cazenaze et al. describe the application of NMCS to the bus regulation problem (7.8.3) and find that NMCS with memorisation clearly outperforms plain NMCS, which in turn outperforms flat Monte Carlo and rule-based approaches [45].

Cazenave and Jouandeau [49] describe parallelised implementations of NMCS. Cazenave [43] also demonstrates the successful application of NMCS methods for the generation of expression trees to solve certain mathematical problems (7.8.2). Rimmel et al. [168] apply a version of nested Monte Carlo search to the Travelling Salesman Problem (TSP) with *time windows* (7.8.1).

4.9.3 Nested Rollout Policy Adaptation (NRPA)

Nested Rollout Policy Adaptation (NRPA) is an extension of nested Monte Carlo search in which a domain-specific policy is associated with the action leading to each child [176]. These are tuned adaptively starting from a uniform random policy. NRPA has achieved superior results in puzzle optimisation tasks, including beating the human world record for Morpion Solitaire (7.4).

4.9.4 Meta-MCTS

Chaslot et al. [56] replace the default policy with a nested MCTS program that plays a simulated sub-game in their *Meta-MCTS* algorithm. They describe two versions of Meta-MCTS: *Quasi Best-First* (which favours exploitation) and *Beta Distribution Sampling* (which favours exploration). Both variants improved the playing strength of the program MoGo for 9×9 Go when used for generating opening books.

4.9.5 Heuristically Guided Swarm Tree Search

Edelkamp et al. [78] introduce the *Heuristically Guided Swarm Tree Search* (HGSTS) algorithm. This algorithm conducts an exhaustive breadth-first search to a certain level in the game tree, adding a node to the UCT tree for each game tree node at that level. These nodes are inserted into a priority queue, prioritised in descending order of UCB value. The algorithm repeatedly takes the front k elements of the queue and executes an iteration of UCT starting from each of them. Heuristics are used to weight the move probabilities used during simulation. Edelkamp et al. [78] describe a parallel implementation of this algorithm (using a technique they term *set-based parallelisation*), and also describe how the breadth-first search portion of the algorithm can be implemented on a GPU for a significant gain in speed.

4.10 Sample-Based Planners

Planners for many complex structured domains can be learned with tractable sample complexity if near optimal policies are known. These are generally similar to Single-Player MCTS techniques, but tend to be applied to domains other than games.

4.10.1 Forward Search Sparse Sampling (FSSS)

Walsh et al. [227] show how to replace known policies with sample-based planners in concert with sample-efficient learners in a method called *Forward Search Sparse Sampling* (FSSS). They describe a negative case for UCT's runtime that can require exponential computation to optimise, in support of their approach.

Asmuth and Littman [9] extend the FSSS technique to Bayesian FSSS (BFS3), which approaches Bayes-optimality as the program's computational budget is increased. They observe that "learning is planning" [10].

4.10.2 Threshold Ascent for Graphs (TAG)

Threshold Ascent for Graphs (TAG) is a method that extends the MCTS paradigm by maximizing an objective function over the sinks of directed acyclic graphs [166] [73]. The algorithm evaluates nodes through random simulation and grows the subgraph in the most promising directions by considering local maximum k-armed bandits. TAG has demonstrated superior performance over standard optimisation methods for automatic performance tuning using DFT and FFT linear transforms in adaptive libraries.

4.10.3 RRTs

Rapidly-exploring Random Trees (RRTs), a special case of Rapidly-exploring Dense Trees (RTDs), were first introduced by Steven LaValle [126]. The basic idea of RRTs is to drive the exploration towards unexplored portions of the search space, incrementally pulling the search tree towards them. The tree is built in a similar way to MCTS, by repeating this process multiple times to explore the search space. RRTs share many ideas with MCTS, such as the use of state-action pairs, the tree structure, and the exploration of the search space based on random actions, often guided by heuristics.

4.10.4 UNLEO

Auger and Teytaud describe the *UNLEO*[14] algorithm as "a heuristic approximation of an optimal optimization algorithm using Upper Confidence Trees" [15]. UNLEO is based on the *No Free Lunch* (NFL) and *Continuous Free Lunch* (CFL) theorems and was inspired by the known optimality of Bayesian inference for supervised learning when a prior distribution is available. Bayesian inference is often very expensive, so Auger and Teytaud use UCT to make the evaluation of complex objective functions achievable.

4.10.5 UCTSAT

Previti et al. [160] introduce the UCTSAT class of algorithms to investigate the application of UCT approaches to the satisfiability of *conjunctive normal form* (CNF) problems (7.8.2). They describe the following variations:

- UCTSAT$_{cp}$ generates a random assignment of variables for each playout.
- UCTSAT$_{sbs}$ assigns variables one-by-one with random legal (satisfying) values.
- UCTSAT$_{h}$ replaces playouts with a simple heuristic based on the fraction of satisfied clauses.

4.10.6 ρUCT

Veness et al. [226] introduce ρUCT, a generalisation of UCT that approximates a finite horizon expectimax operation given an environment model ρ. ρUCT builds a sparse search tree composed of interleaved decision and chance nodes to extend UCT to a wider class of

14. The derivation of this term is not given.

problem domains. They describe the application of ρUCT to create their MC-AIXA agent, which approximates the AIXA[15] model. MC-AIXA was found to approach optimal performance for several problem domains (7.8).

4.10.7 Monte Carlo Random Walks (MRW)

Monte Carlo Random Walks (MRW) selectively build the search tree using random walks [238]. Xie et al. describe the *Monte Carlo Random Walk-based Local Tree Search* (MRW-LTS) method which extends MCRW to concentrate on local search more than standard MCTS methods, allowing good performance in some difficult planning problems [238].

4.10.8 Mean-based Heuristic Search for Anytime Planning (MHSP)

Pellier et al. [158] propose an algorithm for planning problems, called *Mean-based Heuristic Search for Anytime Planning* (MHSP), based on MCTS. There are two key differences between MHSP and a conventional MCTS algorithm. First, MHSP entirely replaces the random simulations of MCTS with a heuristic evaluation function: Pellier et al. [158] argue that random exploration of the search space for a planning problem is inefficient, since the probability of a given simulation actually finding a solution is low. Second, MHSP's selection process simply uses average rewards with no exploration term, but initialises the nodes with "optimistic" average values. In contrast to many planning algorithms, MHSP can operate in a truly anytime fashion: even before a solution has been found, MHSP can yield a good partial plan.

5 Tree Policy Enhancements

This section describes modifications proposed for the tree policy of the core MCTS algorithm, in order to improve performance. Many approaches use ideas from traditional AI search such as α-β, while some have no existing context and were developed specifically for MCTS. These can generally be divided into two categories:

- *Domain Independent:* These are enhancements that could be applied to any domain without prior knowledge about it. These typically offer small improvements or are better suited to a particular type of domain.
- *Domain Dependent:* These are enhancements specific to particular domains. Such enhancements might use prior knowledge about a domain or otherwise exploit some unique aspect of it.

This section covers those enhancements specific to the tree policy, i.e. the selection and expansion steps.

15. AIXI is a mathematical approach based on a Bayesian optimality notion for general reinforcement learning agents.

5.1 Bandit-Based Enhancements

The bandit-based method used for node selection in the tree policy is central to the MCTS method being used. A wealth of different upper confidence bounds have been proposed, often improving bounds or performance in particular circumstances such as dynamic environments.

5.1.1 UCB1-Tuned

UCB1-Tuned is an enhancement suggested by Auer et al. [13] to tune the bounds of UCB1 more finely. It replaces the upper confidence bound $\sqrt{2\ln n/n_j}$ with:

$$\sqrt{\frac{\ln n}{n_j}\min\{\frac{1}{4}, V_j(n_j)\}}$$

where:

$$V_j(s) = (1/2\sum_{\tau=1}^{s} X_{j,\tau}^2) - \overline{X}_{j,s}^2 + \sqrt{\frac{2\ln t}{s}}$$

which means that machine j, which has been played s times during the first t plays, has a variance that is at most the sample variance plus $\sqrt{2\ln t)/s}$ [13]. It should be noted that Auer et al. were unable to prove a regret bound for UCB1-Tuned, but found it performed better than UCB1 in their experiments. UCB1-Tuned has subsequently been used in a variety of MCTS implementations, including Go [95], Othello [103] and the real-time game Tron [184].

5.1.2 Bayesian UCT

Tesauro et al. [213] propose that the Bayesian framework potentially allows much more accurate estimation of node values and node uncertainties from limited numbers of simulation trials. Their Bayesian MCTS formalism introduces two tree policies:

$$\text{maximise } B_i = \mu_i + \sqrt{\frac{2\ln N}{n_i}}$$

where μ_i replaces the average reward of the node with the mean of an extremum (minimax) distribution P_i (assuming independent random variables) and:

$$\text{maximise } B_i = \mu_i + \sqrt{\frac{2\ln N}{n_i}}\sigma_i$$

where σ_i is the square root of the variance of P_i.

Tesauro et al. suggest that the first equation is a strict improvement over UCT if the independence assumption and leaf node priors are correct, while the second equation is motivated by the central limit theorem. They provide convergence proofs for both equations and carry out an empirical analysis in an artificial scenario based on an "idealized bandit-tree simulator". The results indicate that the second equation outperforms the first, and that both outperform the standard UCT approach (although UCT is considerably quicker). McInerney et al. [141] also describe a Bayesian approach to bandit selection, and argue that this, in principle, avoids the need to choose between exploration and exploitation.

5.1.3 EXP3

The *Exploration-Exploitation with Exponential weights* (EXP3) algorithm, originally proposed by Auer et al. [14] and further analysed by Audibert and Bubeck [11], applies in the stochastic case (and hence also in the adversarial case). The EXP3 policy operates as follows:

- Draw an arm I_t from the probability distribution p_t.
- Compute the estimated gain for each arm.
- Update the cumulative gain.

Then one can compute the new probability distribution over the arms. EXP3 has been used in conjunction with UCT to address games with partial observability and simultaneous moves [216], [217].

5.1.4 Hierarchical Optimistic Optimisation for Trees

Bubeck et al. describe the *Hierarchical Optimistic Optimisation* (HOO) algorithm, which is a a generalisation of stochastic bandits [32], [33], [34]. HOO constitutes an arm selection policy with improved regret bounds compared to previous results for a large class of problems.

Mansley et al. [138] extend HOO into the playout planning structure to give the *Hierarchical Optimistic Optimisation applied to Trees* (HOOT) algorithm. The approach is similar to UCT, except that using HOO for action selection allows the algorithm to overcome the discrete action limitation of UCT.

5.1.5 Other Bandit Enhancements

There are a number of other enhancements to bandit-based methods which have not necessarily been used in an MCTS setting. These include UCB-V, PAC-UCB, Gaussian UCB, Meta-Bandits, Hierarchical Bandits, UCB(α), and so on. See also the bandit-based active learner (4.3.3).

5.2 Selection Enhancements

Many enhancements alter the tree policy to change the way MCTS explores the search tree. Generally, selection assigns some numeric score to each action in order to balance exploration with exploitation, for example the use of UCB for node selection in UCT. In many domains it has proved beneficial to influence the score for each action using domain knowledge, to bias the search towards/away from certain actions and make use of other forms of reward estimate.

5.2.1 First Play Urgency

The MCTS algorithm specifies no way of determining the order in which to visit unexplored nodes. In a typical implementation, UCT visits each unvisited action once in random order before revisiting any using the UCB1 formula. This means that exploitation will rarely occur deeper in the tree for problems with large branching factors.

First play urgency (FPU) is a modification to MCTS proposed by Gelly et al. [95] to address this issue, by assigning a fixed value to score unvisited nodes and using the UCB1 formula to score visited nodes. By tuning this fixed value, early exploitations are encouraged.

5.2.2 Decisive and Anti-Decisive Moves

Teytaud and Teytaud [215] demonstrate the benefit of *decisive* and *anti-decisive* moves for the connection game Havannah. Here, a decisive move is one that leads immediately to a win, and an anti-decisive move is one that prevents the opponent from making a decisive move on their next turn. The selection and simulation policies are replaced with the following policy: if either player has a decisive move then play it; otherwise, revert to the standard policy.

Teytaud and Teytaud [215] show that this modification significantly increases playing strength, even when the increased computational cost of checking for decisive moves is taken into account. This approach is reminiscent of the pre-search handling of winning and losing moves suggested earlier [28].

5.2.3 Move Groups

In some games, it may be the case that the branching factor is large but many moves are similar. In particular, MCTS may need a lot of simulation to differentiate between moves that have a highly correlated expected reward. One way of reducing the branching factor to allow exploitation of correlated actions is to use *move groups*. This creates an extra decision layer in which all possible actions are collected into groups and UCB1 is used to select which of these groups to pick a move from. This idea was proposed in [63] and was shown to be beneficial for the game Go. In addition, the use of transpositions allows information to be shared between these extra nodes where the state is unchanged.

5.2.4 Transpositions

MCTS naturally builds a search tree, but in many cases the underlying games can be represented as *directed acyclic graphs* (DAGs), since similar states can be reached through different sequences of move. The search tree is typically much larger than the DAG and two completely different paths from the root of the tree to a terminal state may traverse the same edge in the game's DAG. Hence extra information can be extracted from each simulation by storing the statistics for each edge in the DAG and looking these up during action selection. Whenever an identical state/action pair appears in the MCTS tree, this is referred to as a *transposition*. The use of transposition statistics can be considered as an enhancement to both the selection and backpropagation steps. Methods for making use of transpositions with MCTS are explored in [63] and further covered in Section 6.2.4.

Transposition tables will have greater benefit for some games than others. Transposition tables were used in conjunction with MCTS for the game Arimaa by Kozlek [122], which led to a measurable improvement in performance. Transpositions were also used in a General Game Playing (GGP) context by Méhat et al. [144], giving an equivalent or better playing strength in all domains tested. Saffidine further explores the benefits of

transposition tables to GGP in his thesis [181]. Saffidine et al. [182] also demonstrate the successful extension of MCTS methods to DAGs for correctly handling transpositions for the simple LeftRight game (7.4).

5.2.5 Progressive Bias

Progressive bias describes a technique for adding domain specific heuristic knowledge to MCTS [60]. When a node has been visited only a few times and its statistics are not reliable, then more accurate information can come from a heuristic value H_i for a node with index i from the current position. A new term is added to the MCTS selection formula of the form:

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

where the node with index i has been visited n_i times. As the number of visits to this node increases, the influence of this number decreases.

One advantage of this idea is that many games already have strong heuristic functions, which can be easily injected into MCTS. Another modification used in [60] and [232] was to wait until a node had been visited a fixed number of times before calculating H_i. This is because some heuristic functions can be slow to compute, so storing the result and limiting the number of nodes that use the heuristic function leads to an increase in the speed of the modified MCTS algorithm.

5.2.6 Opening Books

Opening books[16] have been used to improve playing strength in artificial players for many games. It is possible to combine MCTS with an opening book, by employing the book until an unlisted position is reached. Alternatively, MCTS can be used for generating an opening book, as it is largely domain independent. Strategies for doing this were investigated by Chaslot et al. [56] using their Meta-MCTS approach (4.9.4). Their self-generated opening books improved the playing strength of their program MoGo for 9×9 Go.

Audouard et al. [12] also used MCTS to generate an opening book for Go, using MoGo to develop a revised opening book from an initial handcrafted book. This opening book improved the playing strength of the program and was reported to be consistent with expert Go knowledge in some cases. Kloetzer [115] demonstrates the use of MCTS for generating opening books for the game of Amazons (7.3).

5.2.7 Monte Carlo Paraphrase Generation (MCPG)

Monte Carlo Paraphrase Generation (MCPG) is similar to plain UCT except that the maximum reachable score for each state is used for selection rather than the (average) score expectation for that state [62]. This modification is so named by Chevelu et al. due to its application in generating paraphrases of natural language statements (7.8.5).

16. Databases of opening move sequences of known utility.

5.2.8 Search Seeding

In plain UCT, every node is initialised with zero win and visits. Seeding or "warming up" the search tree involves initialising the statistics at each node according to some heuristic knowledge. This can potentially increase playing strength since the heuristically generated statistics may reduce the need for simulations through that node. The function for initialising nodes can be generated either automatically or manually. It could involve adding virtual win and visits to the counts stored in the tree, in which case the prior estimates would remain permanently. Alternatively, some transient estimate could be used which is blended into the regular value estimate as the node is visited more often, as is the case with RAVE (5.3.5) or Progressive Bias (5.2.5).

For example, Szita et al. seeded the search tree with "virtual wins", to significantly improve the playing strength but required hand-tuning to set the appropriate number of virtual wins for each action. Gelly and Silver [92] investigated several different methods for generating prior data for Go and found that prior data generated by a function approximation improved play the most.

5.2.9 Parameter Tuning

Many MCTS enhancements require the optimisation of some parameter, for example the UCT exploration constant C_p or the RAVE constant V (5.3.5). These values may need adjustment depending on the domain and the enhancements used. They are typically adjusted manually, although some approaches to automated parameter tuning have been attempted.

The exploration constant C_p from the UCT formula is one parameter that varies between domains. For high performance programs for both Go [55] and Hex [8] it has been observed that this constant should be zero (no exploration) when history heuristics such as AMAF and RAVE are used (5.3), while other authors use non-zero values of C_p which vary between domains. There have been some attempts to automatically tune this value online such as those described by Kozelek [122].

Given a large set of enhancement parameters there are several approaches to finding optimal values, or improving hand-tuned values. Guillaume et al. used the Cross-Entropy Method to fine tune parameters for the Go playing program MANGO [58]. Cross Entropy Methods were also used in combination with hand-tuning by Chaslot et al. for their Go program MoGo [55], and neural networks have been used to tune the parameters of MoGo [57], using information about the current search as input. Another approach called *dynamic exploration*, proposed by Bourki et al. [25], tunes parameters based on patterns in their Go program MoGo.

5.2.10 History Heuristic

There have been numerous attempts to improve MCTS using information about moves previously played. The idea is closely related to the *history heuristic* [193], and is described by Kozelek [122] as being used on two levels:

- *Tree-tree level*: Using history information to improve action selection in the MCTS tree.
- *Tree-playout level*: Using history information to improve the simulation policy (6.1).

One approach at the tree-tree level was a *grandfather heuristic* approach suggested by Gelly and Silver [92]. History information was used to initialise the action value estimates for new nodes, but was not as effective as other initialisation methods. Kozelek [122] also used a history-based approach at the tree-tree level for the game Arimaa (7.3). A history bonus was given to the bandit score calculated during action selection and the score for an action was updated whenever it was selected independent of depth, giving a significant improvement.

Finnsson [83] describes the benefits of the history heuristic for seeding node values in his world champion general game player CADIAPLAYER (7.5). See also the use of the history heuristic for improving simulation estimates (6.1.5).

5.2.11 Progressive History

Nijssen and Winands [155] propose the *Progressive History* enhancement, which combines Progressive Bias (5.2.5) with the history heuristic by replacing the heuristic value H_i in the progressive bias calculation for each node i with that node's history score, during node selection. Progressive History was shown to perform well for some multi-player board games (7.3), indicating that it may be useful for multi-player games in general.

5.3 All Moves As First (AMAF)

All Moves As First (AMAF) is an enhancement closely related to the history heuristic, first proposed in the context of Monte Carlo Go. The basic idea is to update statistics for all actions selected during a simulation as if they were the first action applied. The first attempt to combine AMAF with UCT was by Gelly et al. in the context of Go [92], and AMAF heuristics have since proved very successful for Go [94].

Figure 4 shows the AMAF heuristic in action on a simple artificial 3×3 game (from [101]). In this situation, UCT selects the actions C2, A1 for black and white respectively, then the simulation plays black B1, white A3 and black C3 leading to a win for black. When UCT selected C2 as a move for black, UCT could have also selected B1 and C3 as alternatives. Since these moves were used during the simulation, these nodes have their reward/visit count updated by the AMAF algorithm. Similarly, UCT selected the move A1 for white, but could have selected A3 which was used in the simulation, so the AMAF algorithm updates the reward/visit for this node too. Nodes that receive the extra AMAF update during backpropagation are marked *.

The AMAF algorithm treats all moves played during selection and simulation as if they were played on a previous selection step. This means that the reward estimate for an action a from a state s is updated whenever a is encountered during a playout, even if a was not the actual move chosen from s. Some implementations keep track of the reward estimate generated this way, as well as the usual reward estimate used in the UCT algorithm, in which case the reward estimate generated by the AMAF heuristic is referred to as the *AMAF score*. Several AMAF variants are listed below.

5.3.1 Permutation AMAF

This algorithm is the same as AMAF but also updates nodes that can be reached by permutations of moves in the simulation that preserve the eventual state reached [101]. For example, it may be possible to permute the actions played by each player during a simulation and reach an identical terminal position. Therefore there may be other leaf nodes in the tree from which the same terminal position could have been reached by playing the same moves but in a different order. Permutation AMAF would also update these nodes.

5.3.2 α-AMAF

The α-AMAF algorithm blends the standard (UCT) score for each node with the AMAF score [101]. This requires that a separate count of rewards and visits for each type of update be maintained. It is called α-AMAF since the total score for an action is:

$$\alpha A + (1 - \alpha)U$$

where U is the UCT score and A is the AMAF score.

5.3.3 Some-First AMAF

This approach is the same as the standard AMAF algorithm except that the history used to update nodes is truncated after the first m random moves in the simulation stage [101]. If $m = 0$ then only actions selected in the tree are used to update nodes, similarly if m is larger than the number of moves in the simulation, this is equivalent to the AMAF algorithm.

5.3.4 Cutoff AMAF

In Cutoff AMAF, the AMAF algorithm is used to update statistics for the first k simulations, after which only the standard UCT algorithm is used [101]. The purpose of Cutoff AMAF is to warm-up the tree with AMAF data, then use the more accurate UCT data later in the search.

5.3.5 RAVE

Rapid Action Value Estimation (RAVE) is a popular AMAF enhancement in computer Go programs such as MOGO [92]. It is similar to α-AMAF, except that the α value used at each node decreases with each visit. Instead of supplying a fixed α value, a fixed positive integer $V > 0$ is supplied instead. Then the value of α is calculated after n visits as [101]:

$$\max\left\{0, \frac{V - v(n)}{V}\right\}$$

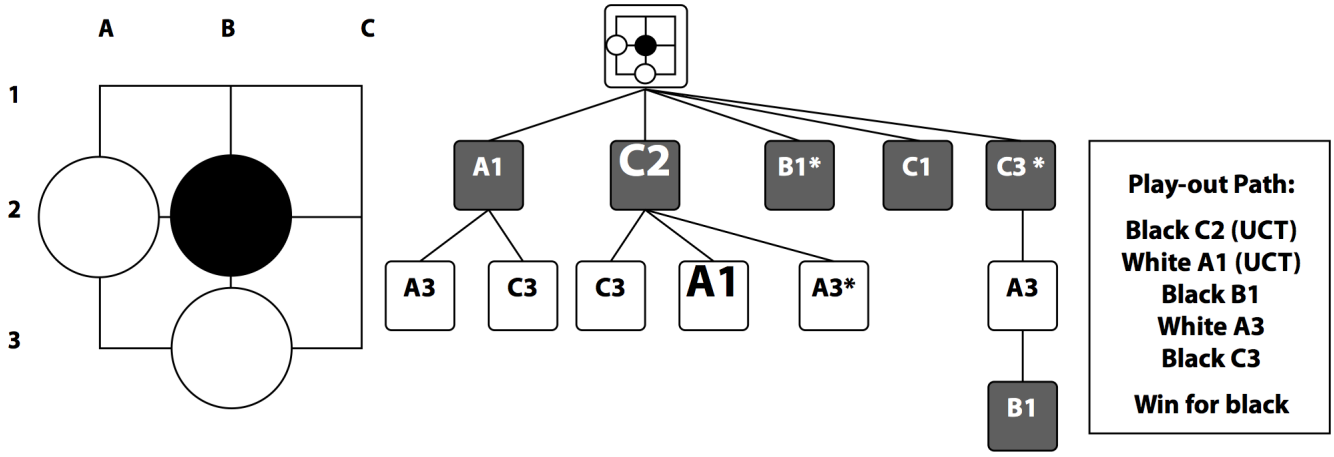


Fig. 4. The All Moves As First (AMAF) heuristic [101].

The parameter V represents the number of visits a node will have before the RAVE values are not being used at all. RAVE is a softer approach than Cutoff AMAF since exploited areas of the tree will use the accurate statistics more than unexploited areas of the tree.

5.3.6 Killer RAVE

Lorentz [133] describes the *Killer RAVE*[17] variant in which only the most important moves are used for the RAVE updates for each iteration. This was found to be more beneficial for the connection game Havannah (7.2) than plain RAVE.

5.3.7 RAVE-max

RAVE-max is an extension intended to make the RAVE heuristic more robust [218], [220]. The RAVE-max update rule and its stochastic variant δ-RAVE-max were found to improve performance in degenerate cases for the Sum of Switches game (7.3) but were less successful for Go.

5.3.8 PoolRAVE

Hoock et al. [104] describe the *poolRAVE* enhancement, which modifies the MCTS simulation step as follows:

- Build a pool of the k best moves according to RAVE.
- Choose one move m from the pool.
- Play m with a probability p, else the default policy.

PoolRAVE has the advantages of being independent of the domain and simple to implement if a RAVE mechanism is already in place. It was found to yield improvements for Havannah and Go programs by Hoock et al. [104] – especially when expert knowledge is small or absent – but not to solve a problem particular to Go known as *semeai*.

Helmbold and Parker-Wood [101] compare the main AMAF variants and conclude that:

- Random playouts provide more evidence about the goodness of moves made earlier in the playout than moves made later.
- AMAF updates are not just a way to quickly initialise counts, they are useful after every playout.
- Updates even more aggressive than AMAF can be even more beneficial.
- Combined heuristics can be more powerful than individual heuristics.

5.4 Game-Theoretic Enhancements

If the game-theoretic value of a state is known, this value may be backed up the tree to improve reward estimates for other non-terminal nodes. This section describes enhancements based on this property.

Figure 5, from [235], shows the backup of proven game-theoretic values during backpropagation. Wins, draws and losses in simulations are assigned rewards of $+1$, 0 and -1 respectively (as usual), but proven wins and losses are assigned rewards of $+\infty$ and $-\infty$.

5.4.1 MCTS-Solver

Proof-number search (PNS) is a standard AI technique for proving game-theoretic values, typically used for endgame solvers, in which terminal states are considered to be proven wins or losses and deductions chained backwards from these [4]. A non-terminal state is a proven win if at least one of its children is a proven win, or a proven loss if all of its children are proven losses. When exploring the game tree, proof-number search prioritises those nodes whose values can be proven by evaluating the fewest children.

Winands et al. [235], [234] propose a modification to MCTS based on PNS in which game-theoretic values[18] are proven and backpropagated up the tree. If the parent node has been visited more than some threshold T times, normal UCB selection applies and a forced loss node is

17. So named due to similarities with the "Killer Move" heuristic in traditional game tree search.

18. That is, known wins, draws or losses.

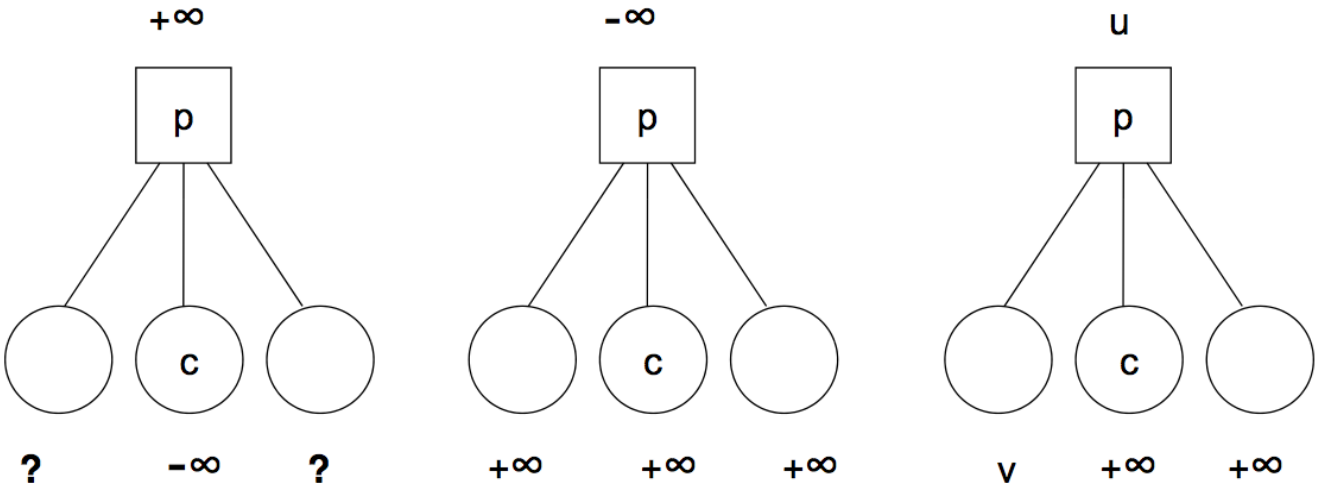


Fig. 5. Backup of proven game-theoretic values [235].

never selected; otherwise, a child is selected according to the simulation policy and a forced loss node may be selected. Nijssen and Winands [155] also describe a multi-player version of their MCTS-Solver (4.5).

5.4.2 Monte Carlo Proof-Number Search (MC-PNS)

Saito et al. [183] introduce *Monte Carlo proof-number search* (MC-PNS), a variant of proof-number search in which nodes that do not immediately prove a game-theoretic value are evaluated by Monte Carlo simulation. Thus MC-PNS uses Monte Carlo evaluations to guide the proof-number search and expand the nodes of the tree in a more efficient order. This allows game-theoretic values to be proven twice as quickly in computer Go experiments [183], with a quarter of the nodes.

5.4.3 Score Bounded MCTS

Cazenave and Saffidine [51] propose an MCTS enhancement for the case of games with multiple outcomes, e.g. a win or a draw, which result in a different score. Each node has a pessimistic and optimistic bound on the score of the node from the point of view of the maximizing player. These bounds converge to the estimated score for a node with more iterations, and a node is considered solved if the two bounds become equal to the score of the node. The two bounds on the score of a node are backpropagated through the tree.

The optimistic and pessimistic bounds can be used to prove nodes from the tree, and also to bias action selection by adding the bounds to the score estimate for a node, multiplied by some constant. MCTS with these enhancements was demonstrated to reduce the number of simulations required to solve *seki* situations in Go (7.1) and was also shown to be beneficial for the game Connect Four (Section 7.3).

5.5 Move Pruning

The *pruning* of suboptimal moves from the search tree is a powerful technique when used with minimax, for example the α-β algorithm yields significant benefits for two-player zero-sum games. Move pruning can be similarly beneficial for MCTS approaches, as eliminating obviously poor choices allows the search to focus more time on the better choices.

An advantage of pruning strategies is that many are domain-independent, making them general improvements for a range of problems. In the absence of a reliable evaluation function, two types of move pruning have been developed for use with MCTS:

- *Soft pruning* of moves that may later be searched and selected, and
- *Hard pruning* of moves that will never be searched or selected.

Soft pruning alleviates the risk that the best move may have been prematurely pruned and removed from consideration. However, some pruning techniques require a reliable evaluation function for states, which is not always available when using MCTS.

5.5.1 Progressive Unpruning/Widening

Progressive unpruning/widening is an example of a heuristic soft pruning technique. Progressive unpruning was proposed by Chaslot et al. [60] and the related idea of progressive widening was proposed by Coulomb [71]. The advantage of this idea over hard pruning is that it exploits heuristic knowledge to immediately reduce the size of the tree, but that all moves will eventually be considered (given enough time). This idea is similar to First Play Urgency (5.2.1) in that it forces earlier exploitation. Teytaud and Teytaud found that progressive widening without heuristic move ordering had little effect on playing strength for the game of Havannah [214]. It was found to give a small improvement in playing strength for the Go program MoGo [128].

Couëtoux et al. describe the extension of UCT to continuous stochastic problems through the use of *double progressive widening* [69], in which child nodes are

either revisited, added or sampled from previously seen children, depending on the number of visits. Double progressive widening worked well for toy problems for which standard UCT failed, but less so for complex real-world problems.

5.5.2 Absolute and Relative Pruning

Absolute pruning and relative pruning are two strategies proposed by Huang [106] to preserve the correctness of the UCB algorithm.

- *Absolute pruning* prunes all actions from a position except the most visited one, once it becomes clear that no other action could become more visited.
- *Relative pruning* uses an upper bound on the number of visits an action has received, to detect when the most visited choice will remain the most visited.

Relative pruning was found to increase the win rate of the Go program LINGO against GNU GO 3.8 by approximately 3% [106].

5.5.3 Pruning with Domain Knowledge

Given knowledge about a domain, it is possible to prune actions known to lead to weaker positions. For example, Huang [106] used the concept of territory in Go to significantly increase the performance of the program LINGO against GNU GO 3.8. Domain knowledge related to predicting opponents' strategies was used by Suoju et al. for move pruning in the game Dead End for a 51.17% improvement over plain UCT [99].

Arneson et al. use domain knowledge to prune *inferior cells* from the search in their world champion Hex program MOHEX [8]. This is computationally expensive to do, so only nodes that had been visited a certain number of times had such domain knowledge applied. An added benefit of this approach is that the analysis would sometimes solve the position to give its true game-theoretic value.

5.6 Expansion Enhancements

No enhancements specific to the expansion step of the tree policy were found in the literature. The particular expansion algorithm used for a problem tends to be more of an implementation choice – typically between single node expansion and full node set expansion – depending on the domain and computational budget.

6 OTHER ENHANCEMENTS

This section describes enhancements to aspects of the core MCTS algorithm other than its tree policy. This includes modifications to the default policy (which are typically domain dependent and involve heuristic knowledge of the problem being modelled) and other more general modifications related to the backpropagation step and parallelisation.

6.1 Simulation Enhancements

The default simulation policy for MCTS is to select randomly amongst the available actions. This has the advantage that it is simple, requires no domain knowledge and repeated trials will most likely cover different areas of the search space, but the games played are not likely to be realistic compared to games played by rational players. A popular class of enhancements makes the simulations more realistic by incorporating domain knowledge into the playouts. This knowledge may be gathered either offline (e.g. from databases of expert games) or online (e.g. through self-play and learning). Drake and Uurtamo describe such biased playouts as *heavy playouts* [77].

6.1.1 Rule-Based Simulation Policy

One approach to improving the simulation policy is to hand-code a domain specific policy. Such rule-based policies should be fast, so as not to unduly impede the simulation process; Silver discusses a number of factors which govern their effectiveness [203].

6.1.2 Contextual Monte Carlo Search

Contextual Monte Carlo Search [104], [167] is an approach to improving simulations that is independent of the domain. It works by combining simulations that reach the same areas of the tree into *tiles* and using statistics from previous simulations to guide the action selection in future simulations. This approach was used to good effect for the game Havannah (7.2), for which each tile described a particular pairing of consecutive moves.

6.1.3 Fill the Board

Fill the Board is an enhancement described in [53], [55] designed to increase simulation diversity for the game of Go. At each step in the simulations, the Fill the Board algorithm picks N random intersections; if any of those intersections and their immediate neighbours are empty then it plays there, else it plays a random legal move. The simulation policy in this case can make use of patterns (6.1.9) and this enhancement fills up board space quickly, so these patterns can be applied earlier in the simulation.

A similar approach to board filling can be used to good effect in games with complementary goals in which exactly one player is guaranteed to win, however the board is filled. Such games include the connection games Hex and Y, as discussed in Sections 6.1.9 and 7.2.

6.1.4 Learning a Simulation Policy

Given a new domain, it is possible to learn a new simulation policy using generic techniques. The relationship between MCTS and TD learning was mentioned in Section 4.3.1; other techniques that learn to adjust the simulation policy by direct consideration of the simulation statistics are listed below.

Move-Average Sampling Technique (MAST) is an

approach first described by Finnsson and Björnsson [84] for the world champion general game playing program CADIAPLAYER [83]. A table is maintained for each action independent of state, in which the average reward $Q(a)$ for each action a is stored and updated during the backpropagation step. Then, during subsequent simulations, these values are used to bias action selection towards more promising moves using a Gibbs distribution. A related technique called *Tree-Only MAST (TO-MAST)*, in which only the actions selected within the search are updated, was also proposed [86].

Predicate-Average Sampling Technique (PAST) is similar to MAST and was proposed in [86]. Each state is represented as a list of predicates that hold true in that state. Then, instead of a table of average values for actions, PAST maintains a table of average values for predicate/action pairs $Q_p(p, a)$. During the backpropagation process, these values are updated for every action selected and every predicate that is true in the state in which that action was selected. As with MAST, simulations select moves according to a Gibbs distribution, here depending on the maximum value of $Q_p(p, a)$ over all predicates p for the current state. MAST biases the simulations towards moves which are good on average, whereas PAST biases the simulations towards moves which are good in a certain context.

Feature-Average Sampling Technique (FAST) is technique related to MAST and PAST and also proposed in [86]. This is designed for use with games specified with the Game Description Language (GDL) used for the AAAI General Game Playing competitions (7.5).

First, features of the game are extracted from the game definition (in this case piece type and board format), then the TD(λ) method is used to learn the relative importance of features, and this is in turn used to calculate the $Q(a)$ values used for a simulation policy. It was found that this technique leads to a big improvement over a random simulation policy, as long as suitable features can be recognised from the game description.

6.1.5 Using History Heuristics

The history heuristic (5.2.10) assumes that a move good in one position may be good in another, to inform action choices during the selection step. A similar approach may also be applied during the simulation step, where it is described as "using history information at the tree-playout level" [122]. MAST (6.1.4) is an example of this approach.

Bouzy [26] experimented with history heuristics for Go. Two versions were tested:

1) an *internal* heuristic that alters moves made during the playouts, and
2) an *external* heuristic that changes the moves selected before the playout.

The external history heuristic led to a significant improvement in playing strength.

Drake and Uurtamo [77] investigated whether search time is better spent improving the tree policy or the simulation policy. Their scenario included using history heuristics for Go and they concluded that it was more efficient to improve the simulation policy.

6.1.6 Evaluation Function

It is possible to use an evaluation function to improve the simulation policy. For example, Winands and Björnsson [232] test several strategies for designing a simulation policy using an evaluation function for the board game Lines of Action (7.2). They found the most successful strategy to be one that initially uses the evaluation function to avoid bad moves, but later in the simulation transitions to greedily selecting the best move.

6.1.7 Simulation Balancing

Silver describes the technique of *simulation balancing* using gradient descent to bias the policy during simulations [203]. While it has been observed that improving the simulation policy does not necessarily lead to strong play [92], Silver and Tesauro demonstrate techniques for learning a simulation policy that works well with MCTS to produce *balanced*[19] if not strong play [203].

6.1.8 Last Good Reply (LGR)

Another approach to improving simulations is the *Last Good Reply* (LGR) enhancement described by Drake [75]. Each move in a game is considered a reply to the previous move, and deemed successful if the player who makes the reply goes on to win. For each move, the last successful reply is stored and used after subsequent occurrences of that move. Since only one reply is stored per move, later replies will overwrite previous ones.

During the simulations, each player will play the last good reply stored if it is legal and otherwise use the default policy. This is referred to as the LGR-1 policy; Drake also defines a variant LGR-2 in [75] which stores replies for the last two moves and uses LGR-1 if there is no LGR-2 entry for the last two moves.

Baier and Drake [17] propose an extension to LGR-1 and LGR-2 called *Last Good Reply with Forgetting* (LGRF). In this context, "forgetting" means removing a stored reply if that reply leads to a loss during the last simulation. Two corresponding enhancements, LGRF-1 and LGRF-2, include forgetting. LGR enhancements were shown to be an improvement over the default policy for 19×19 Go, and storing a reply to the last two moves provided more benefit when forgetting was used.

6.1.9 Patterns

In terms of board games such as Go, a *pattern* is a small non-empty section of the board or a logical test upon it. Patterns may also encode additional information such as the player to move, and are typically incorporated into

19. Games in which errors by one player are on average cancelled out by errors by the opponent on their next move [203].

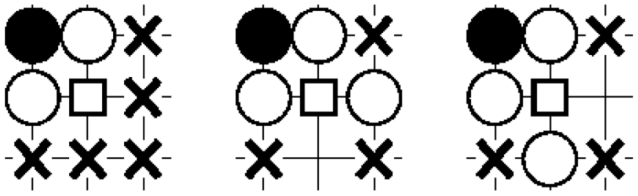


Fig. 6. Patterns for a *cut* move in Go [96].

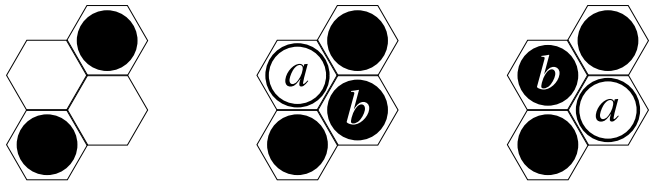


Fig. 7. Bridge completion for connection games.

simulations by detecting pattern matches with the actual board position and applying associated moves.

For example, Figure 6 from [96] shows a set of 3×3 patterns for detecting *cut* moves in Go. The first pattern must be matched and the other two *not* matched for the move to be recognised. Drake and Uurtamo [77] suggest there may be more to gain from applying heuristics such as patterns to the simulation policy rather than the tree policy for Go.

The 3×3 patterns described by Wang and Gelly [228] vary in complexity, and can be used to improve the simulation policy to make simulated games more realistic. Wang and Gelly matched patterns around the last move played to improve the playing strength of their Go program MoGo [228]. Gelly and Silver [92] used a reinforcement learning approach to improve the simulation policy, specifically a function approximator $Q_{\text{RLGO}}(s, a)$, which applied linear weights to a collection of binary features.[20] Several policies using this information were tested and all offered an improvement over a random policy, although a weaker handcrafted policy was stronger when used with UCT.

Coulom [71] searched for useful patterns in Go by computing Elo ratings for patterns, improving their Go program Crazy Stone. Hoock and Teytaud investigate the use of *Bandit-based Genetic Programming* (BGP) to automatically find good patterns that should be more simulated and bad patterns that should be less simulated for their program MoGo, achieving success with 9×9 Go but less so with 19×19 Go [105].

Figure 7 shows a *bridge* pattern that occurs in connection games such as Hex, Y and Havannah (7.2). The two black pieces are *virtually connected* as an intrusion by white in either cell can be answered by black at the other cell to restore the connection. Such intrusions can be detected and completed during simulation to significantly improve playing strength, as this mimics moves that human players would typically perform.

6.2 Backpropagation Enhancements

Modifications to the backpropagation step typically involve special node updates required by other enhancement methods for forward planning, but some constitute enhancements in their own right. We describe those not explicitly covered in previous sections.

20. These features were all 1×1 to 3×3 patterns on a Go board.

6.2.1 Weighting Simulation Results

Xie and Liu [237] observe that some simulations are more important than others. In particular, simulations performed later in the search tend to be more accurate than those performed earlier, and shorter simulations tend to be more accurate than longer ones. In light of this, Xie and Liu propose the introduction of a weighting factor when backpropagating simulation results [237]. Simulations are divided into segments and each assigned a positive integer weight. A simulation with weight w is backpropagated as if it were w simulations.

6.2.2 Score Bonus

In a normal implementation of UCT, the values backpropagated are in the interval $[0, 1]$, and if the scheme only uses 0 for a loss and 1 for a win, then there is no way to distinguish between strong wins and weak wins. One way of introducing this is to backpropagate a value in the interval $[0, \gamma]$ for a loss and $[\gamma, 1]$ for a win with the strongest win scoring 1 and the weakest win scoring γ. This scheme was tested for Sums Of Switches (7.3) but did not improve playing strength [219].

6.2.3 Decaying Reward

Decaying reward is a modification to the backpropagation process in which the reward value is multiplied by some constant $0 < \gamma \leq 1$ between each node in order to weight early wins more heavily than later wins. This was proposed alongside UCT in [119], [120].

6.2.4 Transposition Table Updates

Childs et al. [63] discuss a variety of strategies – labelled *UCT1*, *UCT2* and *UCT3* – for handling transpositions (5.2.4), so that information can be shared between different nodes corresponding to the same state. Each variation showed improvements over its predecessors, although the computational cost of UCT3 was large.

6.3 Parallelisation

The independent nature of each simulation in MCTS means that the algorithm is a good target for parallelisation. Parallelisation has the advantage that more simulations can be performed in a given amount of time and the wide availability of multi-core processors can be exploited. However, parallelisation raises issues such as the combination of results from different sources in a single search tree, and the synchronisation of threads of

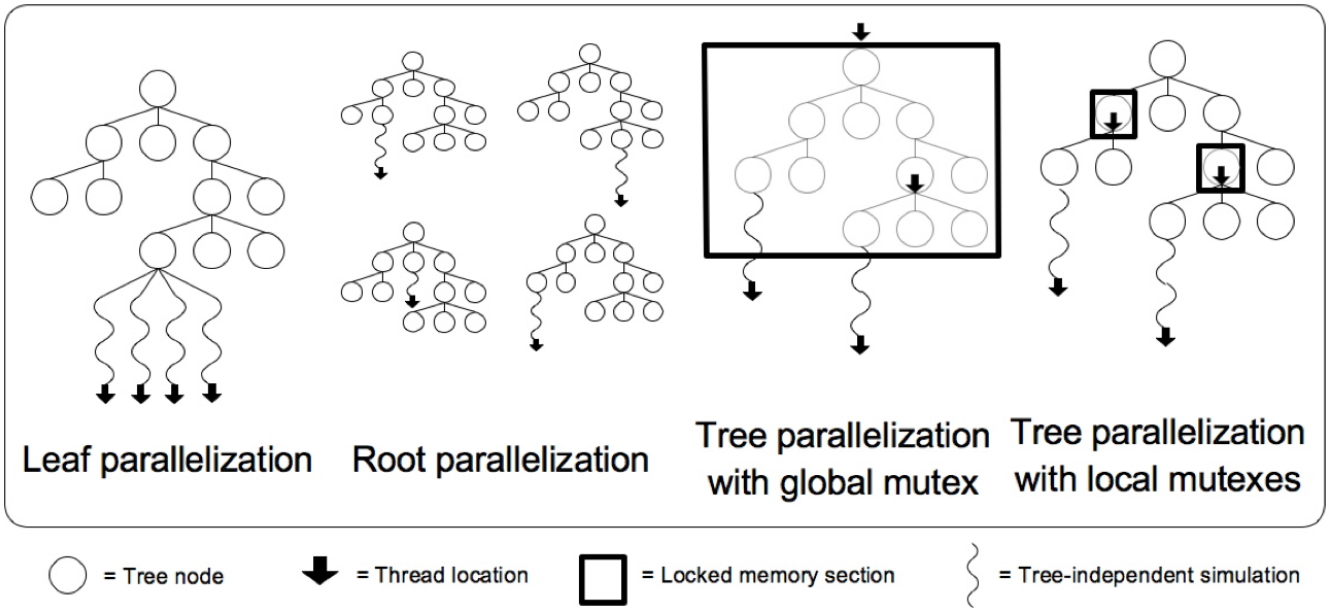


Fig. 8. Parallelisation approaches for MCTS [59].

different speeds over a network. This section describes methods of parallelising MCTS and addressing such issues. Figure 8 shows the main parallelisation approaches for MCTS, as described by Chaslot et al. [59].

6.3.1 Leaf Parallelisation

Leaf parallelisation as defined in [59] involves performing multiple simultaneous simulations every time the MCTS tree policy reaches (or creates) a leaf node. The idea is to collect better statistics at each leaf by achieving a better initial estimate. Cazenave and Jouandeau call this scheme *at-the-leaves parallelisation* [47].

One problem is that the simulations may take differing lengths of time, hence the algorithm is limited to waiting for the longest simulation to finish. Kato and Takeuchi [113] describe how leaf parallelisation can be implemented in a client-server network architecture, with a single client executing the MCTS search and calling upon several servers to perform simulations.

6.3.2 Root Parallelisation

Root parallelisation [59] is sometimes called *multi-tree MCTS* because multiple MCTS search trees are built simultaneously (i.e. parallelised at the root). Usually the information from the first layer in each tree is used to inform the move chosen by the algorithm. One advantage of this approach is that each thread can run for a fixed length of time and stop at any moment. Note that UCT with root parallelisation is not algorithmically equivalent to plain UCT, but is equivalent to Ensemble UCT (4.6.1).

Soejima et al. analyse the performance of root parallelisation in detail [205]. They provide evidence that a majority voting scheme gives better performance than the conventional approach of playing the move with the greatest total number of visits across all trees.

Cazenave and Jouandeau also describe root parallelisation under the name *single-run parallelisation* [47] and a related scheme called *multiple-runs parallelisation* in which the statistics for moves from the root of each tree are periodically shared between processes. Multiple-runs parallelisation is similar to the *slow root parallelisation* of Bourki et al. [24].

6.3.3 Tree Parallelisation

Tree parallelisation is a parallelisation process which involves simultaneous MCTS simulation steps on the same tree [59]. Care must be taken to protect the tree from simultaneous access by different threads; each thread must gain exclusive access to a subtree of the whole search tree and the other threads must explore other areas until the lock is released. One scheme proposed in [59] makes use of a *global lock* (mutex) at the root node. This would be a reasonable approach if the simulations took much longer than traversing the tree, since one thread can traverse or update the tree while others perform simulations. Another scheme uses *local locks* (mutexes) on each internal node, which are locked and unlocked every time a thread visits a node.

One issue with tree parallelisation is that each thread is likely to traverse the tree in mostly the same way as the others. One suggested solution is to assign a temporary "virtual loss" to a node when it is first encountered during action selection [59]. This encourages different threads to select different nodes whilst any nodes that are clearly better than the others will still be preferred. This virtual loss is then removed immediately prior to the backpropagation step to restore the tree statistics.

Bourki et al. suggest a variation called *slow tree parallelisation*, in which statistics are synchronised between

trees periodically and only on parts of the tree[21] [24]. This is better suited to implementation in a message-passing setting, where communication between processes is limited, e.g. when parallelising across clusters of machines. Bourki et al. find that slow tree parallelisation slightly outperforms slow root parallelisation, despite the increased communication overheads of the former. The idea of periodically synchronising statistics between trees is also explored in [91].

6.3.4 UCT-Treesplit

Schaefers and Platzner [192] describe an approach they call *UCT-Treesplit* for performing a single MCTS search efficiently across multiple computer nodes. This allows an equal distribution of both the work and memory load among all computational nodes within distributed memory. Graf et al. [98] demonstrate the application of UCT-Treesplit in their Go program GOMORRA to achieve high-level play. GOMORRA scales up to 16 nodes before diminishing returns reduce the benefit of splitting further, which they attribute to the high number of simulations being computed in parallel.

6.3.5 Threading and Synchronisation

Cazenave and Jouandeau [48] describe a parallel Master-Slave algorithm for MCTS, and demonstrate consistent improvement with increasing parallelisation until 16 slaves are reached.[22] The performance of their 9×9 Go program increases from 40.5% with one slave to 70.5% with 16 slaves against GNU GO 3.6.

Enzenberger and Müller [80] describe an approach to multi-threaded MCTS that requires no locks, despite each thread working on the same tree. The results showed that this approach has much better scaling on multiple threads than a locked approach.

Segal [195] investigates why the parallelisation of MCTS across multiple machines has proven surprisingly difficult. He finds that there is an upper bound on the improvements from additional search in single-threaded scaling for FUEGO, that parallel speedup depends critically on how much time is given to each player, and that MCTS can scale nearly perfectly to at least 64 threads when combined with virtual loss, but without virtual loss scaling is limited to just eight threads.

6.4 Considerations for Using Enhancements

MCTS works well in some domains but not in others. The many enhancements described in this section and the previous one also have different levels of applicability to different domains. This section describes efforts to understand situations in which MCTS and its enhancements may or may not work, and what conditions might cause problems.

21. For example, only on nodes above a certain depth or with more than a certain number of visits.

22. At which point their algorithm is 14 times faster than its sequential counterpart.

6.4.1 Consistency

Heavily modified MCTS algorithms may lead to incorrect or undesirable behaviour as computational power increases. An example of this is a game played between the Go program MOGO and a human professional, in which MOGO incorrectly deduced that it was in a winning position despite its opponent having a winning killer move, because that move matched a number of very bad patterns so was not searched once [19]. Modifying MCTS enhancements to be consistent can avoid such problems without requiring that the entire search tree eventually be visited.

6.4.2 Parameterisation of Game Trees

It has been observed that MCTS is successful for trick-taking card games, but less so for poker-like card games. Long et al. [130] define three measurable parameters of game trees and show that these parameters support this view. These parameters could also feasibly be used to predict the success of MCTS on new games.

6.4.3 Comparing Enhancements

One issue with MCTS enhancements is how to measure their performance consistently. Many enhancements lead to an increase in computational cost which in turn results in fewer simulations per second; there is often a trade-off between using enhancements and performing more simulations.

Suitable metrics for comparing approaches include:

- Win rate against particular opponents.
- Elo[23] ratings against other opponents.
- Number of iterations per second.
- Amount of memory used by the algorithm.

Note that the metric chosen may depend on the reason for using a particular enhancement.

7 APPLICATIONS

Chess has traditionally been the focus of most AI games research and been described as the "drosophila of AI" as it had – until recently – been the standard yardstick for testing and comparing new algorithms [224]. The success of IBM's DEEP BLUE against grandmaster Gary Kasparov has led to a paradigm shift away from computer Chess and towards computer Go. As a domain in which computers are not yet at the level of top human players, Go has become the new benchmark for AI in games [123].

The most popular application of MCTS methods is to games and of these the most popular application is to Go; however, MCTS methods have broader use beyond games. This section summarises the main applications of MCTS methods in the literature, including computer Go, other games, and non-game domains.

23. A method for calculating relative skill levels between players that is widely used for Chess and Go, named after Arpad Elo.

7.1 Go

Go is a traditional board game played on the intersections of a square grid, usually 19×19. Players alternately place stones on the board; orthogonally adjacent stones form *groups*, which are captured if they have no *liberties* (orthogonally adjacent empty spaces). The game ends when both players pass, and is won by the player who controls the most board territory.

Compared with Chess, strong AI methods for Go are a hard problem; computer Go programs using α-β search reached the level of a strong beginner by around 1997, but stagnated after that point until 2006, when the first programs using MCTS were implemented. Since then, progress has been rapid, with the program MoGo beating a professional player on a 9×9 board in 2008 [128] and on a large board (with a large handicap) also in 2008. This success is also summarised in [129]. Today, the top computer Go programs all use MCTS and play at the strength of a good amateur player. Computer Go tournaments are also dominated by MCTS players [127].

7.1.1 Evaluation

There are several obstacles to making strong AI players for Go; games are long (around 200 moves) and have a large branching factor (an average of 250 legal plays per move), which poses a challenge for traditional AI techniques that must expand every node. However, a bigger obstacle for traditional search techniques is the lack of a good static evaluation function for non-terminal nodes [61]. Evaluation functions are problematic for several reasons:

- A piece placed early in the game may have a strong influence later in the game, even if it will eventually be captured [76].
- It can be impossible to determine whether a group will be captured without considering the rest of the board.
- Most positions are dynamic, i.e. there are always unsafe stones on the board [70].

MCTS programs avoid these issues by using random simulations and naturally handling problems with delayed rewards.

7.1.2 Agents

It is indicative of the power of MCTS that over three dozen of the leading Go programs now use the algorithm. Of particular note are:

- MoGo [90] [55], the first Go player to use MCTS and still an innovation in the field. It was the first program to use RAVE (5.3) and sequence-like patterns (6.1.9) and is currently the only top Go program using the Fill the Board technique (6.1.3).
- Crazy Stone [72] was the first Go program using MCTS to win a tournament, and the first to beat a professional player with less than a 9 stone handicap. Crazy Stone uses AMAF with a learned pattern library and other features to improve the default policy and perform progressive widening.
- Leela was the first commercial Go program to embrace MCTS, though also one of the weaker ones.
- Fuego [79] was the first program to beat a professional Go player in an even 9×9 game as white, and uses RAVE.

At the 15th Computer Olympiad, Erica won the 19×19 category using RAVE with progressive bias (5.2.5), a learned 3×3 pattern library [107] and sophisticated time management [108]. Commercial programs MyGoFriend and Many Faces of Go won the 9×9 and 13×13 categories respectively; both use MCTS, but no other details are available. The Fourth UEC Cup was won by Fuego, with MCTS players Zen and Erica in second and third places; Zen uses RAVE and a full-board probabilistic model to guide playouts. Table 2 from [94] shows the relative Elo rankings of the main 9×9 Go programs, both MCTS and non-MCTS. Fuego GB Prototype[24] produced excellent results against human experts for 9×9 Go [148]. While its performance was less successful for 13×13, Müller observes that it still performed at a level that would have been unthinkable a few years ago.

7.1.3 Approaches

Most of the current Go programs use AMAF or RAVE (5.3), allowing the reuse of simulation information. Additionally, it has been observed by several authors that when using AMAF or RAVE, the exploration constant for the UCB formula should be set to zero. CrazyStone and Zen go further in extracting information from playouts, using them to build up a probabilistic score for each cell on the board. Drake [75] suggests using the Last Good Reply heuristic (6.1.8) to inform simulations, modified by Baier and Drake [17] to include the forgetting of bad moves. Most programs use parallelisation, often with lock-free hashtables [80] and message-passing parallelisation for efficient use of clusters (6.3). Silver [201] uses temporal difference learning methods (4.3.1) to extend the MCTS algorithm for superior results in 9×9 Go with MoGo.

Cazenave advocates the use of abstract game knowledge as an alternative to pattern-based heuristics [38]. For example, his *playing atari*[25] heuristic, which modifies move urgency depending on whether the move threatens atari on enemy groups or addresses atari for friendly groups, was found to significantly improve play in his program Golois. Cazenave also encouraged his program to spend more time on earlier and more important moves by stopping the search when each game is clearly decided.

Genetic Programming methods were used by Cazenave to evolve heuristic functions to bias move

24. A variant of Fuego that uses machine-learnt pattern knowledge and an extra additive term in the UCT formula [148].

25. A group of stones under imminent threat of capture is in *atari*.

Year	Program	Description	Elo
2006	INDIGO	Pattern database, Monte Carlo simulation	1400
2006	GNU GO	Pattern database, α-β search	1800
2006	MANY FACES	Pattern database, α-β search	1800
2006	NEUROGO	TDL, neural network	1850
2007	RLGO	TD search	2100
2007	MOGO	MCTS with RAVE	2500
2007	CRAZY STONE	MCTS with RAVE	2500
2008	FUEGO	MCTS with RAVE	2700
2010	MANY FACES	MCTS with RAVE	2700
2010	ZEN	MCTS with RAVE	2700

TABLE 2
Approximate Elo rankings of 9×9 Go programs [94].

choice in the default policy for Go [37]. These heuristic functions were in the form of symbolic expressions, and outperformed UCT with RAVE.

Cazenave [44] also demonstrates how to incorporate *thermography* calculations into UCT to improve playing strength for 9×9 Go. Thermography, in the context of combinatorial game theory, is the study of a game's "temperature" as indicated by the prevalence of either warm (advantageous) moves or cool (disadvantageous) moves. It appears more beneficial to approximate the temperature separately on each game rather than globally over all games.

Huang et al. [110] demonstrate how the clever use of time management policies can lead to significant improvements in 19×19 Go for their program ERICA. Examples of time management policies include the self-explanatory Think Longer When Behind approach and better use of the additional time that becomes available as the opponent ponders their move.

7.1.4 Domain Knowledge

Patterns (6.1.9) are used extensively in Go programs in both search and simulation; Chaslot et al. [55] provide an excellent description of common patterns, tactical and strategic rules. Chaslot et al. [60], Huang et al. [109], Coulom [71] and others all describe methods of learning patterns; Lee et al. [128] show that hand-tuning pattern values is worthwhile. Aduard et al. [12] show that opening books make a big improvement in play level; progressive widening or progressive unpruning (5.5.1) is used to manage the large branching factor, with patterns, tactical, and strategic rules [55] used to determine the move priorities.

Wang et al. [228] and Gelly et al. [96] note that balanced playouts (equal strength for both players) are important and that increasing simulation strength may lead to weaker performance overall, so rules are chosen empirically to improve performance and vary from implementation to implementation. Wang and Gelly [228] describe sequence-like 3×3 patterns which are now used widely to direct playouts, low liberty rules used to ensure sensible play when a group is in danger of being captured, and approximate rules for handling *nakade*[26] and *semeai*.[27]

7.1.5 Variants

MCTS has been applied to the following Go variants.

Random Go Helmstetter et al. [102] describe an experiment where a strong human player played against MOGO (MCTS player) from randomly generated, fair positions. They conclude that randomly generated positions are harder for the human to analyse; with 180 or more random stones on the board, the artificial player becomes competitive with the human.

Phantom Go has imperfect information: each player can see only his own stones. The standard rules of Go apply, but each player reports their move to a referee, who reports back: illegal (stone may not be placed), legal (placement accepted), or a list of captured stones if captures are made. Cazenave [36] applied flat Monte Carlo with AMAF to Phantom Go. Cazenave's program GOLOIS was the strongest Phantom Go program at the 2007 Computer Olympiad [46]. Borsboom et al. [23] found that Cazenave's technique outperforms several techniques based on UCT with determinization (4.8.1).

Blind Go follows the normal rules of Go, except that the human player cannot see the Go board. In contrast to Phantom Go, players have complete knowledge of their opponent's moves, the only source of "imperfect information" being the human player's imperfect memory. Chou et al. [65], pitting blindfold humans against the MCTS-based player MOGOTW on small boards, found that performance drops greatly for beginners, who were not able to complete a blindfold game, but noted only a small drop in play strength by the top players.

NoGo is a variant of Go in which players lose if they capture a group or are forced to suicide, which is equivalent to forbidding all captures and ending the game when there are no legal moves. Chou et al. [64] implemented an artificial player for NoGo and

26. A *nakade* is a dead group that looks alive due to an internal space.

27. A *semeai* is a capturing race.

tested several standard enhancements. They conclude that RAVE and anti-decisive moves (5.2.2) lead to improvements in playing strength, *slow node creation*[28] leads to benefits for situations in which time or memory are the limiting factors, and that adding domain knowledge to playouts was most beneficial.

Multi-player Go is simply Go with more than two players. Cazenave [40] compares several versions of UCT ranging from *paranoid*,[29] to one that actively seeks alliances with the other players. He concludes that in a competition, there is no best algorithm independent of the other competitors.

7.1.6 Future Work on Go

Rimmel et al. [170] identify four types of flaws in the current generation of Go programs:

1) flaws in the opening library,
2) unwillingness to play in corners,
3) over-agressive play, and
4) handling of *semeais* and *sekis* (two groups that cannot be captured, but are not absolutely alive).

Option (1) at least is an easy avenue for improvement.

Takeuchi et al. [210], [211] use the relationship between the win probability obtained from playouts with actual games to calculate evaluation curves, which allow the comparison of different search methods, search parameters, and search performance at different stages of the game. These measurements promise to improve performance in Go and other MCTS applications.

Silver et al. [202] describe *Dyna-2*, a learning system with permanent and dynamic values with parallels to RAVE, which can beat standard UCT. Sylvester et al. [208] built a neural network that is stronger than standard UCT and found that a simple linear classifier was stronger still. Marcolino and Matsubara suggest that the next step in computer Go might be emergent behaviour [139].

7.2 Connection Games

Connection games are games in which players strive to complete a specified type of connection with their pieces, be it connecting two or more goal regions, forming a loop, or gathering pieces into connected sets. The strongest known connection game agents at competition board sizes are currently all MCTS implementations.

Hex is the quintessential connection game, in which players strive to connect the opposite sides of a hexagonally tessellated rhombus marked with their colour with a chain of their pieces. Hex has the feature that exactly one player must win (since one player winning explicitly blocks the other from doing so), hence simulations may be performed until the board is full and the win test applied only once, for efficiency. This is similar to the Fill the Board policy used to improve simulations in Go (6.1.3).

Raiko [161] first demonstrated the use of UCT for Hex in 2008, using domain knowledge in the form of *bridge completion* (6.1.9) during playouts. The resulting player was unranked and performed best on smaller boards, but also performed equally well on other hexagonally based connection games without modification.

Arneson et al. [8] developed MoHex, which uses UCT in conjunction with RAVE and domain knowledge in the form of inferior cell analysis to prune the search tree, and bridge completion during simulations. MoHex has won the 14th and 15th Computer Olympiads to become the reigning Computer Hex world champion [7]. Other MCTS Hex players that competed include MIMHex and Yopt [50], [180].

Y, *Star and Renkula! Y is the most fundamental of connection games, in which players share the same goal of connecting the three sides of a hexagonally tessellated triangle with a chain of their pieces. *Star is one of the more complex connection games, which is played on a hexagonally tiled hexagon and involves outer cell and group scores. Renkula! is a 3D connection game played on the sphere which only exists virtually. Raiko's UCT connection game agent [161] plays all of these three games and is the strongest known computer player at all board sizes.

Havannah is a connection race game with more complex rules, played on a hexagonal board tessellated by hexagons. A player wins by completing with their pieces:

1) a *bridge* connecting any two corners,
2) a *fork* connecting any three sides, and/or
3) a closed *loop* around any cells.

The complexity of these multiple winning conditions, in addition to the large standard board of side length 10 (271 cells), makes it difficult to program an effective agent and perhaps even more difficult than Go [214]. In terms of number of MCTS enhancements tested upon it, Havannah is arguably second only to Go (see Table 3).

Könnecke and Waldmann implemented a UCT Havannah player with AMAF and a playout horizon [121], concentrating on efficient implementation but not finding any other reliable computer opponent to test the playing strength of their agent. Teytaud and Teytaud [214] then implemented another UCT player for Havannah and demonstrated that some lessons learnt from UCT for computer Go also apply in this context while some do not. Specifically, the RAVE heuristic improved playing strength while progressive widening did not. Teytaud and Teytaud [215] further demonstrate the benefit of decisive and anti-decisive moves (5.2.2) to improve playing

28. A technique in which a node is not created unless its parent has already been created and it has been simulated a certain number of times.

29. The paranoid player assumes that all other players will make the moves that are most harmful towards it.

strength.

Rimmel et al. [169] describe a general method for biasing UCT search using RAVE values and demonstrate its success for both Havannah and Go. Rimmel and Teytaud [167] and Hook et al. [104] demonstrate the benefit of Contextual Monte Carlo Search (6.1.2) for Havannah.

Lorentz [133] compared five MCTS techniques for his Havannah player WANDERER and reports near-perfect play on smaller boards (size 4) and good play on medium boards (up to size 7). A computer Havannah tournament was conducted in 2010 as part of the 15th Computer Olympiad [134]. Four of the five entries were MCTS-based; the entry based on α-β search came last.

Stankiewicz [206] improved the performance of his MCTS Havannah player to give a win rate of 77.5% over unenhanced versions of itself by biasing move selection towards key moves during the selection step, and combining the Last Good Reply heuristic (6.1.8) with *N-grams*[30] during the simulation step.

Lines of Action is a different kind of connection game, played on a square 8×8 grid, in which players strive to form their pieces into a single connected group (counting diagonals). Winands et al. [236] have used Lines of Action as a test bed for various MCTS variants and enhancements, including:

- The MCTS-Solver approach (5.4.1) which is able to prove the game-theoretic values of positions given sufficient time [234].
- The use of positional evaluation functions with Monte Carlo simulations [232].
- Monte Carlo α-β (4.8.7), which uses a selective two-ply α-β search at each playout step [233].

They report significant improvements in performance over straight UCT, and their program MC-LOA$_{\alpha\beta}$ is the strongest known computer player for Lines of Action.

7.3 Other Combinatorial Games

Combinatorial games are zero-sum games with discrete, finite moves, perfect information and no chance element, typically involving two players (2.2.1). This section summarises applications of MCTS to combinatorial games other than Go and connection games.

P-Game A *P-game* tree is a minimax tree intended to model games in which the winner is decided by a global evaluation of the final board position, using some counting method [119]. Accordingly, rewards are only associated with transitions to terminal states. Examples of such games include Go, Othello, Amazons and Clobber.

Kocsis and Szepesvári experimentally tested the performance of UCT in random P-game trees and found empirically that the convergence rates of UCT is of order $B^{D/2}$, similar to that of α-β search for the trees investigated [119]. Moreover, Kocsis et al. observed that the convergence is not impaired significantly when transposition tables with realistic sizes are used [120].

Childs et al. use P-game trees to explore two enhancements to the UCT algorithm: treating the search tree as a graph using transpositions and grouping moves to reduce the branching factor [63]. Both enhancements yield promising results.

Clobber is played on an 8×8 square grid, on which players take turns moving one of their pieces to an adjacent cell to capture an enemy piece. The game is won by the last player to move. Kocsis et al. compared flat Monte Carlo and plain UCT Clobber players against the current world champion program MILA [120]. While the flat Monte Carlo player was consistently beaten by MILA, their UCT player won 44.5% of games, averaging 80,000 playouts per second over 30 seconds per move.

Othello is played on an 8×8 square grid, on which players take turns placing a piece of their colour to *flip* one or more enemy pieces by capping lines at both ends. Othello, like Go, is a game of delayed rewards; the board state is quite dynamic and expert players can find it difficult to determine who will win a game until the last few moves. This potentially makes Othello less suited to traditional search and more amenable to Monte Carlo methods based on complete playouts, but it should be pointed out that the strongest Othello programs were already stronger than the best human players even before MCTS methods were applied.

Nijssen [152] developed a UCT player for Othello called MONTHELLO and compared its performance against standard α-β players. MONTHELLO played a non-random but weak game using straight UCT and was significantly improved by preprocessed move ordering, both before and during playouts. MONTHELLO achieved a reasonable level of play but could not compete against human experts or other strong AI players.

Hingston and Masek [103] describe an Othello player that uses straight UCT, but with playouts guided by a weighted distribution of rewards for board positions, determined using an evolutionary strategy. The resulting agent played a competent game but could only win occasionally against the stronger established agents using traditional hand-tuned search techniques.

Osaki et al. [157] apply their TDMC(λ) algorithm (4.3.2) to Othello, and report superior performance over standard TD learning methods. Robles et al. [172] also employed TD methods to automatically integrate domain-specific knowledge into MCTS, by learning a linear function approximator to bias move selection in the algorithm's default policy. The resulting program demonstrated improvements over a plain UCT player but was again weaker than established agents for Othello

30. Markovian sequences of words (or in this case moves) that predict the next action.

using α-β search.

Takeuchi et al. [210], [211] compare the win probabilities obtained for various search methods, including UCT, to those observed in actual games, to evaluate the effectiveness of each search method for Othello. Othello remains an open challenge for future MCTS research.

Amazons is one of the more interesting combinatorial games to emerge in recent years, remarkable for its large move complexity, having on average over 1,000 move combinations to choose from each turn. It is played on a 10×10 square grid, on which players take turns moving one of their *amazons* as per a Chess queen, then shooting an *arrow* from that piece along any unobstructed line (orthogonal or diagonal) to block the furthest cell. The number of playable cells thus shrinks with each turn, and the last player to move wins. Amazons has an obvious similarity to Go due to the importance of territory and connectivity.

Kocsis et al. demonstrated the superiority of plain UCT over flat Monte Carlo for Amazons [120]. Similarly, Lorentz found that flat Monte Carlo performed poorly against earlier α-β players in their Amazons players INVADER and INVADERMC [132]. The inclusion of UCT into INVADERMC elevated its playing strength to defeat all previous versions and all other known Amazon agents. Forward pruning and progressive widening (5.5.1) are used to focus the UCT search on key moves.

Kloetzer has studied MCTS approaches to Amazons [116], [114] culminating in a PhD thesis on the topic [115]. This includes MCTS approaches to endgame analysis [117], [118] and more recently the generation of opening books [115].

Arimaa is a Chess-like game designed in 1997 to defeat traditional AI analysis through its huge move space complexity; its branching factor averages between 17,000 to 50,000 move combinations per turn.

Kozelek [122] describes the implementation of a UCT player for Arimaa. The basic player using straight UCT played a weak game, which was improved significantly using a technique described as the *tree-tree history heuristic* (5.2.10), parallelisation, and information sharing across the tree through transpositions (6.2.4). Implementing heavy playouts that incorporate tactical information and positional information from move advisers was also beneficial, but standard MCTS enhancements such as UCB tuning and RAVE were not found to work for this game. This was probably due to Arimaa's explosive combinatorial complexity requiring an infeasible number of simulations before significant learning could occur.

Kozelek [122] found it preferable to handle each component sub-move as an individual action in the UCT tree, rather than entire move combinations. This reduces the search space complexity of such games with compound moves to a reasonable level, at the expense of strategic coherence within and between moves.

Khet is played on an 8×10 square board, on which players place and move pieces with mirrors on some sides. At the end of each turn, the mover activates a laser and captures enemy pieces that the reflected beam encounters, and wins by capturing the enemy *pharaoh*. The average branching factor is 69 moves and the average game length is 68 moves, giving an average game tree complexity of around 10^{25} (similar to Checkers).

Nijssen [153], [154] developed an MCTS Khet player using straight UCT with transposition tables but no other enhancements. Random playouts were found to take too long on average (many taking over 1,000 turns), so playouts were capped at a certain length and the game declared a draw at that point. The straight UCT player did not win a single game against their earlier α-β player.

Shogi is a Chess-like game most popular in Japan, in which captured pieces may be dropped back into play under the capturer's control during a standard move. Sato et al. [186] describe a UCT Shogi player with a number of enhancements: history heuristic, progressive widening, killer moves, checkmate testing and the use of heavy playouts based on Elo rankings of move features as proposed for Go by Coulom [71]. Sato et al. found that UCT without enhancement performed poorly for Shogi, but that their enhanced UCT player competed at the level of a strong amateur. However, even their enhanced program fared poorly against state of the art Shogi agents using traditional search techniques. These have now reached a high level of play due to the popularity of Shogi and it is unlikely that MCTS approaches will supersede them without significant research effort.

Takeuchi et al. [210], [211] compare the win probabilities obtained for various search methods, including UCT, to those observed in actual games, to investigate the effectiveness of each method for Shogi.

Mancala is one of the oldest families of traditional combinatorial games. It is typically played on two lines of six holes from which stones are picked up and sown around subsequent holes on each turn, according to the rules for the variant being played.

Ramanujan and Selman [165] implemented a UCT player for Mancala and found it to be the first known game for which minimax search and UCT both perform at a high level with minimal enhancement. It was shown that in this context, if the computational budget is fixed, then it is far better to run more UCT iterations with fewer playouts per leaf than to run fewer iterations with more playouts. Ramanujan and Selman also demonstrate the benefit of a hybrid UCT/minimax approach if some heuristic knowledge of the domain is available. Their work on comparing the performance of UCT with minimax in various search spaces (3.5) is

continued elsewhere [164].

Blokus Duo is played on a 14×14 square grid with 21 polyominoes of size 3, 4 and 5 belonging to each player. Players take turns adding a piece to the board to touch at least one existing friendly *at the corners only*, and the game is won by the player to place the largest total piece area.

Shibahara and Kotani [200] describe an MCTS player for Blokus Duo using plain UCT without enhancement, as the game is relatively new, hence it is difficult to reliably evaluate non-terminal board positions given the lack heuristic knowledge about it. Their program uses a sigmoid function to combine the search score and winning percentage in its search results, which was found to make more moves that they describe as "human" and "amusing" when losing. The program placed seventh out of 16 entries in a Computer Blokus Duo contest held in Japan.

Focus (also called Domination) is played on an 8×8 square board with truncated corners by two to four players. Players start with a number of pieces on the board, which they may stack, move and split, in order to force their opponent(s) into a position with no legal moves[31]. Nijssen and Winands [155] applied their Multi-Player Monte-Carlo Tree Search Solver (4.5) and Progressive History (5.2.11) techniques to Focus to significantly improve playing strength against a standard MCTS player.

Chinese Checkers is a traditional game played on a star-shaped board by two to six players. Players aim to move their pieces from their home area to a target area on the opposite side of the board through a series of steps and jumps over adjacent pieces.

Nijssen and Winands [155] also applied their Multi-Player Monte-Carlo Tree Search Solver (MP-MCTS-Solver) and Progressive History techniques to Chinese Checkers, but found that only Progressive History significantly improved playing strength against a standard MCTS player. The failure of the MP-MCTS-Solver enhancement in this case may be due to the fact that Chinese Checkers is a sudden-death game while Focus is not. In any event, Progressive History appears to be a useful enhancement for multi-player games.

Yavalath is played on a hexagonally tessellated hexagon of size 5, on which players strive to make 4-in-a-row of their colour without making 3-in-a-row beforehand. It is the first computer-designed board game to be commercially released. A plain UCT player with no enhancements beyond pre-search handling of winning and losing moves (similar to decisive and anti-decisive moves [215]) played a competent game [28].

31. A simplified winning condition was used in the experiments to speed up the self-play trials.

Connect Four is a well known children's game played on a 7x6 square grid, in which players drop pieces down to make four in a row of their colour. Cazenave and Saffidine demonstrated the benefit of α-β-style cuts in solving the game for smaller boards using a Score Bounded MCTS (5.4.3) approach [51].

Tic Tac Toe is a convenient test bed for MCTS algorithms due to its simplicity and small search space, but is rarely used as a benchmark for this very reason. One exception is Veness et al. who describe the application of ρUCT (4.10.6) in their MC-AIXA agent for Tic Tac Toe and a number of other simple games [226]. Auger describes the application of MCTS methods to the partially observable case of Phantom Tic Tac Toe [16].

Sum of Switches (SOS) is an artificial number picking game played by two players, designed to represent the best-case scenario for history heuristics such as RAVE (5.3.5) for experimental purposes [219], [218], [220]. A problem with the RAVE heuristic is that it can accumulate strong bias against correct moves when some moves are very good if played early, but very bad if played later in a simulation. This is a problem that does not happen in SOS. Tom and Müller [219] indicate that UCT performance can be improved through careful tuning of the RAVE parameters to suit the situation, rather than necessarily focussing on parallelisation and ever greater numbers of playouts. Their extension RAVE-max (5.3.7) was found to improve RAVE performance for degenerate cases in SOS [220].

Chess and Draughts Ramanujan et al. [163] describe pathologies in behaviour that result from UCT Chess players carefully constructed to explore synthetic search spaces. Surprisingly, however, there are no human-competitive MCTS implementations reported in the literature for either Chess or Draughts, probably the western world's two most well known and widely played board games. Existing agents for these games may simply be too strong to invite competition or allow meaningful comparisons.

The commercial Chess program RYBKA provides a Monte Carlo feature to help players analyse positions [131]. It is unclear exactly what "Monte Carlo" entails in this instance, but this feature can provide an alternative interpretation of degenerate board positions that confuse even strong Chess programs.

The relatively poor performance of UCT for Chess compared to other games may also be due to the occurrence of trap states (3.5) [162]. Takeuchi et al. [210], [211] compare the win probabilities obtained for various search methods, including UCT, to those observed in actual games, to investigate the effectiveness of each search method for Chess.

Gomoku is typically played with Go pieces on a

Go board, although 15×15 is also a common board size. Players take turns adding a piece of their colour and win by making 5-in-a-row orthogonally or diagonally.

Gomoku is popular (especially as a recreation among Go players), simple to program, and makes an excellent test case for UCT; it is a very good game for quickly checking that a UCT implementation is working, and its similarity to Go makes it an obvious stepping stone towards a full Go program. Gomoku was an early UCT test case for several of this paper's authors, and is likely to have been an early test case for others as well. However, there is little mention of Gomoku in the literature and no specific Gomoku programs are described, possibly because the game has been solved up to at least 15×15.

7.4 Single-Player Games

Single-player (solitaire or puzzle) games are a special case of combinatorial game in which the solver competes against the null player or puzzle setter. This section describes the use of MCTS methods to solve various types of logic puzzles.

Leftmost Path and Left Move Problems The *Leftmost Path* and *Left Move* problems [42] are simple artificial games designed to test the nested Monte Carlo search algorithm (4.9.2). The Leftmost Path Problem involves constructing a binary tree and scoring the number of moves on the leftmost part of the tree, hence leaf scores are extremely correlated with the structure of the search tree. This game is called LeftRight in [182], where it is used to demonstrate the successful extension of MCTS methods to DAGs for correctly handling transpositions (5.2.4). In the Left Move Problem the score of a leaf is the number of moves to the left that have been made during a game, hence leaf scores are less correlated with tree structure and NMCS is less informed.

Morpion Solitaire is an NP-hard solitaire puzzle, in which the player successively colours a vertex of an undirected graph, such that a line containing five coloured vertices can be drawn. The aim is to make as many moves as possible. Figure 9 from [42] shows the standard board configuration. There are *touching* and *non-touching* versions of the puzzle, in which two moves in the same direction that share a circle at the end of a line are either legal or non-legal respectively.

Cazenave applied a Reflexive Monte Carlo Search (4.9.2) to solve the non-touching puzzle in 78 moves, beating the existing human record of 68 moves and AI record of 74 moves using simulated annealing [39]. Cazenave then applied nested Monte Carlo search (NMCS) (4.9.2) to find an improved solution of 80 moves [42]. The parallelisation of this problem technique is discussed in further detail in [49].

Akiyama et al. incorporated the AMAF heuristic (5.3) into NMCS to find a new world record solution of 146 moves for the touching version of the puzzle after about 36 days of computation [3]. This record was for computer-generated solutions, since a human generated solution of 170 is known. Edelkamp et al. achieved a score of 128 using UCT with a number of enhancements in their *heuristically guided swarm tree search* [78] and reproduced the score of 170 when the search was seeded with 111 moves. Rosin [176] applied a Nested Rollout Policy Adaptation approach (4.9.3) to achieve a new record of 177 for touching Morpion Solitaire. This is the first automated method to improve upon the human-generated record that had stood for over 30 years.

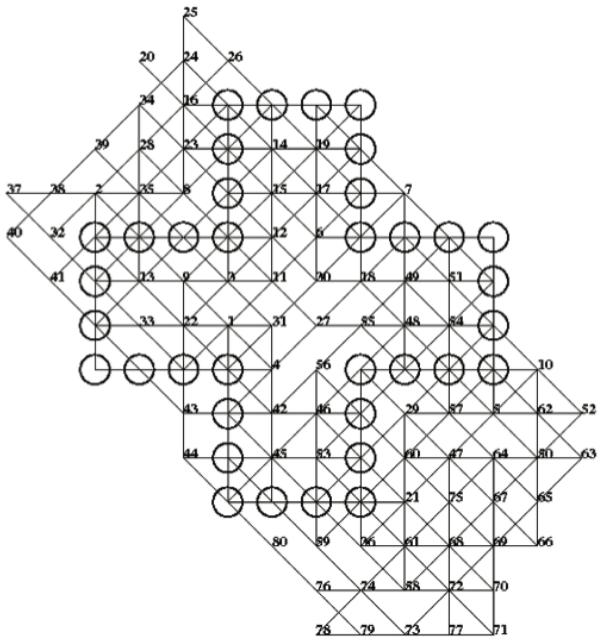


Fig. 9. 80 move Morpion Solitaire solution [42].

Crossword Construction The construction of crosswords is technically a single-player game, though played from the designer's view rather than that player's; the goal is to devise the most amusing and challenging puzzles. Rosin's Nested Rollout Policy Adaptation (NRPA) approach (4.9.3) was also applied to crossword construction, seeking to use as many words as possible per puzzle [176].

SameGame, also called Bubble Breaker, is a logic puzzle game played on a 15×15 square grid which is initially coloured at random in five shades. At each turn, the player selects a coloured group of at least two orthogonally adjacent cells of the same colour, these are removed and the remaining cells collapse down to fill the gap. The game ends if the player fails to clear all cells on a given level, i.e. if some singleton groups remain. The average game length is estimated to be 64.4 moves and the average branching factor 20.7 moves, resulting in a game-tree complexity of 10^{85} and

state-space complexity of 10^{159} [191].

Schadd et al. describe the Single-Player MCTS (SP-MCTS) variant (4.4) featuring modified backpropagation, parameter tuning and meta-search extension, and apply it to SameGame [191] [190]. Their player achieved a higher score than any previous AI player (73,998). Cazenave then applied Nested Monte Carlo Search (4.9.2) to achieve an even higher score of 77,934 [42].

Matsumoto et al. later applied SP-MCTS with domain knowledge to bias move choices during playouts, for superior performance with little impact on computational time [140]. Edelkamp et al. [78] achieved a score of 82,604 using enhanced UCT in their *heuristically guided swarm tree search* (4.9.5).

Sudoku and Kakuro Sudoku, the popular logic puzzle, needs no introduction except perhaps to point out that it is NP-complete for arbitrarily large boards. Cazenave [42] applied nested Monte Carlo search (4.9.2) to 16×16 Sudoku as the standard 9×9 puzzle proved too easy for comparison purposes and reported solution rates over 300,000 times faster than existing Forward Checking methods and almost 50 times faster than existing Iterative Sampling approaches.

Kakuro, also known as Cross Sums, is a similar logic puzzle in the same class as Sudoku that is also NP-complete. Cazenave [41] applied nested Monte Carlo search (4.9.2) to 8×8 Kakaru puzzles for solution rates over 5,000 times faster than existing Forward Checking and Iterative Sampling approaches.

Wumpus World Asmuth and Littman [9] apply their Bayesian FSSS (BFS3) technique to the classic 4x4 video game Wumpus World [178]. Their BFS3 player clearly outperformed a variance-based reward bonus strategy, approaching Bayes-optimality as the program's computational budget was increased.

Mazes, Tigers and Grids Veness et al. [226] describe the application of ρUCT (4.10.6) in their MC-AIXA agent to a range of puzzle games including:

- maze games,
- *Tiger* games in which the player must select the door that maximises some reward, and
- a 4×4 *grid world* game in which the player moves and teleports to maximise their score.

Veness et al. [226] also describe the application of ρUCT to a number of nondeterministic games, which are summarised in a following section.

7.5 General Game Playing

General Game Players (GGPs) are software agents intended to play a range of games well rather than any single game expertly. Such systems move more of the mental work from the human to the machine: while the programmer may fine-tune a dedicated single-game agent to a high level of performance based on their knowledge of the game, GGPs must find good solutions to a range of previously unseen problems. This is more in keeping with the original aims of AI research to produce truly intelligent automata that can perform well when faced with complex real-world problems.

GGP is another arena that MCTS-based agents have dominated since their introduction several years ago. The use of random simulations to estimate move values is well suited to this domain, where heuristic knowledge is not available for each given game.

CADIAPLAYER was the first MCTS-based GGP player, developed by Hilmar Finnsson for his Masters Thesis in 2007 [83]. The original incarnation of CADIAPLAYER used a form of history heuristic and parallelisation to improve performance, but otherwise used no enhancements such as heavy playouts. Finnsson and Björnsson point out the suitability of UCT for GGP as random simulations implicitly capture, in real-time, game properties that would be difficult to explicitly learn and express in a heuristic evaluation function [84]. They demonstrate the clear superiority of their UCT approach over flat MC. CADIAPLAYER went on to win the 2007 and 2008 AAAI GGP competitions [21].

Finnsson and Björnsson added a number of enhancements for CADIAPLAYER, including the Move-Average Sampling Technique (MAST; 6.1.4), Tree-Only MAST (TO-MAST; 6.1.4), Predicate-Average Sampling Technique (PAST; 6.1.4) and RAVE (5.3.5), and found that each improved performance for some games, but no combination proved generally superior [85]. Shortly afterwards, they added the Features-to-Action Sampling Technique (FAST; 6.1.4), in an attempt to identify common board game features using template matching [86]. CADIAPLAYER did not win the 2009 or 2010 AAAI GGP competitions, but a general increase in playing strength was noted as the program was developed over these years [87].

ARY is another MCTS-based GGP player, which uses nested Monte Carlo search (4.9.2) and transposition tables (5.2.4 and 6.2.4), in conjunction with UCT, to select moves [144]. Early development of ARY is summarised in [142] and [143]. ARY came third in the 2007 AAAI GGP competition [143] and won the 2009 [145] and 2010 competitions to become world champion. Méhat and Cazenave demonstrate the benefits of tree parallelisation (6.3.3) for GPP, for which playouts can be slow as games must typically be interpreted [146].

Other GGPs Möller et al. [147] describe their programme CENTURIO which combines MCTS with *Answer Set Programming* (ASP) to play general games. CENTURIO came fourth in the 2009 AAAI GGP competition.

Sharma et al. [197] describe domain-independent methods for generating and evolving domain-specific knowledge using both state and move patterns, to improve convergence rates for UCT in general games

and improve performance against a plain UCT player. They then extended this approach using Reinforcement Learning and Ant Colony Algorithms, resulting in huge improvements in AI player ability [198]. Mahlmann et al. [135] use an MCTS agent for testing and evaluating games described in their Strategy Game Description Game Language (SGDL).

7.6 Real-time Games

MCTS has been applied to a diverse range of real-time games of varying complexity, ranging from Tron and Ms. Pac-Man to a variety of real-time strategy games akin to Starcraft. The greatest challenge facing MCTS approaches is to achieve the same level of intelligence and realistic behaviour achieved by standard methods of scripting, triggers and animations.

Tron Samothrakis et al. [184] present an initial investigation into the suitability of UCT for Tron. They apply a standard implementation of MCTS to Tron: the only two game-specific modifications include the prevention of self-entrapment during the random simulation phase (1-ply look-ahead) and the distinction of a "survival mode" (once the players are physically separated), where the game essentially turns into a single-player game (a simple game tree is used here instead). They compare different MCTS variants, using UCB1, UCB-Tuned (5.1.1) and UCB-E (a modification of UCB1 due to Coquelin and Munos [68]). Samothrakis et al. find that MCTS works reasonably well but that a large proportion of the random playouts produce meaningless outcomes due to ineffective play.

Den Teuling [74] applies several enhancements to plain UCT for Tron, including progressive bias (5.2.5), MCTS-Solver (5.4.1), a game-specific mechanism for handling simultaneous moves (4.8.10), and game-specific simulation policies and heuristics for predicting the outcome of the game without running a complete simulation. These enhancements in various combinations increase the playing strength in certain situations, but their effectiveness is highly dependent on the layout of the board.

Ms. Pac-Man Numerous tree-search and Monte Carlo sampling approaches have been proposed in the past to tackle the game of Ms. Pac-Man. For example, Robles and Lucas [171] expand a route-tree based on possible moves that Ms. Pac-Man can take,[32] and a flat Monte Carlo approach for the endgame strategy was proposed by Tong and Sung [222] and Tong et al. [221], based on path generation and path testing components. The latter is carried out by means of Monte Carlo simulations, making some basic assumptions regarding the movement of Ms. Pac-Man and the four ghosts. This strategy, which may be used in conjunction with other algorithms such as minimax or MCTS, improved the agent's score by 20%.

Samothrakis et al. [185] used MCTS with a 5-player $\max^n$ game tree, in which each ghost is treated as an individual player. Unlike traditional tree searches, MCTS's anytime nature lends itself nicely to the real-time constraints of the game. Knowledge about the opponent is clearly beneficial in this case, as it allows not only for a smaller tree but also much more accurate simulations in the forward projection.

Another application of MCTS to Ms. Pac-Man is due to Ikehata and Ito [111], who use MCTS to avoid pincer moves (i.e. moves where Ms. Pac-Man is trapped by ghosts covering all exits). Nguyen et al. [151] also describe the use of MCTS for move planning in Ms. Pac-Man. In a follow-up paper [112], they extend their MCTS agent to use heuristics learned from game-play, such as the most dangerous places in the maze. Their improved agent won the Ms. Pac-Man screen-capture competition at IEEE CIG 2011, beating the previous best winner of the competition by a significant margin.

Pocman and Battleship Silver and Veness [204] apply a POMDP (2.1.2) approach to Pocman (partially observable Pac-Man) and the classic children's game Battleship. Their players perform on a par with full-width planning methods, but require orders of magnitude less computation time and are applicable to much larger problem instances; performance is far superior to that of flat Monte Carlo. Veness et al. [226] describe the application of ρUCT (4.10.6) in their MC-AIXA agent for partially observable Pac-Man.

Dead-End is a real-time predator/prey game whose participants are a cat (the player) and two dogs. The aim of the cat is to reach the exit of the board, starting from the bottom of the stage. On the other hand, the aim of the dogs is to catch the cat or to prevent it from reaching the exit within a period of time.

He et al. [100] use UCT for the behaviour of the dogs in their artificial player. Their results show how the performance is better when the simulation time is higher and that UCT outperforms the flat Monte Carlo approach. The same authors [99] used a more complex approach based on a KNN classifier that predicts the strategy of the player, to prune the search space in a knowledge-based UCT (KB-UCT). Results show that the pruned UCT outperforms the UCT that has no access to player strategy information.

Yang et al. [239] and Fu et al. [88] used MCTS methods to improve the performance of their joint ANN-based Dead End player. Zhang et al. [240] deal with the problem of Dynamic Difficulty Adjustment (DDA) using a time-constrained UCT. The results show the importance of the length of simulation time for UCT. The performance obtained is seriously affected by this parameter, and it is used to obtain different difficulty levels for the game.

32. The best path was subsequently evaluated using hand-coded heuristics.

Real-time Strategy (RTS) Games Numerous studies have been published that evaluate the performance of MCTS on different variants of real-time strategy games. These games are usually modelled on well-known and commercially successful games such as Warcraft, Starcraft or Command & Conquer, but have been simplified to reduce the number of available actions at any moment in time (the branching factor of the decision trees in such games may be unlimited).

Initial work made use of Monte Carlo simulations as a replacement for evaluation functions; the simulations were embedded in other algorithms such as minimax, or were used with a 1-ply look-ahead and made use of numerous abstractions to make the search feasible given the time constraints.

Wargus Balla and Fern [18] apply UCT to a RTS game called Wargus. Here the emphasis is on tactical assault planning and making use of numerous abstractions, most notably the grouping of individual units. The authors conclude that MCTS is a promising approach: despite the lack of domain-specific knowledge, the algorithm outperformed baseline and human players across 12 scenarios.

ORTS Naveed et al. [150] apply UCT and RRTs (4.10.3) to the RTS game engine ORTS. Both algorithms are used to find paths in the game and the authors conclude that UCT finds solutions with less search effort than RRT, although the RRT player outperforms the UCT player in terms of overall playing strength.

7.7 Nondeterministic Games

Nondeterministic games have hidden information and/or a random element. Hidden information may arise through cards or tiles visible to the player, but not the opponent(s). Randomness may arise through the shuffling of a deck of cards or the rolling of dice. Hidden information and randomness generally make game trees much harder to search, greatly increasing both their branching factor and depth.

The most common approach to dealing with this increase in branching factor is to use determinization, which involves sampling over the perfect information game instances that arise when it is assumed that all hidden and random outcomes are known in advance (see Section 4.8.1).

Skat is a trick-taking card game with a bidding phase. Schafer describes the UCT player XSKAT which uses information sets to handle the nondeterministic aspect of the game, and various optimisations in the default policy for both bidding and playing [194]. XSKAT outperformed flat Monte Carlo players and was competitive with the best artificial Skat players that use traditional search techniques. A discussion of the methods used for opponent modelling is given in [35].

Poker Monte Carlo approaches have also been used for the popular gambling card game Poker [177]. The poker game tree is too large to compute Nash strategies precisely, so states must be collected in a small number of *buckets*. Monte Carlo methods such as Monte Carlo Counter Factual Regret (MCCFR) [125] (4.8.8) are then able to find approximate Nash equilibria. These approaches represent the current state of the art in computer Poker.

Maîtrepierre et al. [137] use UCB to select strategies, resulting in global play that takes the opponent's strategy into account and results in unpredictable behaviour. Van den Broeck et al. apply MCTS methods to multi-player no-limit Texas Hold'em Poker [223], enabling strong exploitative behaviour against weaker rule-based opponents and competitive performance against experienced human opponents.

Ponsen et al. [159] apply UCT to Poker, using a learned opponent model (Section 4.8.9) to bias the choice of determinizations. Modelling the specific opponent by examining games they have played previously results in a large increase in playing strength compared to UCT with no opponent model. Veness et al. [226] describe the application of ρUCT (4.10.6) to Kuhn Poker using their MC-AIXA agent.

Dou Di Zhu is a popular Chinese card game with hidden information. Whitehouse et al. [230] use *information sets* of states to store rollout statistics, in order to collect simulation statistics for sets of game states that are indistinguishable from a player's point of view. One surprising conclusion is that overcoming the problems of strategy fusion (by using expectimax rather than a determinization approach) is more beneficial than having a perfect opponent model.

Other card games such as Hearts and Spades are also interesting to investigate in this area, although work to date has only applied MCTS to their perfect information versions [207].

Klondike Solitaire is a well known single-player card game, which can be thought of as a single-player stochastic game: instead of the values of the hidden cards being fixed at the start of the game, they are determined by chance events at the moment the cards are turned over.[33]

Bjarnason et al. [20] apply a combination of the determinization technique of hindsight optimisation (HOP) with UCT to Klondike solitaire (Section 4.8.1). This system achieves a win rate more than twice that estimated for a human player.

Magic: The Gathering is a top-selling two-player

33. This idea can generally be used to transform a single-player game of imperfect information into one of perfect information with stochasticity.

card game. Ward and Cowling [229] show bandit-based approaches using random rollouts to be competitive with sophisticated rule-based players. The rules of Magic: The Gathering are to a great extent defined by the cards in play, so the creation of strong techniques for Magic: The Gathering can be seen as an exercise in, or at least a stepping stone towards, GGP (7.5).

Phantom Chess[34] is a Chess variant played on three chessboards – one for each player and one for the referee – that incorporates the notion of "fog of war" as players can only see their own pieces while the opponent's pieces are in the dark.

Ciancarini and Favini developed an MCTS-based Phantom Chess player [66], [67] based on previous studies of Phantom Go (7.1.5). They tried different models from the player's and referee's perspectives, based on the partial information available to them, and used probabilities based on experience to influence moves during the playouts to simulate realistic behaviour, for unexpectedly good results.

Urban Rivals is a free internet game played by more than 10,000,000 registered users. Teytaud and Flory [217] observe links between hidden information and simultaneous moves (4.8.10), in order to extend MCTS methods to this class of games and implement a UCT player for Urban Rivals. They find that UCT with EXP3 (5.1.3) outperforms plain UCT and UCT with greedy enhancements for this game.

Backgammon The best current Backgammon agents use reinforcement learning on millions of offline games to learn positional evaluations, and are stronger than the best human players. The UCT-based player MCGAMMON developed by Van Lishout et al. [225] only implemented a simplification of the game, but was found to correctly choose expert moves in some cases, despite making unfortunate choices in others. MCGAMMON achieved around 6,500 playouts per second and based its initial move on 200,000 playouts.

Settlers of Catan is a nondeterministic multi-player game that has won several major game design awards, and was the first "eurogame" to become widely popular outside Germany. Szita et al. [209] implemented a multi-player MCTS player (4.5) for Settlers of Catan, using domain knowledge based on players' resources and current position to bias move selection. Their program performed well against an existing artificial player, JSETTLERS, achieving victory in 49% of games and still achieving good scores in games that it lost. While the agent made generally competent moves against human players, it was found that expert human players could confidently beat it.

34. Phantom Chess is sometimes called *Kriegspiel*, but should not be confused with the board game *Kriegsspiel* to which it bears little resemblance.

Scotland Yard is a turn-based video game with imperfect information and fixed coalitions. Nijssen and Winands describe the application of MCTS to Scotland Yard using a *coalition reduction* method (4.5.1) to outperform a commercial program for this game [156].

Roshambo is a child's game more commonly known as Rock, Paper, Scissors. Veness et al. [226] describe the application of ρUCT (4.10.6) to biased Roshambo using their MC-AIXA agent.

Thurn and Taxis is a German board game in the "eurogame" style for two or more players, with imperfect information and nondeterministic elements, including cards and virtual assistants. Schadd [188] implemented an MCTS player for Thurn and Taxis that incorporated domain knowledge into the playout policy to improve performance (slightly) over a flat UCB implementation.

OnTop is a non-deterministic board game for two to four players. Briesemeister [27] compared an MCTS OnTop player against a number of Minimax, Expectimax and flat Monte Carlo variants, and found that the MCTS implementation won 80% of games.

7.8 Non-Game Applications

This section lists known examples of the application of MCTS methods to domains other than games. These domains include combinatorial optimisation, scheduling tasks, sample based planning, and procedural content generation. Other non-game MCTS applications are known,[35] but have not yet been published in the literature, so are not listed here.

7.8.1 Combinatorial Optimisation

This sections lists applications of MCTS to combinatorial optimisation problems found in the literature.

Security Tanabe [212] propose MCTS methods to evaluate the vulnerability to attacks in an image-based authentication system. The results obtained are promising and suggest a future development of an MCTS based algorithm to evaluate the security strength of the image-based authentication systems.

Mixed Integer Programming In the study performed by Sabharwal and Samulowitz [179], UCT is applied to guide Mixed Integer Programming (MIP), comparing the performance of the UCT based node selection with that of CPLEX, a traditional MIP solver, and best-first, breadth-first and depth-first strategies, showing very promising results.

35. Including, for example, financial forecasting for the stock market and power plant management.

Travelling Salesman Problem The *Travelling Salesman Problem* (TSP) is addressed in [168] using a nested Monte Carlo search algorithm (4.9.2) with *time windows*. State of the art solutions were reached up to 29 nodes, although performance on larger problems is less impressive.

The *Canadian Traveller Problem* (CTP) is a variation of the TSP in which some of the edges may be blocked with given probability. Bnaya et al. [22] propose several new policies and demonstrate the application of UCT to the CTP, achieving near-optimal results for some graphs.

Sailing Domain Kocsis and Szepesvári [119] apply UCT to the *sailing domain*, which is a *stochastic shortest path* (SSP) problem that describes a sailboat searching for the shortest path between two points under fluctuating wind conditions. It was found that UCT scales better for increasing problem size than other techniques tried, including asynchronous real-time dynamic programming (ARTDP) and a Markov decision process called PG-ID based on online sampling.

Physics Simulations Mansely et al. apply the Hierarchical Optimistic Optimisation applied to Trees (HOOT) algorithm (5.1.4) to a number of physics problems [138]. These include the Double Integrator, Inverted Pendulum and Bicycle problems, for which they demonstrate the general superiority of HOOT over plain UCT.

Function Approximation Coquelin and Munos [68] compare their BAST approach (4.2) with flat UCB for the approximation of Lipschitz functions, and observe that BAST outperforms flat UCB and is less dependent on the size of the search tree. BAST returns a good value quickly, and improves towards the optimal value as the computational budget is increased.

Rimmel et al. [166] apply the MCTS-based Threshold Ascent for Graphs (TAG) method (4.10.2) to the problem of automatic performance tuning using DFT and FFT linear transforms in adaptive libraries. They demonstrate superior performance of TAG over standard optimisation methods.

7.8.2 Constraint Satisfaction

This sections lists applications of MCTS methods to constraint satisfaction problems.

Constraint Problems Satomi et al. [187] proposed a real-time algorithm based on UCT to solve a *quantified constraint satisfaction problems* (QCSP).[36] Plain UCT did not solve their problems more efficiently than random selections, so Satomi et al. added a constraint propagation technique that allows the tree to focus in the most favourable parts of the search space. This combined algorithm outperforms the results obtained by state of the art α-β search algorithms for large-scale problems [187].

Previti et al. [160] investigate UCT approaches to the satisfiability of *conjunctive normal form* (CNF) problems. They find that their UCTSAT class of algorithms do not perform well if the domain being modelled has no underlying structure, but can perform very well if the information gathered on one iteration can successfully be applied on successive visits to the same node.

Mathematical Expressions Cazenave [43] applied his nested Monte Carlo search method (4.9.2) to the generation of expression trees for the solution of mathematical problems. He achieved better results than existing methods for the *Prime generating polynomials* problem[37] and a finite algebra problem called the A_2 primal algebra, for which a particular discriminator term must be found.

7.8.3 Scheduling Problems

Planning is also a domain in which Monte Carlo tree based techniques are often utilised, as described below.

Benchmarks Nakhost and Müller apply their Monte Carlo Random Walk (MRW) planner (4.10.7) to all of the supported domains from the 4th International Planning Competition (IPC-4) [149]. MRW shows promising results compared to the other planners tested, including FF, Marvin, YASHP and SG-Plan.

Pellier et al. [158] combined UCT with heuristic search in their Mean-based Heuristic Search for anytime Planning (MHSP) method (4.10.8) to produce an anytime planner that provides partial plans before building a solution. The algorithm was tested on different classical benchmarks (Blocks World, Towers of Hanoi, Ferry and Gripper problems) and compared to some major planning algorithms (A*, IPP, SatPlan, SG Plan-5 and FDP). MHSP performed almost as well as classical algorithms on the problems tried, with some pros and cons. For example, MHSP is better than A* on the Ferry and Gripper problems but worse on Blocks World and the Towers of Hanoi.

Printer Scheduling Matsumoto et al. [140] applied Single Player Monte Carlo Tree Search (4.4) to the game *Bubble Breaker* (7.4). Based on the good results obtained in this study, where the heuristics employed improved the quality of the solutions, the application of this technique is proposed for a re-entrant scheduling problem, trying to manage the printing process of the auto-mobile parts supplier problem.

Rock-Sample Problem Silver et al. [204] apply MCTS and UCT to the rock-sample problem (which simulates a Mars explorer robot that has to analyse and collect

36. A QCSP is a constraint satisfaction problem in which some variables are universally quantified.

37. Finding a polynomial that generates as many different primes in a row as possible.

rocks) and two games: Battleship and Pocman (a partially observable variation of Pac-Man), showing a high degree of performance in all cases.

Production Management Problems (PMPs) can be defined as planning problems that require a parameter optimisation process. Chaslot et al. propose the use of an MCTS algorithm to solve PMPs, getting results faster than Evolutionary Planning Heuristics (EPH), reaching at least the same score in small problems and outperforming EPH in large scenarios [54].

Double progressive widening (5.5.1) has been shown to work well for energy stock management and other toy problems, outperforming plain UCT with progressive widening and Q-learning methods [69], but did not work so well for complex real-world problems.

Bus Regulation The *bus regulation problem* is the task of scheduling bus waiting times so as to minimise delays for passengers [45]. Nested Monte Carlo search with memorisation (4.9.2) was found to clearly outperform the other methods tested.

7.8.4 Sample-Based Planning

Planners for many complex structured domains can be learned with tractable sample complexity if near optimal policies are known.

Large State Spaces Walsh et al. [227] apply Forward Search Sparse Sampling (FSSS) to domains with large state spaces (4.10.1), where neither its sample nor computational efficiency is made intractable by the exponential number of states. They describe a negative case for UCT's runtime that can require exponential computation to optimise, in support of their approach.

Feature Selection To test their Feature UCT Selection (FUSE) algorithm (4.4.1), Gaudel and Sebag [89] use three benchmark data sets from the NIPS 2003 FS Challenge competition in feature selection. The *Arcene* data set contains 10,000 features, which Gaudel and Sebag reduce to 2000 for tractability; the *Madelon* and *Colon* sets contain 500 and 2,000 features respectively. FUSE is found to achieve state of the art performance on these data sets.

7.8.5 Procedural Content Generation (PCG)

Browne describes ways in which MCTS methods may be extended to *procedural content generation* (PCG) for creative domains, such as game design, linguistics, and generative art and music [30]. An important difference from the standard approach is that each search attempts to produce not a single optimal decision but rather a range of good solutions according to the target domain, for which variety and originality can be as important as quality. The fact that MCTS has an inherent restart mechanism (3.3.1) and inherently performs a local iterated search at each decision step makes it a promising approach for PCG tasks.

Chevelu et al. propose the *Monte Carlo Paraphrase Generation* (MCPG) modification to UCT (5.2.7) intended for natural language processing (NLP) tasks such as the paraphrasing of natural language statements [62].

Mahlmann et al. describe the use of UCT for content creation in a strategy game [136]. This algorithm performs battle simulations as the fitness function of an evolutionary strategy, in order to fine tune the game unit types and their parameters. Again, the aim is not to produce the strongest AI player but to generate a satisfactory range of digital in-game content.

8 Summary

The previous sections have provided a snapshot of published work on MCTS to date. In this section, we briefly reflect on key trends and possible future directions for MCTS research. Tables 3 and 4 summarise the many variations and enhancements of MCTS and the domains to which they have been applied, divided into combinatorial games (Table 3) and other domains (Table 4).

The tables show us that UCT is by far the most widely used MCTS technique, and that Go is by far the domain for which most enhancements have been tried, followed by Havannah and General Game Playing. MCTS enhancements are generally applied to combinatorial games, while MCTS variations are generally applied to other domain types.

8.1 Impact

MCTS has had a remarkable impact in the five years since researchers first used Monte Carlo simulation as a method for heuristically growing an interesting part of the search tree for a game. Generally speaking, MCTS appears to work for games and decision problems when:

- We can characterise the problem of making a good decision as a search problem on a large directed graph or tree (e.g. a state-action graph).
- We can sample decisions by conducting random simulations, much faster than real-time. These simulations are (weakly) correlated with the true (expected) value of a given decision state.

A good deal of the MCTS research has focussed on computer Go, spurred on by the success of MCTS players against human professionals on small boards in recent years. This success is remarkable, since human-competitive computer Go was perceived by the AI community as an intractable problem until just a few years ago – or at least a problem whose solution was some decades away.

In the past, there have been two primary techniques for decision-making in adversarial games: minimax α-β search and knowledge-based approaches. MCTS provides an effective third way, particularly for games in which it is difficult to evaluate intermediate game states

or to capture rules in sufficient detail. Hybridisation of MCTS with traditional approaches provides a rich area for future research, which we will discuss further below.

This survey has demonstrated the power of MCTS across a wide range of game domains, in many cases providing the strongest computer players to date. While minimax search has proven to be an effective technique for games where it is possible to evaluate intermediate game states, e.g. Chess and Checkers, MCTS does not require such intermediate evaluation and has proven to be a more robust and general search approach. Its success in such a wide range of games, and particularly in General Game Playing, demonstrates its potential across a broad range of decision problems. Success in non-game applications further emphasises its potential.

8.2 Strengths

Using MCTS, effective game play can be obtained with no knowledge of a game beyond its rules. This survey demonstrates that this is true for a wide range of games, and particularly for General Game Playing, where rules are not known in advance. With further enhancement to the tree or simulation policy, very strong play is achievable. Thus enhanced, MCTS has proven effective in domains of high complexity that are otherwise opaque to traditional AI approaches.

Enhancements may result from incorporating human knowledge, machine learning or other heuristic approaches. One of the great advantages of MCTS is that even when the information given by an enhancement is noisy or occasionally misleading, the MCTS sampling approach is often robust enough to handle this noise and produce stronger play. This is in contrast with minimax search, where the search is brittle with respect to noise in the evaluation function for intermediate states, and this is especially true for games with delayed rewards.

Another advantage of MCTS is that the forward sampling approach is, in some ways, similar to the method employed by human game players, as the algorithm will focus on more promising lines of play while occasionally checking apparently weaker options. This is especially true for new games, such as those encountered in the AAAI General Game Playing competitions, for which no strategic or heuristic knowledge exists. This "humanistic" nature of MCTS makes it easier to explain to the general public than search paradigms that operate very differently to the way in which humans search.

MCTS is often effective for small numbers of simulations, for which mistakes often appear plausible to human observers. Hence the approach is genuinely an "anytime" approach, producing results of plausibility that grows with increasing CPU time, through growing the tree asymmetrically.

8.3 Weaknesses

Combining the precision of tree search with the generality of random sampling in MCTS has provided stronger decision-making in a wide range of games. However, there are clear challenges for domains where the branching factor and depth of the graph to be searched makes naive application of MCTS, or indeed any other search algorithm, infeasible. This is particularly the case for video game and real-time control applications, where a systematic way to incorporate knowledge is required in order to restrict the subtree to be searched.

Another issue arises when simulations are very CPU-intensive and MCTS must learn from relatively few samples. Work on Bridge and Scrabble shows the potential of very shallow searches in this case, but it remains an open question as to whether MCTS is the best way to direct simulations when relatively few can be carried out.

Although basic implementations of MCTS provide effective play for some domains, results can be weak if the basic algorithm is not enhanced. This survey presents the wide range of enhancements considered in the short time to date. There is currently no better way than a manual, empirical study of the effect of enhancements to obtain acceptable performance in a particular domain.

A primary weakness of MCTS, shared by most search heuristics, is that the dynamics of search are not yet fully understood, and the impact of decisions concerning parameter settings and enhancements to basic algorithms are hard to predict. Work to date shows promise, with basic MCTS algorithms proving tractable to "in the limit" analysis. The simplicity of the approach, and effectiveness of the tools of probability theory in analysis of MCTS, show promise that in the future we might have a better theoretical understanding of the performance of MCTS, given a realistic number of iterations.

A problem for any fast-growing research community is the need to unify definitions, methods and terminology across a wide research field. We hope that this paper may go some way towards such unification.

8.4 Research Directions

Future research in MCTS will likely be directed towards:

- Improving MCTS performance in general.
- Improving MCTS performance in specific domains.
- Understanding the behaviour of MCTS.

MCTS is the most promising research direction to date in achieving human-competitive play for Go and other games which have proved intractable for minimax and other search approaches. It seems likely that there will continue to be substantial effort on game-specific enhancements to MCTS for Go and other games.

8.4.1 General-Purpose Enhancements

Many of the enhancements that have emerged through the study of Go have proven applicable across a wide range of other games and decision problems. The empirical exploration of general-purpose enhancements to the MCTS algorithm will likely remain a major area of investigation. This is particularly important for MCTS, as the approach appears to be remarkably general-purpose

and robust across a range of domains. Indeed, MCTS may be considered as a high-level "meta-" technique, which has the potential to be used in conjunction with other techniques to produce good decision agents. Many of the papers surveyed here use MCTS in conjunction with other algorithmic ideas to produce strong results. If we compare other powerful "meta-" approaches such as metaheuristics and evolutionary algorithms, we can see that there is the potential for MCTS to grow into a much larger field of research in the future, capable of solving a very wide range of problems.

8.4.2 MCTS Search Dynamics

Alongside the application-led study of possible enhancements, there are many questions about the dynamics of MCTS search. The theoretical and empirical work here shows promise and is ripe for further study, for example in comparing MCTS, minimax, A* and other search approaches on an empirical and theoretical level, and for understanding the impact of parameters and effective ways for (adaptively) finding suitable values. A related area is the idea of automatic methods for pruning subtrees based on their probability of containing game states that will actually be reached. While UCB1 has made the biggest impact as a bandit algorithm to date, the investigation of other bandit algorithms is an interesting area, for example when the branching factor and depth of the tree is very large, or when the goal of search is to find a mixed strategy, which yields a strategy giving a probability to each of several possible decisions at each decision point.

8.4.3 Hybridisation

The flexibility of MCTS allows it to be hybridised with a range of other techniques, particularly minimax search, heuristic evaluation of intermediate game states and knowledge-based approaches. Hybridisation may allow problems that were intractable for search-based approaches to be effectively handled, in areas such as video games and real-time control. Work to date on MCTS for video games and other complex environments has focussed on easily-modelled decisions or games with fairly simple representation of state, where performing simulation playouts is straightforward. Work is needed on encoding state and incorporating human knowledge or adaptive learning approaches to create a tractable (state, action) graph in more general, complex environments.

The integration of MCTS with knowledge capture, and with data mining and other machine learning methods for automatically capturing knowledge, provides a ripe area for investigation, with the potential to provide a way forward for difficult problems in video gaming and real-time control where large volumes of data are available, e.g. from network play between human players.

8.4.4 Dealing with Uncertainty and Hidden Information

Games with hidden information and stochastic elements often have intractably wide game trees for standard tree search approaches. Here MCTS has shown that it can create approximate Nash players that represent best-possible play, or at least play that is impossible for an opponent to exploit. The integration of MCTS with game-theoretic tools and with opponent-modelling approaches is a promising research direction, due to the importance of hidden information in video games and in practical problems in systems and economic modelling. We may use the tools of data mining and opponent modelling to infer both the hidden information and the strategy of an opponent. The tools of game theory may also prove to be effective for analysis of MCTS in this case. The need for mixed strategies in this case requires a rethink of the basic exploration/exploitation paradigm of MCTS.

8.4.5 Non-Game Applications

MCTS shows great promise in non-game applications, in areas such as procedural content generation (indeed a recent issue of this journal was devoted to this topic) as well as planning, scheduling, optimisation and a range of other decision domains. For example, the introduction of an adversarial opponent provides an ability to work with "worst-case" scenarios, which may open up a new range of problems which can be solved using MCTS in safety critical and security applications, and in applications where simulation rather than optimisation is the most effective decision support tool.

9 Conclusion

MCTS has become the pre-eminent approach for many challenging games, and its application to a broader range of domains has also been demonstrated. In this paper we present by far the most comprehensive survey of MCTS methods to date, describing the basics of the algorithm, major variations and enhancements, and a representative set of problems to which it has been applied. We identify promising avenues for future research and cite almost 250 articles, the majority published within the last five years, at a rate of almost one paper per week.

Over the next five to ten years, MCTS is likely to become more widely used for all kinds of challenging AI problems. We expect it to be extensively hybridised with other search and optimisation algorithms and become a tool of choice for many researchers. In addition to providing more robust and scalable algorithms, this will provide further insights into the nature of search and optimisation in difficult domains, and into how intelligent behaviour can arise from simple statistical processes.

Acknowledgments

Thanks to the anonymous reviewers for their helpful suggestions. This work was funded by EPSRC grants

EP/I001964/1, EP/H048588/1 and EP/H049061/1, as part of the collaborative research project *UCT for Games and Beyond* being undertaken by Imperial College, London, and the Universities of Essex and Bradford.

REFERENCES

Entries marked with an asterisk * denote support material that does not directly cite MCTS methods.

[1] * B. Abramson, "Expected-Outcome: A General Model of Static Evaluation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, pp. 182 – 193, 1990.

[2] * R. Agrawal, "Sample mean based index policies with zero (log n) regret for the multi-armed bandit problem," *Adv. Appl. Prob.*, vol. 27, no. 4, pp. 1054–1078, 1995.

[3] H. Akiyama, K. Komiya, and Y. Kotani, "Nested Monte-Carlo Search with AMAF Heuristic," in *Proc. Int. Conf. Tech. Applicat. Artif. Intell.*, Hsinchu, Taiwan, Nov. 2010, pp. 172–176.

[4] * L. V. Allis, M. van der Meulen, and H. J. van den Herik, "Proof-Number Search," *Artif. Intell.*, vol. 66, no. 1, pp. 91–124, 1994.

[5] * I. Althöfer, "On the Laziness of Monte-Carlo Game Tree Search in Non-tight Situations," Friedrich-Schiller Univ., Jena, Tech. Rep., 2008.

[6] * ——, "Game Self-Play with Pure Monte-Carlo: The Basin Structure," Friedrich-Schiller Univ., Jena, Tech. Rep., 2010.

[7] B. Arneson, R. B. Hayward, and P. Henderson, "MoHex Wins Hex Tournament," *Int. Comp. Games Assoc. J.*, vol. 32, no. 2, pp. 114–116, 2009.

[8] ——, "Monte Carlo Tree Search in Hex," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 251–258, 2010.

[9] J. Asmuth and M. L. Littman, "Approaching Bayes-optimalilty using Monte-Carlo tree search," in *Proc. 21st Int. Conf. Automat. Plan. Sched.*, Freiburg, Germany, 2011.

[10] ——, "Learning is planning: near Bayes-optimal reinforcement learning via Monte-Carlo tree search," in *Proc. Conf. Uncert. Artif. Intell.*, Barcelona, Spain, 2011, pp. 19–26.

[11] * J.-Y. Audibert and S. Bubeck, "Minimax policies for adversarial and stochastic bandits," in *Proc. 22nd Annu. Conf. Learn. Theory*, Montreal, Canada, 2009, pp. 773–818.

[12] P. Audouard, G. M. J.-B. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud, "Grid coevolution for adaptive simulations; application to the building of opening books in the game of Go," in *Proc. Evol. Games*, Tübingen, Germany, 2009, pp. 323–332.

[13] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Mach. Learn.*, vol. 47, no. 2, pp. 235–256, 2002.

[14] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "Gambling in a rigged casino: The adversarial multi-armed bandit problem," in *Proc. Annu. Symp. Found. Comput. Sci.*, Milwaukee, Wisconsin, 1995, pp. 322–331.

[15] A. Auger and O. Teytaud, "Continuous Lunches are Free Plus the Design of Optimal Optimization Algorithms," *Algorithmica*, vol. 57, no. 1, pp. 121–146, 2010.

[16] D. Auger, "Multiple Tree for Partially Observable Monte-Carlo Tree Search," in *Proc. Evol. Games.*, Torino, Italy, 2011, pp. 53–62.

[17] H. Baier and P. D. Drake, "The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 303–309, 2010.

[18] R.-K. Balla and A. Fern, "UCT for Tactical Assault Planning in Real-Time Strategy Games," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, Pasadena, California, 2009, pp. 40–45.

[19] V. Berthier, H. Doghmen, and O. Teytaud, "Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search," in *Proc. Learn. Intell. Optim.*, Venice, Italy, 2010, pp. 111–124.

[20] R. Bjarnason, A. Fern, and P. Tadepalli, "Lower Bounding Klondike Solitaire with Monte-Carlo Planning," in *Proc. 19th Int. Conf. Automat. Plan. Sched.*, Thessaloniki, Greece, 2009, pp. 26–33.

[21] Y. Björnsson and H. Finnsson, "CadiaPlayer: A Simulation-Based General Game Player," *IEEE Trans. Comp. Intell. AI Games*, vol. 1, no. 1, pp. 4–15, 2009.

[22] Z. Bnaya, A. Felner, S. E. Shimony, D. Fried, and O. Maksin, "Repeated-task Canadian traveler problem," in *Proc. Symp. Combin. Search*, Barcelona, Spain, 2011, pp. 24–30.

[23] J. Borsboom, J.-T. Saito, G. M. J.-B. Chaslot, and J. W. H. M. Uiterwijk, "A Comparison of Monte-Carlo Methods for Phantom Go," in *Proc. BeNeLux Conf. Artif. Intell.*, Utrecht, Netherlands, 2007, pp. 57–64.

[24] A. Bourki, G. M. J.-B. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hérault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssière, and Z. Yu, "Scalability and Parallelization of Monte-Carlo Tree Search," in *Proc. Int. Conf. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 48–58.

[25] A. Bourki, M. Coulm, P. Rolet, O. Teytaud, and P. Vayssière, "Parameter Tuning by Simple Regret Algorithms and Multiple Simultaneous Hypothesis Testing," in *Proc. Int. Conf. Inform. Control, Autom. and Robot.*, Funchal, Portugal, 2010, pp. 169–173.

[26] * B. Bouzy, "Move Pruning Techniques for Monte-Carlo Go," in *Proc. Adv. Comput. Games, LNCS 4250*, Taipei, Taiwan, 2005, pp. 104–119.

[27] R. Briesemeister, "Analysis and Implementation of the Game OnTop," M.S. thesis, Maastricht Univ., Netherlands, 2009.

[28] C. Browne, "Automatic Generation and Evaluation of Recombination Games," Ph.D. dissertation, Qld. Univ. Tech. (QUT), Brisbane, 2008.

[29] ——, "On the Dangers of Random Playouts," *Int. Comp. Games Assoc. J.*, vol. 34, no. 1, pp. 25–26, 2010.

[30] ——, "Towards MCTS for Creative Domains," in *Proc. Int. Conf. Comput. Creat.*, Mexico City, Mexico, 2011, pp. 96–101.

[31] * B. Brügmann, "Monte Carlo Go," Max-Planke-Inst. Phys., Munich, Tech. Rep., 1993.

[32] S. Bubeck, R. Munos, and G. Stoltz, "Pure Exploration in Finitely-Armed and Continuously-Armed Bandits," *Theor. Comput. Sci.*, vol. 412, pp. 1832–1852, 2011.

[33] S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvári, "Online Optimization in X-Armed Bandits," in *Proc. Adv. Neur. Inform. Process. Sys.*, vol. 22, Vancouver, Canada, 2009, pp. 201–208.

[34] ——, "X-Armed Bandits," *J. Mach. Learn. Res.*, vol. 12, pp. 1587–1627, 2011.

[35] M. Buro, J. R. Long, T. Furtak, and N. R. Sturtevant, "Improving State Evaluation, Inference, and Search in Trick-Based Card Games," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, Pasadena, California, 2009, pp. 1407–1413.

[36] T. Cazenave, "A Phantom Go Program," in *Proc. Adv. Comput. Games*, Taipei, Taiwan, 2006, pp. 120–125.

[37] ——, "Evolving Monte-Carlo Tree Search Algorithms," Univ. Paris 8, Dept. Inform., Tech. Rep., 2007.

[38] ——, "Playing the Right Atari," *Int. Comp. Games Assoc. J.*, vol. 30, no. 1, pp. 35–42, 2007.

[39] ——, "Reflexive Monte-Carlo Search," in *Proc. Comput. Games Workshop*, Amsterdam, Netherlands, 2007, pp. 165–173.

[40] ——, "Multi-player Go," in *Proc. Comput. and Games, LNCS 5131*, Beijing, China, 2008, pp. 50–59.

[41] ——, "Monte-Carlo Kakuro," in *Proc. Adv. Comput. Games, LNCS 6048*, Pamplona, Spain, 2009, pp. 45–54.

[42] ——, "Nested Monte-Carlo Search," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, Pasadena, California, 2009, pp. 456–461.

[43] ——, "Nested Monte-Carlo Expression Discovery," in *Proc. Euro. Conf. Artif. Intell.*, Lisbon, Portugal, 2010, pp. 1057–1058.

[44] ——, "Monte-Carlo Approximation of Temperature," *Games of No Chance*, vol. 4, 2011.

[45] T. Cazenave, F. Balbo, and S. Pinson, "Monte-Carlo Bus Regulation," in *Proc. Int. IEEE Conf. Intell. Trans. Sys.*, St Louis, Missouri, 2009, pp. 340–345.

[46] T. Cazenave and J. Borsboom, "Golois Wins Phantom Go Tournament," *Int. Comp. Games Assoc. J.*, vol. 30, no. 3, pp. 165–166, 2007.

[47] T. Cazenave and N. Jouandeau, "On the Parallelization of UCT," in *Proc. Comput. Games Workshop*, Amsterdam, Netherlands, 2007, pp. 93–101.

[48] ——, "A parallel Monte-Carlo tree search algorithm," in *Proc. Comput. and Games, LNCS 5131*, Beijing, China, 2008, pp. 72–80.

[49] ——, "Parallel Nested Monte-Carlo search," in *Proc. IEEE Int. Parallel Distrib. Processes Symp.*, Rome, Italy, 2009, pp. 1–6.

[50] T. Cazenave and A. Saffidine, "Monte-Carlo Hex," in *Proc. Board Games Studies Colloq.*, Paris, France, 2010.

[51] ——, "Score Bounded Monte-Carlo Tree Search," in *Proc. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 93–104.

[52] G. M. J.-B. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo Tree Search: A New Framework for Game AI," in *Proc. Artif. Intell. Interact. Digital Entert. Conf.*, Stanford Univ., California, 2008, pp. 216–217.

[53] G. M. J.-B. Chaslot, L. Chatriot, C. Fiter, S. Gelly, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud, "Combining expert, offline, transient and online knowledge in Monte-Carlo exploration," Lab. Rech. Inform. (LRI), Paris, Tech. Rep., 2008.

[54] G. M. J.-B. Chaslot, S. de Jong, J.-T. Saito, and J. W. H. M. Uiterwijk, "Monte-Carlo Tree Search in Production Management Problems," in *Proc. BeNeLux Conf. Artif. Intell.*, Namur, Belgium, 2006, pp. 91–98.

[55] G. M. J.-B. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud, "Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search," in *Proc. Adv. Comput. Games, LNCS 6048*, vol. 6048, Pamplona, Spain, 2010, pp. 1–13.

[56] G. M. J.-B. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, O. Teytaud, and M. H. M. Winands, "Meta Monte-Carlo Tree Search for Automatic Opening Book Generation," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, Pasadena, California, 2009, pp. 7–12.

[57] G. M. J.-B. Chaslot, J.-B. Hoock, F. Teytaud, and O. Teytaud, "On the huge benefit of quasi-random mutations for multimodal optimization with application to grid-based tuning of neurocontrollers," in *Euro. Symp. Artif. Neur. Net.*, Bruges, Belgium, 2009.

[58] G. M. J.-B. Chaslot, M. H. M. Winands, I. Szita, and H. J. van den Herik, "Cross-Entropy for Monte-Carlo Tree Search," *Int. Comp. Games Assoc. J.*, vol. 31, no. 3, pp. 145–156, 2008.

[59] G. M. J.-B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo Tree Search," in *Proc. Comput. and Games, LNCS 5131*, Beijing, China, 2008, pp. 60–71.

[60] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive Strategies for Monte-Carlo Tree Search," *New Math. Nat. Comput.*, vol. 4, no. 3, pp. 343–357, 2008.

[61] K.-H. Chen, D. Du, and P. Zhang, "Monte-Carlo Tree Search and Computer Go," *Adv. Inform. Intell. Sys.*, vol. 251, pp. 201–225, 2009.

[62] J. Chevelu, T. Lavergne, Y. Lepage, and T. Moudenc, "Introduction of a new paraphrase generation tool based on Monte-Carlo sampling," in *Proc. 4th Int. Joint Conf. Natur. Lang. Process.*, vol. 2, Singapore, 2009, pp. 249–252.

[63] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and Move Groups in Monte Carlo Tree Search," in *Proc. IEEE Symp. Comput. Intell. Games*, Perth, Australia, 2008, pp. 389–395.

[64] C.-W. Chou, O. Teytaud, and S.-J. Yen, "Revisiting Monte-Carlo Tree Search on a Normal Form Game: NoGo," *Proc. Applicat. Evol. Comput., LNCS 6624*, pp. 73–82, 2011.

[65] P.-C. Chou, H. Doghmen, C.-S. Lee, F. Teytaud, O. Teytaud, H.-M. Wang, M.-H. Wang, L.-W. Wu, and S.-J. Yen, "Computational and Human Intelligence in Blind Go," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 235–242.

[66] P. Ciancarini and G. P. Favini, "Monte Carlo Tree Search Techniques in the Game of Kriegspiel," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, Pasadena, California, 2009, pp. 474–479.

[67] ——, "Monte Carlo tree search in Kriegspiel," *Artif. Intell.*, vol. 174, no. 11, pp. 670–684, Jul. 2010.

[68] R. Coquelin, Pierre-Arnaud and Munos, "Bandit Algorithms for Tree Search," in *Proc. Conf. Uncert. Artif. Intell.* Vancouver, Canada: AUAI Press, 2007, pp. 67–74.

[69] A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous Upper Confidence Trees," in *Proc. Learn. Intell. Optim.*, Rome, Italy, 2011, pp. 433–445.

[70] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Proc. 5th Int. Conf. Comput. and Games*, Turin, Italy, 2006, pp. 72–83.

[71] ——, "Computing Elo Ratings of Move Patterns in the Game of Go," *Int. Comp. Games Assoc. J.*, vol. 30, no. 4, pp. 198–208, 2007.

[72] ——, "Monte-Carlo Tree Search in Crazy Stone," in *Proc. Game Prog. Workshop*, Tokyo, Japan, 2007, pp. 74–75.

[73] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel, "Bandit-Based Optimization on Graphs with Application to Library Performance Tuning," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, Montreal, Canada, 2009, pp. 729–736.

[74] N. G. P. Den Teuling, "Monte-Carlo Tree Search for the Simultaneous Move Game Tron," Univ. Maastricht, Netherlands, Tech. Rep., 2011.

[75] P. D. Drake, "The Last-Good-Reply Policy for Monte-Carlo Go," *Int. Comp. Games Assoc. J.*, vol. 32, no. 4, pp. 221–227, 2009.

[76] P. D. Drake and S. Uurtamo, "Heuristics in Monte Carlo Go," in *Proc. Int. Conf. Artif. Intell.*, Las Vegas, Nevada, 2007, pp. 171–175.

[77] ——, "Move Ordering vs Heavy Playouts: Where Should Heuristics be Applied in Monte Carlo Go," in *Proc. 3rd North Amer. Game-On Conf.*, Gainesville, Florida, 2007, pp. 35–42.

[78] S. Edelkamp, P. Kissmann, D. Sulewski, and H. Messerschmidt, "Finding the Needle in the Haystack with Heuristically Guided Swarm Tree Search," in *Multikonf. Wirtschaftsinform.*, Gottingen, Germany, 2010, pp. 2295–2308.

[79] M. Enzenberger and M. Müller, "Fuego - An Open-source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search," Univ. Alberta, Edmonton, Tech. Rep. April, 2009.

[80] ——, "A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm," in *Proc. Adv. Comput. Games, LNCS 6048*, vol. 6048, Pamplona, Spain, 2010, pp. 14–20.

[81] M. Enzenberger, M. Müller, B. Arneson, and R. B. Segal, "Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 259–270, 2010.

[82] A. Fern and P. Lewis, "Ensemble Monte-Carlo Planning: An Empirical Study," in *Proc. 21st Int. Conf. Automat. Plan. Sched.*, Freiburg, Germany, 2011, pp. 58–65.

[83] H. Finnsson, "CADIA-Player: A General Game Playing Agent," M.S. thesis, Reykjavik Univ., Iceland, Mar. 2007.

[84] H. Finnsson and Y. Björnsson, "Simulation-Based Approach to General Game Playing," in *Proc. Assoc. Adv. Artif. Intell.*, Chicago, Illinois, 2008, pp. 259–264.

[85] ——, "Simulation Control in General Game Playing Agents," in *Proc. Int. Joint Conf. Artif. Intell. Workshop Gen. Game Playing*, Pasadena, California, 2009, pp. 21–26.

[86] ——, "Learning Simulation Control in General Game-Playing Agents," in *Proc. 24th AAAI Conf. Artif. Intell.*, Atlanta, Georgia, 2010, pp. 954–959.

[87] ——, "CadiaPlayer: Search-Control Techniques," *Künstliche Intelligenz*, vol. 25, no. 1, pp. 9–16, Jan. 2011.

[88] Y. Fu, S. Yang, S. He, J. Yang, X. Liu, Y. Chen, and D. Ji, "To Create Intelligent Adaptive Neuro-Controller of Game Opponent from UCT-Created Data," in *Proc. Fuzzy Sys. Knowl. Disc.*, Tianjin, China, 2009, pp. 445–449.

[89] R. Gaudel and M. Sebag, "Feature Selection as a One-Player Game," in *Proc. 27th Int. Conf. Mach. Learn.*, Haifa, Israel, 2010, pp. 359–366.

[90] S. Gelly, "A Contribution to Reinforcement Learning; Application to Computer-Go," Ph.D. dissertation, Univ. Paris-Sud, France, 2007.

[91] S. Gelly, J.-B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian, "The Parallelization of Monte-Carlo Planning," in *Proc. 5th Int. Conf. Inform. Control, Automat. and Robot.*, Funchal, Portugal, 2008, pp. 244–249.

[92] S. Gelly and D. Silver, "Combining Online and Offline Knowledge in UCT," in *Proc. 24th Annu. Int. Conf. Mach. Learn.* Corvalis, Oregon: ACM, 2007, pp. 273–280.

[93] ——, "Achieving Master Level Play in 9 x 9 Computer Go," in *Proc. Assoc. Adv. Artif. Intell.*, vol. 1, Chicago, Illinois, 2008, pp. 1537–1540.

[94] ——, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artif. Intell.*, vol. 175, no. 11, pp. 1856–1875, Jul. 2011.

[95] S. Gelly and Y. Wang, "Exploration exploitation in Go: UCT for Monte-Carlo Go," in *Proc. Adv. Neur. Inform. Process. Syst.*, Vancouver, Canada, 2006.

[96] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with Patterns in Monte-Carlo Go," Inst. Nat. Rech. Inform. Auto. (INRIA), Paris, Tech. Rep., 2006.

[97] * M. L. Ginsberg, "GIB: Imperfect Information in a Computationally Challenging Game," *J. Artif. Intell. Res.*, vol. 14, pp. 303–358, 2001.

[98] T. Graf, U. Lorenz, M. Platzner, and L. Schaefers, "Parallel Monte-Carlo Tree Search for HPC Systems," in *Proc. 17th*

Int. Euro. Conf. Parallel Distrib. Comput., LNCS 6853, Bordeaux, France, 2011, pp. 365–376.

[99] S. He, Y. Wang, F. Xie, J. Meng, H. Chen, S. Luo, Z. Liu, and Q. Zhu, "Game Player Strategy Pattern Recognition and How UCT Algorithms Apply Pre-knowledge of Player's Strategy to Improve Opponent AI," in *Proc. 2008 Int. Conf. Comput. Intell. Model. Control Automat.*, Vienna, Austria, Dec. 2008, pp. 1177–1181.

[100] S. He, F. Xie, Y. Wang, S. Luo, Y. Fu, J. Yang, Z. Liu, and Q. Zhu, "To Create Adaptive Game Opponent by Using UCT," in *Proc. 2008 Int. Conf. Comput. Intell. Model. Control Automat.*, Vienna, Austria, Dec. 2008, pp. 67–70.

[101] D. P. Helmbold and A. Parker-Wood, "All-Moves-As-First Heuristics in Monte-Carlo Go," in *Proc. Int. Conf. Artif. Intell.*, Las Vegas, Nevada, 2009, pp. 605–610.

[102] B. Helmstetter, C.-S. Lee, F. Teytaud, O. Teytaud, M.-H. Wang, and S.-J. Yen, "Random positions in Go," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 250–257.

[103] P. Hingston and M. Masek, "Experiments with Monte Carlo Othello," in *Proc. IEEE Congr. Evol. Comput.*, Singapore, 2007, pp. 4059–4064.

[104] J.-B. Hoock, C.-S. Lee, A. Rimmel, F. Teytaud, O. Teytaud, and M.-H. Wang, "Intelligent Agents for the Game of Go," *IEEE Comput. Intell. Mag.*, vol. 5, no. 4, pp. 28–42, 2010.

[105] J.-B. Hoock and O. Teytaud, "Bandit-Based Genetic Programming," in *Proc. Euro. Conf. Gen. Prog.*, vol. 6021, Istanbul, Turkey, 2010, pp. 268–277.

[106] J. Huang, Z. Liu, B. Lu, and F. Xiao, "Pruning in UCT Algorithm," in *Proc. Int. Conf. Tech. Applicat. Artif. Intell.*, Hsinchu, Taiwan, 2010, pp. 177–181.

[107] S.-C. Huang, "New Heuristics for Monte Carlo Tree Search Applied to the Game of Go," Ph.D. dissertation, Nat. Taiwan Normal Univ., Taipei, 2011.

[108] S.-C. Huang, R. Coulom, and S.-S. Lin, "Monte-Carlo Simulation Balancing Applied to 9x9 Go," *Int. Comp. Games Assoc. J.*, vol. 33, no. 4, pp. 191–201, 2010.

[109] ——, "Monte-Carlo Simulation Balancing in Practice," in *Proc. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 81–92.

[110] ——, "Time Management for Monte-Carlo Tree Search Applied to the Game of Go," in *Proc. Int. Conf. Tech. Applicat. Artif. Intell.*, Hsinchu City, Taiwan, 2010, pp. 462–466.

[111] N. Ikehata and T. Ito, "Monte Carlo Tree Search in Ms. Pac-Man," in *Proc. 15th Game Progr. Workshop*, Kanagawa, Japan, 2010, pp. 1–8.

[112] ——, "Monte-Carlo Tree Search in Ms. Pac-Man," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 39–46.

[113] H. Kato and I. Takeuchi, "Parallel Monte-Carlo Tree Search with Simulation Servers," in *Proc. Int. Conf. Tech. Applicat. Artif. Intell.*, Hsinchu City, Taiwan, 2010, pp. 491–498.

[114] J. Kloetzer, "Experiments in Monte-Carlo Amazons," *J. Inform. Process. Soc. Japan*, vol. 2010-GI-24, no. 6, pp. 1–4, 2010.

[115] ——, "Monte-Carlo Opening Books for Amazons," in *Proc. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 124–135.

[116] J. Kloetzer, H. Iida, and B. Bouzy, "The Monte-Carlo Approach in Amazons," in *Proc. Comput. Games Workshop*, Amsterdam, Netherlands, 2007, pp. 113–124.

[117] ——, "A Comparative Study of Solvers in Amazons Endgames," in *Proc. IEEE Conf. Comput. Intell. Games*, Perth, Australia, 2008, pp. 378–384.

[118] ——, "Playing Amazons Endgames," *Int. Comp. Games Assoc. J.*, vol. 32, no. 3, pp. 140–148, 2009.

[119] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," in *Euro. Conf. Mach. Learn.* Berlin, Germany: Springer, 2006, pp. 282–293.

[120] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved Monte-Carlo Search," Univ. Tartu, Estonia, Tech. Rep. 1, 2006.

[121] S. Könnecke and J. Waldmann, "Efficient Playouts for the Havannah Abstract Board Game," Hochschule Technik, Leipzig, Tech. Rep., 2009.

[122] T. Kozelek, "Methods of MCTS and the game Arimaa," M.S. thesis, Charles Univ., Prague, 2009.

[123] K. L. Kroeker, "A New Benchmark for Artificial Intelligence," *Commun. ACM*, vol. 54, no. 8, pp. 13–15, Aug. 2011.

[124] * T. L. Lai and H. Robbins, "Asymptotically Efficient Adaptive Allocation Rules," *Adv. Appl. Math.*, vol. 6, pp. 4–22, 1985.

[125] * M. Lanctot, K. Waugh, M. Zinkevich, and M. Bowling, "Monte Carlo Sampling for Regret Minimization in Extensive Games," in *Proc. Adv. Neur. Inform. Process. Sys.*, Vancouver, Canada, 2009, pp. 1078–1086.

[126] * S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," Iowa State Univ., Comp Sci. Dept., TR 98-11, Tech. Rep., 1998.

[127] C.-S. Lee, M. Müller, and O. Teytaud, "Guest Editorial: Special Issue on Monte Carlo Techniques and Computer Go," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 225–228, Dec. 2010.

[128] C.-S. Lee, M.-H. Wang, G. M. J.-B. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong, "The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments," *IEEE Trans. Comp. Intell. AI Games*, vol. 1, no. 1, pp. 73–89, 2009.

[129] C.-S. Lee, M.-H. Wang, T.-P. Hong, G. M. J.-B. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, and Y.-H. Kuo, "A Novel Ontology for Computer Go Knowledge Management," in *Proc. IEEE Int. Conf. Fuzzy Sys.*, Jeju Island, Korea, Aug. 2009, pp. 1056–1061.

[130] J. R. Long, N. R. Sturtevant, M. Buro, and T. Furtak, "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search," in *Proc. Assoc. Adv. Artif. Intell.*, Atlanta, Georgia, 2010, pp. 134–140.

[131] * S. Lopez, "Rybka's Monte Carlo analysis," 2008. [Online]. Available: http://www.chessbase.com/newsdetail.asp?newsid=5075

[132] R. J. Lorentz, "Amazons Discover Monte-Carlo," in *Proc. Comput. and Games, LNCS 5131*, Beijing, China, 2008, pp. 13–24.

[133] ——, "Improving Monte-Carlo Tree Search in Havannah," in *Proc. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 105–115.

[134] ——, "Castro Wins Havannah Tournament," *Int. Comp. Games Assoc. J.*, vol. 33, no. 4, p. 232, 2011.

[135] T. Mahlmann, J. Togelius, and G. N. Yannakakis, "Modelling and evaluation of complex scenarios with the Strategy Game Description Language," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 174–181.

[136] ——, "Towards Procedural Strategy Game Generation: Evolving Complementary Unit Types," in *Proc. Applicat. Evol. Comput., LNCS 6624*, Torino, Italy, 2011, pp. 93–102.

[137] R. Maîtrepierre, J. Mary, and R. Munos, "Adaptive play in Texas Hold'em Poker," in *Proc. Euro. Conf. Artif. Intell.*, Patras, Greece, 2008, pp. 458–462.

[138] C. Mansley, A. Weinstein, and M. L. Littman, "Sample-Based Planning for Continuous Action Markov Decision Processes," in *Proc. 21st Int. Conf. Automat. Plan. Sched.*, Freiburg, Germany, 2011, pp. 335–338.

[139] L. S. Marcolino and H. Matsubara, "Multi-Agent Monte Carlo Go," in *Proc. Int. Conf. Auton. Agents Multi. Sys.*, Taipei, Taiwan, 2011, pp. 21–28.

[140] S. Matsumoto, N. Hirosue, K. Itonaga, K. Yokoo, and H. Futahashi, "Evaluation of Simulation Strategy on Single-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem," in *Proc. Int. Multi Conf. Eng. Comput. Scientists*, vol. 3, Hong Kong, 2010, pp. 2086–2091.

[141] * R. E. McInerney, "Multi-Armed Bandit Bayesian Decision Making," Univ. Oxford, Oxford, Tech. Rep., 2010.

[142] J. Méhat and T. Cazenave, "Ary: A Program for General Game Playing," Univ. Paris 8, Dept. Inform., Tech. Rep., 2008.

[143] ——, "Monte-Carlo Tree Search for General Game Playing," Univ. Paris 8, Dept. Info., Tech. Rep., 2008.

[144] ——, "Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 271–277, 2010.

[145] ——, "A Parallel General Game Player," *Künstliche Intelligenz*, vol. 25, no. 1, pp. 43–47, 2011.

[146] ——, "Tree Parallelization of Ary on a Cluster," in *Proc. Int. Joint Conf. Artif. Intell.*, Barcelona, Spain, 2011, pp. 39–43.

[147] M. Möller, M. Schneider, M. Wegner, and T. Schaub, "Centurio, a General Game Player: Parallel, Java- and ASP-based," *Künstliche Intelligenz*, vol. 25, no. 1, pp. 17–24, Dec. 2010.

[148] M. Müller, "Fuego-GB Prototype at the Human machine competition in Barcelona 2010: a Tournament Report and Analysis," Univ. Alberta, Edmonton, Tech. Rep., 2010.

[149] H. Nakhost and M. Müller, "Monte-Carlo Exploration for Deterministic Planning," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, Pasadena, California, 2009, pp. 1766–1771.

[150] M. Naveed, D. E. Kitchin, and A. Crampton, "Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games," in *Proc. 28th Workshop UK Spec. Inter. Group Plan. Sched.*, Brescia, Italy, 2010, pp. 125–132.

[151] K. Q. Nguyen, T. Miyama, A. Yamada, T. Ashida, and R. Thawonmas, "ICE gUCT," Int. Comput. Entertain. Lab., Ritsumeikan Univ., Tech. Rep., 2011.

[152] J. P. A. M. Nijssen, "Playing Othello Using Monte Carlo," *Strategies*, pp. 1–9, 2007.

[153] ——, "Using Intelligent Search Techniques to Play the Game Khet," M. S. Thesis, Maastricht Univ., Netherlands, 2009.

[154] J. P. A. M. Nijssen and J. W. H. M. Uiterwijk, "Using Intelligent Search Techniques to Play the Game Khet," Maastricht Univ., Netherlands, Netherlands, Tech. Rep., 2009.

[155] J. P. A. M. Nijssen and M. H. M. Winands, "Enhancements for Multi-Player Monte-Carlo Tree Search," in *Proc. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 238–249.

[156] ——, "Monte-Carlo Tree Search for the Game of Scotland Yard," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 158–165.

[157] Y. Osaki, K. Shibahara, Y. Tajima, and Y. Kotani, "An Othello Evaluation Function Based on Temporal Difference Learning using Probability of Winning," in *Proc. IEEE Conf. Comput. Intell. Games*, Perth, Australia, Dec. 2008, pp. 205–211.

[158] D. Pellier, B. Bouzy, and M. Métivier, "An UCT Approach for Anytime Agent-Based Planning," in *Proc. Int. Conf. Pract. Appl. Agents Multi. Sys.*, Salamanca, Spain, 2010, pp. 211–220.

[159] M. Ponsen, G. Gerritsen, and G. M. J.-B. Chaslot, "Integrating Opponent Models with Monte-Carlo Tree Search in Poker," in *Proc. Conf. Assoc. Adv. Artif. Intell.: Inter. Decis. Theory Game Theory Workshop*, Atlanta, Georgia, 2010, pp. 37–42.

[160] A. Previti, R. Ramanujan, M. Schaerf, and B. Selman, "Monte-Carlo Style UCT Search for Boolean Satisfiability," in *Proc. 12th Int. Conf. Ital. Assoc. Artif. Intell., LNCS 6934*, Palermo, Italy, 2011, pp. 177–188.

[161] T. Raiko and J. Peltonen, "Application of UCT Search to the Connection Games of Hex, Y, *Star, and Renkula!" in *Proc. Finn. Artif. Intell. Conf.*, Espoo, Finland, 2008, pp. 89–93.

[162] R. Ramanujan, A. Sabharwal, and B. Selman, "On Adversarial Search Spaces and Sampling-Based Planning," in *Proc. 20th Int. Conf. Automat. Plan. Sched.*, Toronto, Canada, 2010, pp. 242–245.

[163] ——, "Understanding Sampling Style Adversarial Search Methods," in *Proc. Conf. Uncert. Artif. Intell.*, Catalina Island, California, 2010, pp. 474–483.

[164] ——, "On the Behavior of UCT in Synthetic Search Spaces," in *Proc. 21st Int. Conf. Automat. Plan. Sched.*, Freiburg, Germany, 2011.

[165] R. Ramanujan and B. Selman, "Trade-Offs in Sampling-Based Adversarial Planning," in *Proc. 21st Int. Conf. Automat. Plan. Sched.*, Freiburg, Germany, 2011, pp. 202–209.

[166] A. Rimmel, "Improvements and Evaluation of the Monte-Carlo Tree Search Algorithm," Ph.D. dissertation, Lab. Rech. Inform. (LRI), Paris, 2009.

[167] A. Rimmel and F. Teytaud, "Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search," in *Proc. Applicat. Evol. Comput. 1, LNCS 6624*, Torino. Italy, 2010, pp. 201–210.

[168] A. Rimmel, F. Teytaud, and T. Cazenave, "Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows," in *Proc. Applicat. Evol. Comput. 2, LNCS 6625*, Torino, Italy, 2011, pp. 501–510.

[169] A. Rimmel, F. Teytaud, and O. Teytaud, "Biasing Monte-Carlo Simulations through RAVE Values," in *Proc. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 59–68.

[170] A. Rimmel, O. Teytaud, C.-S. Lee, S.-J. Yen, M.-H. Wang, and S.-R. Tsai, "Current Frontiers in Computer Go," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 229–238, 2010.

[171] * D. Robles and S. M. Lucas, "A Simple Tree Search Method for Playing Ms. Pac-Man," in *Proc. IEEE Conf. Comput. Intell. Games*, Milan, Italy, 2009, pp. 249–255.

[172] D. Robles, P. Rohlfshagen, and S. M. Lucas, "Learning Non-Random Moves for Playing Othello: Improving Monte Carlo Tree Search," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 305–312.

[173] P. Rolet, M. Sebag, and O. Teytaud, "Boosting Active Learning to Optimality: a Tractable Monte-Carlo, Billiard-based Algorithm," in *Proc. Euro. Conf. Mach. Learn. Knowl. Disc. Datab.* Bled, Slovenia: Springer, 2009, pp. 302–317.

[174] ——, "Optimal Robust Expensive Optimization is Tractable," in *Proc. 11th Annu. Conf. Genet. Evol. Comput.*, Montreal, Canada, 2009, pp. 1951–1956.

[175] ——, "Upper Confidence Trees and Billiards for Optimal Active Learning," in *Proc. Conf. l'Apprentissage Autom.*, Hammamet, Tunisia, 2009.

[176] C. D. Rosin, "Nested Rollout Policy Adaptation for Monte Carlo Tree Search," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, Barcelona, Spain, 2011, pp. 649–654.

[177] J. Rubin and I. Watson, "Computer poker: A review," *Artif. Intell.*, vol. 175, no. 5-6, pp. 958–987, Apr. 2011.

[178] * S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, New Jersey: Prentice Hall, 2009.

[179] A. Sabharwal and H. Samulowitz, "Guiding Combinatorial Optimization with UCT," in *Proc. 21st Int. Conf. Automat. Plan. Sched.*, Freiburg, Germany, 2011.

[180] A. Saffidine, "Utilisation dUCT au Hex," Ecole Normale Super. Lyon, France, Tech. Rep., 2008.

[181] ——, "Some Improvements for Monte-Carlo Tree Search, Game Description Language Compilation, Score Bounds and Transpositions," M.S. thesis, Paris-Dauphine Lamsade, France, 2010.

[182] A. Saffidine, T. Cazenave, and J. Méhat, "UCD: Upper Confidence bound for rooted Directed acyclic graphs," in *Proc. Conf. Tech. Applicat. Artif. Intell.*, Hsinchu City, Taiwan, 2010, pp. 467–473.

[183] * J.-T. Saito, G. M. J.-B. Chaslot, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Monte-Carlo Proof-Number Search for Computer Go," in *Proc. 5th Int. Conf. Comput. and Games*, Turin, Italy, 2006, pp. 50–61.

[184] S. Samothrakis, D. Robles, and S. M. Lucas, "A UCT Agent for Tron: Initial Investigations," in *Proc. IEEE Symp. Comput. Intell. Games*, Dublin, Ireland, 2010, pp. 365–371.

[185] ——, "Fast Approximate Max-n Monte-Carlo Tree Search for Ms Pac-Man," *IEEE Trans. Comp. Intell. AI Games*, vol. 3, no. 2, pp. 142–154, 2011.

[186] Y. Sato, D. Takahashi, and R. Grimbergen, "A Shogi Program Based on Monte-Carlo Tree Search," *Int. Comp. Games Assoc. J.*, vol. 33, no. 2, pp. 80–92, 2010.

[187] B. Satomi, Y. Joe, A. Iwasaki, and M. Yokoo, "Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, Barcelona, Spain, 2011, pp. 655–662.

[188] F. C. Schadd, "Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis," M.S. thesis, Maastricht Univ., Netherlands, 2009.

[189] M. P. D. Schadd, "Selective Search in Games of Different Complexity," Ph.D. dissertation, Maastricht Univ., Netherlands, 2011.

[190] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, and H. Aldewereld, "Addressing NP-Complete Puzzles with Monte-Carlo Methods," in *Proc. Artif. Intell. Sim. Behav. Symp. Logic Sim. Interact. Reason.*, Aberdeen, UK, 2008, pp. 55–61.

[191] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J.-B. Chaslot, and J. W. H. M. Uiterwijk, "Single-Player Monte-Carlo Tree Search," in *Proc. Comput. and Games, LNCS 5131*, Beijing, China, 2008, pp. 1–12.

[192] L. Schaefers, M. Platzner, and U. Lorenz, "UCT-Treesplit - Parallel MCTS on Distributed Memory," in *Proc. 21st Int. Conf. Automat. Plan. Sched.*, Freiburg, Germany, 2011.

[193] * J. Schaeffer, "The History Heuristic and Alpha-Beta Search Enhancements in Practice," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 11, no. 11, pp. 1203–1212, 1989.

[194] J. Schäfer, "The UCT Algorithm Applied to Games with Imperfect Information," Diploma thesis, Otto-Von-Guericke Univ. Magdeburg, Germany, 2008.

[195] R. B. Segal, "On the Scalability of Parallel UCT," in *Proc. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 36–47.

[196] M. Shafiei, N. R. Sturtevant, and J. Schaeffer, "Comparing UCT versus CFR in Simultaneous Games," in *Proc. Int. Joint Conf. Artif. Intell. Workshop Gen. Game Playing*, Pasadena, California, 2009.

[197] S. Sharma, Z. Kobti, and S. Goodwin, "Knowledge Generation for Improving Simulations in UCT for General Game Playing,"

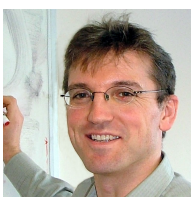
in *Proc. Adv. Artif. Intell.*, Auckland, New Zealand, 2008, pp. 49–55.
[198] ——, "Learning and Knowledge Generation in General Games," in *Proc. IEEE Symp. Comput. Intell. Games*, Perth, Australia, Dec. 2008, pp. 329–335.
[199] * B. Sheppard, "World-championship-caliber Scrabble," *Artif. Intell.*, vol. 134, pp. 241–275, 2002.
[200] K. Shibahara and Y. Kotani, "Combining Final Score with Winning Percentage by Sigmoid Function in Monte-Carlo Simulations," in *Proc. IEEE Conf. Comput. Intell. Games*, Perth, Australia, Dec. 2008, pp. 183–190.
[201] D. Silver, "Reinforcement Learning and Simulation-Based Search in Computer Go," Ph.D. dissertation, Univ. Alberta, Edmonton, 2009.
[202] D. Silver, R. S. Sutton, and M. Müller, "Sample-Based Learning and Search with Permanent and Transient Memories," in *Proc. 25th Annu. Int. Conf. Mach. Learn.*, Helsinki, Finland, 2008, pp. 968–975.
[203] D. Silver and G. Tesauro, "Monte-Carlo Simulation Balancing," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, Montreal, Canada, 2009, pp. 945–952.
[204] D. Silver and J. Veness, "Monte-Carlo Planning in Large POMDPs," in *Proc. Neur. Inform. Process. Sys.*, Vancouver, Canada, 2010, pp. 1–9.
[205] Y. Soejima, A. Kishimoto, and O. Watanabe, "Evaluating Root Parallelization in Go," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 278–287, 2010.
[206] J. A. Stankiewicz, "Knowledge-Based Monte-Carlo Tree Search in Havannah," M.S. thesis, Maastricht Univ., Netherlands, 2011.
[207] N. R. Sturtevant, "An Analysis of UCT in Multi-Player Games," in *Proc. Comput. and Games, LNCS 5131*, Beijing, China, 2008, pp. 37–49.
[208] N. Sylvester, B. Lohre, S. Dodson, and P. D. Drake, "A Linear Classifier Outperforms UCT in 9x9 Go," in *Proc. Int. Conf. Artif. Intell.*, Las Vegas, Nevada, 2011, pp. 804–808.
[209] I. Szita, G. M. J.-B. Chaslot, and P. Spronck, "Monte-Carlo Tree Search in Settlers of Catan," in *Proc. Adv. Comput. Games*, Pamplona, Spain, 2010, pp. 21–32.
[210] S. Takeuchi, T. Kaneko, and K. Yamaguchi, "Evaluation of Monte Carlo Tree Search and the Application to Go," in *Proc. IEEE Conf. Comput. Intell. Games*, Perth, Australia, Dec. 2008, pp. 191–198.
[211] ——, "Evaluation of Game Tree Search Methods by Game Records," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 288–302, 2010.
[212] Y. Tanabe, K. Yoshizoe, and H. Imai, "A Study on Security Evaluation Methodology for Image-Based Biometrics Authentication Systems," in *Proc. IEEE Conf. Biom.: Theory, Applicat. Sys.*, Washington, DC, 2009, pp. 1–6.
[213] G. Tesauro, V. T. Rajan, and R. B. Segal, "Bayesian Inference in Monte-Carlo Tree Search," in *Proc. Conf. Uncert. Artif. Intell.*, Catalina Island, California, 2010, pp. 580–588.
[214] F. Teytaud and O. Teytaud, "Creating an Upper-Confidence-Tree program for Havannah," in *Proc. Adv. Comput. Games, LNCS 6048*, Pamplona, Spain, 2010, pp. 65–74.
[215] ——, "On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms," in *Proc. IEEE Symp. Comput. Intell. Games*, no. 1, Dublin, Ireland, 2010, pp. 359–364.
[216] ——, "Lemmas on Partial Observation, with Application to Phantom Games," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 243–249.
[217] O. Teytaud and S. Flory, "Upper Confidence Trees with Short Term Partial Information," in *Proc. Applicat. Evol. Comput. 1, LNCS 6624*, Torino, Italy, 2011, pp. 153–162.
[218] D. Tom, "Investigating UCT and RAVE: Steps towards a more robust method," M.S. thesis, Univ. Alberta, Edmonton, 2010.
[219] D. Tom and M. Müller, "A Study of UCT and its Enhancements in an Artificial Game," in *Proc. Adv. Comput. Games, LNCS 6048*, Pamplona, Spain, 2010, pp. 55–64.
[220] ——, "Computational Experiments with the RAVE Heuristic," in *Proc. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 69–80.
[221] B. K.-B. Tong, C. M. Ma, and C. W. Sung, "A Monte-Carlo Approach for the Endgame of Ms. Pac-Man," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 9–15.
[222] B. K.-B. Tong and C. W. Sung, "A Monte-Carlo Approach for Ghost Avoidance in the Ms. Pac-Man Game," in *Proc. IEEE Consum. Elect. Soc. Games Innov. Conf.*, Hong Kong, 2011, pp. 1–8.
[223] G. van den Broeck, K. Driessens, and J. Ramon, "Monte-Carlo Tree Search in Poker using Expected Reward Distributions," *Adv. Mach. Learn., LNCS 5828*, no. 1, pp. 367–381, 2009.
[224] H. J. van den Herik, "The Drosophila Revisited," *Int. Comp. Games Assoc. J.*, vol. 33, no. 2, pp. 65–66., 2010.
[225] F. van Lishout, G. M. J.-B. Chaslot, and J. W. H. M. Uiterwijk, "Monte-Carlo Tree Search in Backgammon," in *Proc. Comput. Games Workshop*, Amsterdam, Netherlands, 2007, pp. 175–184.
[226] J. Veness, K. S. Ng, M. Hutter, W. Uther, and D. Silver, "A Monte-Carlo AIXI Approximation," *J. Artif. Intell. Res.*, vol. 40, pp. 95–142, 2011.
[227] T. J. Walsh, S. Goschin, and M. L. Littman, "Integrating Sample-based Planning and Model-based Reinforcement Learning," in *Proc. Assoc. Adv. Artif. Intell.*, Atlanta, Georgia, 2010, pp. 612–617.
[228] Y. Wang and S. Gelly, "Modifications of UCT and sequence-like simulations for Monte-Carlo Go," in *Proc. IEEE Symp. Comput. Intell. Games*, Honolulu, Hawaii, 2007, pp. 175–182.
[229] C. D. Ward and P. I. Cowling, "Monte Carlo Search Applied to Card Selection in Magic: The Gathering," in *Proc. IEEE Symp. Comput. Intell. Games*, Milan, Italy, 2009, pp. 9–16.
[230] D. Whitehouse, E. J. Powley, and P. I. Cowling, "Determinization and Information Set Monte Carlo Tree Search for the Card Game Dou Di Zhu," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 87–94.
[231] G. M. J. Williams, "Determining Game Quality Through UCT Tree Shape Analysis," M.S. thesis, Imperial Coll., London, 2010.
[232] M. H. M. Winands and Y. Björnsson, "Evaluation Function Based Monte-Carlo LOA," in *Proc. Adv. Comput. Games, LNCS 6048*, Pamplona, Spain, 2010, pp. 33–44.
[233] ——, "$\alpha\beta$-based Play-outs in Monte-Carlo Tree Search," in *IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 110–117.
[234] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, "Monte-Carlo Tree Search Solver," in *Proc. Comput. and Games, LNCS 5131*, Beijing, China, 2008, pp. 25–36.
[235] ——, "Monte-Carlo Tree Search in Lines of Action," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 239–250, 2010.
[236] M. H. M. Winands, Y. Björnsson, and J.-t. Saito, "Monte Carlo Tree Search in Lines of Action," *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 239–250, 2010.
[237] F. Xie and Z. Liu, "Backpropagation Modification in Monte-Carlo Game Tree Search," in *Proc. Int. Symp. Intell. Inform. Tech. Applicat.*, NanChang, China, 2009, pp. 125–128.
[238] F. Xie, H. Nakhost, and M. Müller, "A Local Monte Carlo Tree Search Approach in Deterministic Planning," in *Proc. Assoc. Adv. Artif. Intell.*, San Francisco, California, 2011, pp. 1832–1833.
[239] J. Yang, Y. Gao, S. He, X. Liu, Y. Fu, Y. Chen, and D. Ji, "To Create Intelligent Adaptive Game Opponent by Using Monte-Carlo for Tree Search," in *Proc. 5th Int. Conf. Natural Comput.*, Tianjian, China, 2009, pp. 603–607.
[240] Y. Zhang, S. He, J. Wang, Y. Gao, J. Yang, X. Yu, and L. Sha, "Optimizing Player's Satisfaction through DDA of Game AI by UCT for the Game Dead-End," in *Proc. 6th Int. Conf. Natural Comput.*, Yantai, China, 2010, pp. 4161–4165.

Cameron Browne (IEEE) received a Ph.D. in Computer Science from the Queensland University of Technology (QUT), Australia, in 2008, winning the Dean's Award for Outstanding Thesis. He was Canon Australia's Inventor of the Year for 1998. He is currently a Research Fellow at Imperial College, London, working on the EPSRC project *UCT for Games and Beyond*, investigating MCTS methods for procedural content generation in creative domains such as game design, linguistics, and generative art and music.

Edward Powley (IEEE) received an M.Math degree in Mathematics and Computer Science from the University of York, UK, in 2006, and was awarded the P. B. Kennedy Prize and the BAE Systems ATC Prize. He received a Ph.D. in Computer Science from the University of York in 2010. He is currently a Research Fellow at the University of Bradford, investigating MCTS for games with hidden information and stochastic outcomes. His other research interests include cellular automata, and game theory for security.

Daniel Whitehouse (IEEE) received the Master of Mathematics degree in Mathematics from the University of Manchester, UK, in 2010. He is currently pursuing a Ph.D. in Artificial Intelligence in the School of Computing, Informatics and Media at the University of Bradford and is funded as part of the EPSRC project *UCT for games and Beyond*. He is published in the domain of Monte Carlo Tree Search and is investigating the application of Monte Carlo Tree Search methods to games with chance and hidden information.

Simon Lucas (SMIEEE) is a professor of Computer Science at the University of Essex (UK) where he leads the Game Intelligence Group. His main research interests are games, evolutionary computation, and machine learning, and he has published widely in these fields with over 130 peer-reviewed papers. He is the inventor of the scanning n-tuple classifier, and is the founding Editor-in-Chief of the IEEE Transactions on Computational Intelligence and AI in Games.



Peter Cowling (IEEE) is Professor of Computer Science and Associate Dean (Research and Knowledge Transfer) at the University of Bradford (UK), where he leads the Artificial Intelligence research centre. His work centres on computerised decision-making in games, scheduling and resource-constrained optimisation. He has worked with a wide range of industrial partners, is director of 2 research spin-out companies, and has published over 80 scientific papers in high-quality journals and conferences.

Philipp Rohlfshagen received a B.Sc. in Computer Science and Artificial Intelligence from the University of Sussex, UK, in 2003, winning the prize for best undergraduate final year project. He received the M.Sc. in Natural Computation in 2004 and a Ph.D. in Evolutionary Computation in 2007, both from the University of Birmingham, UK. Philipp is now a senior research officer at the University of Essex, working together with Professor Simon Lucas on Monte Carlo Tree Search for real-time video games.

Stephen Tavener received a B.Sc. from Queen Mary and Westfield College, London, in 1989. He is currently pursuing a Ph.D. in the Computational Creativity Group, Imperial College, London, on benevolence in artificial agents. He has worked for the BBC and Zillions of Games, reviewed for major abstract games magazines, and run the London-based retail shop Gamesale. His interests include board game design, fractal geometry and program optimisation.

Diego Perez received a B.Sc. and a M.Sc. in Computer Science from University Carlos III, Madrid, in 2007. He is currently pursuing a Ph.D. in Artificial Intelligence applied to games at the University of Essex, Colchester. He has published in the domain of artificial intelligence applied to games and participated in several AI competitions such as the Simulated Car Racing Competition and Mario AI. He also has programming experience in the videogames industry with titles published for game consoles and PC.

Spyridon Samothrakis is currently pursuing a Ph.D. in Computational Intelligence and Games at the University of Essex. He holds a B.Sc. from the University of Sheffield (Computer Science) and an M.Sc. from the University of Sussex (Intelligent Systems). His interests include game theory, machine learning, evolutionary algorithms, consciousness and abolishing work.

Simon Colton is Reader in Computational Creativity at Imperial College, London, and an EPSRC Leadership Fellow. He heads the Computational Creativity Group, which studies notions related to creativity in software. He has published over 120 papers on AI topics such as machine learning, constraint solving, computational creativity, evolutionary techniques, the philosophy of science, mathematical discovery, visual arts and game design. He is the author of the programs HR and The Painting Fool.

	Go	Phantom Go	Blind Go	NoGo	Multi-player Go	Hex	Y, Star, Renkula!	Havannah	Lines of Action	P-Game	Clobber	Othello	Amazons	Arimaa	Khet	Shogi	Mancala	Blokus Duo	Focus	Chinese Checkers	Yavalath	Connect Four	Tic Tac Toe	Sum of Switches	Chess	LeftRight Games	Morpion Solitaire	Crossword	SameGame	Sudoku, Kakuro	Wumpus World	Mazes,Tigers,Grids	CADIAPLAYER	ARY
Flat MC/UCB	+										+		+																					
BAST																																		
TDMC(λ)												+																						
BB Active Learner																																		
UCT	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+			+	?								+	+
SP-MCTS																												+						
FUSE																																		
MP-MCTS																			+	+														
Coalition Reduction																																		
Multi-agent MCTS	+																																	
Ensemble MCTS	+																																	
HOP																																		
Sparse UCT																																		
Info Set UCT																																		
Multiple MCTS																							+											
UCT+																																		
$MC_{\alpha\beta}$									+																?									
MCCFR																																		
Reflexive MC																											+							
Nested MC																										+	+		+	+				+
NRPA																											+	+						
HGSTS																											+		+					
FSSS, BFS3																															+			
TAG																																		
UNLEO																																		
UCTSAT																																		
ρUCT																							+									+		
MRW																																		
MHSP																																		
UCB1-Tuned																																		
Bayesian UCT																																		
EXP3																																		
HOOT																																		
First Play Urgency	+																																	
(Anti)Decisive Moves	+			+				+													+													
Move Groups	+									+																								
Move Ordering	+											+				+																		
Transpositions	+									+				+	+											+								
Progressive Bias	+																																	
Opening Books	+												+																					
MCPG																																		
Search Seeding	+																																	
Parameter Tuning	+																																	
History Heuristic	+													+		+			+	+													+	
AMAF	+	+						+																			+							
RAVE	+	+	+	+	+	+		+						+										+									+	
Killer RAVE								+																										
RAVE-max																								+										
PoolRAVE	+							+																										
MCTS-Solver									+																									
MC-PNS	+																																	
Score Bounded MCTS	+																					+												
Progressive Widening	+												+																					
Pruning	+					+							+									+												
Contextual MC	+							+																										
Fill the Board	+					+	+																											
MAST, PAST, FAST																																	+	
Simulation Balancing	+																																	
Last Good Reply	+							+																										
Patterns	+					+	+	+																										
Score Bonus																								+										
Decaying Reward										+																								
Leaf Parallelisation	+													+			+																+	
Root Parallelisation	+													+																				
Tree Parallelisation	+																																	+
UCT-Treesplit	+																																	+

TABLE 3
Summary of MCTS variations and enhancements applied to combinatorial games.

	Tron	Ms. Pac-Man	Pocman, Battleship	Dead-End	Wargus	ORTS	Skat	Bridge	Poker	Dou Di Zhu	Klondike Solitaire	Magic: The Gathering	Phantom Chess	Urban Rivals	Backgammon	Settlers of Catan	Scotland Yard	Roshambo	Thurn and Taxis	OnTop	Security	Mixed Integer Prog.	TSP, CTP	Sailing Domain	Physics Simulations	Function Approx.	Constraint Satisfaction	Schedul. Benchmarks	Printer Scheduling	Rock-Sample Problem	PMPs	Bus Regulation	Large State Spaces	Feature Selection	PCG
Flat MC/UCB		+	+	+				+	+			+							+	+						+									
BAST																										+									
TDMC(λ)																																			
BB Active Learner																																			
UCT	+	+		+	+	+			+		+		+	+	+				+	+	+	+	+	+		+	+			+		+			+
SP-MCTS																														+					
FUSE																																		+	
MP-MCTS		+														+																			
Coalition Reduction																	+																		
Multi-agent MCTS																																			
Ensemble MCTS																																			
HOP											+																								
Sparse UCT													+																						
Info Set UCT							+			+																									
Multiple MCTS																																			
UCT+									+																										
MC$_{\alpha\beta}$																																			
MCCFR									+																										
Reflexive MC																																			
Nested MC																						+			+						+				
NRPA																																			
HGSTS																																			
FSSS, BFS3																																	+		
TAG																									+										
UNLEO																																			
UCTSAT																										+									
ρUCT			+						+									+																	
MRW																									+										
MHSP																											+								
UCB1-Tuned	+																																		
Bayesian UCT																																			
EXP3														+																					
HOOT																									+										
First Play Urgency																																			
(Anti)Decisive Moves																																			
Move Groups																																			
Move Ordering																																			
Transpositions																																			
Progressive Bias	+																																		
Opening Books																																			
MCPG																																			+
Search Seeding																																			
Parameter Tuning																																			
History Heuristic																																			
AMAF													+																						
RAVE													+																						
Killer RAVE																																			
RAVE-max																																			
PoolRAVE																																			
MCTS-Solver	+																																		
MC-PNS																																			
Score Bounded MCTS																																			
Progressive Widening																															+				
Pruning				+																															
Contextual MC																																			
Fill the Board																																			
MAST, PAST, FAST																																			
Simulation Balancing																																			
Last Good Reply																																			
Patterns																																			
Score Bonus																																			
Decaying Reward																							+												
Leaf Parallelisation																																			
Root Parallelisation																																			
Tree Parallelisation																																			
UCT-Treesplit																																			

TABLE 4
Summary of MCTS variations and enhancements applied to other domains.