# An Analysis of Monte Carlo Tree Search

**Steven James[*], George Konidaris[†] & Benjamin Rosman[*‡]**
[*]University of the Witwatersrand, Johannesburg, South Africa
[†]Brown University, Providence RI 02912, USA
[‡]Council for Scientific and Industrial Research, Pretoria, South Africa
steven.james@students.wits.ac.za, gdk@cs.brown.edu, brosman@csir.co.za

**Abstract**

Monte Carlo Tree Search (MCTS) is a family of directed search algorithms that has gained widespread attention in recent years. Despite the vast amount of research into MCTS, the effect of modifications on the algorithm, as well as the manner in which it performs in various domains, is still not yet fully known. In particular, the effect of using knowledge-heavy rollouts in MCTS still remains poorly understood, with surprising results demonstrating that better-informed rollouts often result in worse-performing agents. We present experimental evidence suggesting that, under certain smoothness conditions, uniformly random simulation policies preserve the ordering over action preferences. This explains the success of MCTS despite its common use of these rollouts to evaluate states. We further analyse non-uniformly random rollout policies and describe conditions under which they offer improved performance.

## Introduction

Monte Carlo Tree Search (MCTS) is a general-purpose planning algorithm that has found great success in a number of seemingly unrelated applications, ranging from Bayesian reinforcement learning (Guez, Silver, and Dayan 2013) to General-Game Playing (Finnsson and Björnsson 2008). Originally developed to tackle the game of Go (Coulom 2007), it is often applied to domains where it is difficult to incorporate expert knowledge. MCTS combines a tree search approach with Monte Carlo simulations (also known as rollouts), and uses the outcome of these simulations to evaluate states in a lookahead tree. It has also shown itself to be a flexible planner, recently combining with deep neural networks to achieve superhuman performance in Go (Silver et al. 2016).

While many variants of MCTS exist, the Upper Confidence bound applied to Trees (UCT) algorithm (Kocsis and Szepesvári 2006) is widely used in practice, despite its shortcomings (Domshlak and Feldman 2013). A great deal of analysis on UCT revolves around the tree-building phase of the algorithm, which provides theoretical convergence guarantees and upper-bounds on the regret (Coquelin and Munos 2007). Less, however, is understood about the simulation phase.

UCT calls for rollouts to be performed by randomly selecting actions until a terminal state is reached. The outcome of the simulation is then propagated to the root of the tree. Averaging these results over many iterations performs remarkably well, despite the fact that actions during the simulation are executed completely at random. As the outcome of the simulations directly affects the entire algorithm, one might expect that the manner in which they are performed has a major effect on the overall strength of the algorithm.

A natural assumption to make is that completely random simulations are not ideal, since they do not map to realistic actions. A different approach is that of so-called *heavy rollouts*, where moves are intelligently selected using domain-specific rules or knowledge. Counterintuitively, some results indicate that using these stronger rollouts can actually result in a decrease in overall performance (Gelly and Silver 2007).

We propose that the key aspect of a rollout policy is the way in which it ranks the available actions. We demonstrate that under certain smoothness conditions, a uniformly random rollout policy preserves the optimal action ranking, which in turn allows UCT to select the correct action at the root of the tree.

We also investigate the effect of heavy rollouts. Given that both heavy and uniformly random policies are suboptimal, we are interested in the reason these objectively stronger rollouts can often negatively affect the performance of UCT. We show that heavy rollouts can indeed improve performance, but identify low-variance policies as potentially dangerous choices, which can lead to worse performance.

There are a number of conflating factors that make analysing UCT in the context of games difficult, especially in the multi-agent case: the strength of our opponents, whether they adapt their policies in response to our own, and the requirement of rollout policies for multiple players. Aside from Silver and Tesauro (2009) who propose the concept of simulation balancing to learn a Go rollout policy that is weak but "fair" to both players, there is little to indicate how best to simulate our opponents. Furthermore, the vagaries of the domain itself can often add to the complexity—Nau (1983) demonstrates how making better decisions throughout a game does not necessarily result in the expected increase in winning rate. Given all of the above, we simplify matters by restricting our investigation to the single-agent case.

## Background

In this paper we consider only environments that can be modelled by a finite Markov Decision Process (MDP). An MDP is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ over states $\mathcal{S}$, actions $\mathcal{A}$, transition function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, reward function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, and discount factor $\gamma \in (0, 1)$ (Sutton and Barto 1998).

Our aim is to learn a *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that specifies the probability of executing an action in a given state so as to maximise our expected return. Suppose that after time step $t$ we observe the sequence of rewards $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$ Our expected return $\mathbb{E}[R_t]$ is then simply the discounted sum of rewards, where $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1}$.

For a given policy $\pi$, we can calculate the value of any state $s$ as the expected reward gained by following $\pi$:

$$V^\pi(s) = \mathbb{E}_\pi[R_t \mid s_t = s].$$

A policy $\pi^*$ is said to be optimal if it achieves the maximum possible value in all states. That is, $\forall s \in \mathcal{S}$,

$$V^{\pi^*}(s) = V^*(s) = \max_\pi V^\pi(s).$$

One final related concept is that of the $Q$-value function which assigns a value to an action in a given state under policy $\pi$:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_t \mid s_t = s, a_t = a].$$

Similarly, the optimal $Q$-value function is given by

$$Q^{\pi^*}(s, a) = Q^*(s, a) = \max_\pi Q^\pi(s, a).$$

### Monte Carlo Tree Search

MCTS iteratively builds a search tree by executing four phases (Figure 1). Each node in the tree represents a single state, while the tree's edges correspond to actions. In the selection phase, a child-selection policy is recursively applied until a leaf node[1] is reached.

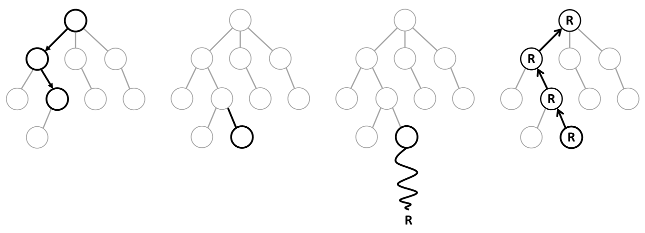


Figure 1: Phases of the Monte Carlo tree search algorithm. A search tree, rooted at the current state, is grown through repeated application of the above four phases.

UCT uses a policy known as UCB1, a solution to the multi-armed bandit problem (Auer, Cesa-Bianchi, and Fischer 2002). At each state $s$, we store the visitation count $n_s$ and average return $\overline{X}_s$. For a given node $s$, the policy then selects child $i$ that maximises the upper confidence bound

$$\overline{X}_i + C_p \sqrt{\frac{2 \ln(n_s)}{n_i}},$$

where $C_p$ is an exploration parameter.

Once a leaf node is reached, the expansion phase adds a new node to the tree. A simulation is then run from this node according to the rollout policy, with the outcome being backpropagated up through the tree, updating the nodes' average scores and visitation counts.

This cycle of selection, expansion, simulation and backpropagation is repeated until some halting criteria is met, at which point the best action is selected. There are numerous strategies for the final action selection, such as choosing the action that leads to the most visited state (*robust child*), or the one that leads to the highest valued state (*max child*). We adopt the *robust child* strategy throughout this paper, noting that previous results have shown little difference between the two (Chaslot et al. 2008).

### Smoothness

There is some evidence to suggest that the key property of a domain is the smoothness of its underlying value function. The phenomenon of game tree pathology (Nau 1982), as well as work by Ramanujan, Sabharwal, and Selman (2011), advance the notion of *trap states*, which occur when the value of two sibling nodes differs greatly—the latter argue that UCT is unsuited to domains possessing many such states. Furthermore, in the context of $\mathcal{X}$-armed bandits (where $\mathcal{X}$ is some measurable space), UCT can be seen as a specific instance of the Hierarchical Optimistic Optimisation algorithm, which attempts to find the global maximum of the expected payoff function using MCTS. Its selection policy is similar to UCB1, but contains an additional term that depends on the smoothness of the function. For an infinitely smooth function, this term goes to 0 and the algorithm becomes UCT (Bubeck et al. 2011).

In defining what is meant by smoothness, one notion that can be employed is that of Lipschitz continuity, which limits the rate of change of a function. Formally, a value function $V$ is $M$-Lipschitz continuous if $\forall s, t \in \mathcal{S}$,

$$|V(s) - V(t)| \leq M d(s, t),$$

where $M \geq 0$ is a constant, $d(s, t) = \|k(s) - k(t)\|$ and $k$ is a mapping from state-space to some normed vector space (Pazis and Parr 2011).

## UCT in Smooth Domains

We tackle the analysis of UCT by first considering only the role of the simulation policy in evaluating states (ignoring the selection and iterative tree-building phase of UCT), and then discussing the full algorithm.

The simulation policy of UCT replaces the classical evaluation function of algorithms such as minimax, which assigns a value to each state. The ultimate purpose of evaluating states is not to calculate their values as accurately as

[1]A leaf node is not to be confused with a terminal state: the former represents a state at the agent's current search horizon, while the latter is any state that does not admit further action by the agent.

possible, but rather to compute the correct action to select. Thus the choice of simulation policy need not be optimal, as long as it preserves the correct preferences over possible actions.

The key idea here is that if we have any value function that is a positive affine transformation of $V^*$, then the action rankings remain correct and the optimal policy follows. In particular, notice that the optimal policy depends only on the most highly ranked action of the optimal Q-value function $Q^*$, and not on the actual values of $Q^*$. Any simulation policy $\pi$ thus induces the optimal policy if $\forall s \in \mathcal{S}$,

$$\operatorname*{argmax}_{a\in\mathcal{A}} Q^{\pi}(s,a) = \operatorname*{argmax}_{a\in\mathcal{A}} Q^{*}(s,a). \tag{1}$$

Under an assumption of a sufficiently smooth optimal value function, we can show that Equation 1 holds for a uniformly random policy. Assume a domain in which rewards are only assigned at terminal states, and whose optimal value function is $M$-Lipschitz. Furthermore, assume that for some state $s$, we have $\epsilon = Q^*(s,a) - Q^*(s,b) > 0$ for actions $a$ and $b$. Let $d_{\max}$ be the largest distance between any pair of states and assume there are $N$ reachable terminal states from the states we transition into after taking either action. Then the Q-value function under the random policy $\tilde{Q}$ is bounded below, *viz.*:

$$Q^*(s,a) - \left(\frac{N-1}{N}\right) M d_{\max} \leq \tilde{Q}(s,a).$$

As $\tilde{Q}(s,b)$ is bounded above by $Q^*(s,b)$, we can preserve the correct action ranking by setting $M < \dfrac{N\epsilon}{d_{\max}(N-1)}$, which ensures that $\tilde{Q}(s,a) > \tilde{Q}(s,b)$.

## Chain walk

As a simple example, consider a deterministic 1-dimensional *chain walk* problem with the following dynamics: states are indexed as integers from 0 to $N$, and the actions available to the agent are `LEFT` and `RIGHT`. Executing these actions decrements or increments the agent's state respectively unless it is at a boundary, in which case it does not move. The task is undiscounted and episodic, with rewards of $-1$ on all transitions. The episode ends when the agent enters the goal state at index $N$.

It is clear that the optimal value function is given by $V^*(s) = (s - N)$, and by solving a second-order recurrence relation we have that for the uniformly random policy $\tilde{\pi}$:

$$V^{\tilde{\pi}}(s) = (N + s + 1)(s - N).$$

Since $V^{\tilde{\pi}}$ is simply a positive affine transform of $V^*$, it follows that greedily selecting an action according to $V^{\tilde{\pi}}$ by performing a one-step lookahead will produce the optimal policy. In other words, policy iteration (Howard 1960) starting from a uniformly random policy converges in one step.

## Grid world

We are interested in the effect of the smoothness of a domain on the efficacy of a uniformly random policy. To this end, we consider *grid world*—the 2-dimensional generalisation of the above. The dynamics of the environment remain the same, but states are indexed by integer coordinates $(x, y)$, while the agent has an additional two actions which we nominally term `UP` and `DOWN`.

To control the smoothness of the domain, we generate random transition functions to be Lipschitz continuous. A transition function $T$ is $M$-Lipschitz if $\forall s, s', t \in \mathcal{S}, a \in \mathcal{A}$,

$$|T(s,a,t) - T(s',a,t)| \leq M \left\| s - s' \right\|_1 .$$

This is achieved by first randomly assigning transitions for state-action pairs, and then altering them to ensure the above equation is satisfied. The probabilities are finally normalised to ensure they sum to 1.

Beginning with a uniformly random policy, we perform one step of policy iteration ($\gamma = 0.95$) on a $10 \times 10$ grid. The resulting policy can then be used to measure how accurately the initial policy evaluated states. As we are only interested in selecting the correct action (and not, say, the values of states), we judge its correctness by the percentage of states in which the policy suggests an optimal action. The results of this experiment are illustrated by Figure 2, and demonstrate that as the bound on the Lipschitz condition increases, the uniformly random policy becomes less and less effective in suggesting the correct action.

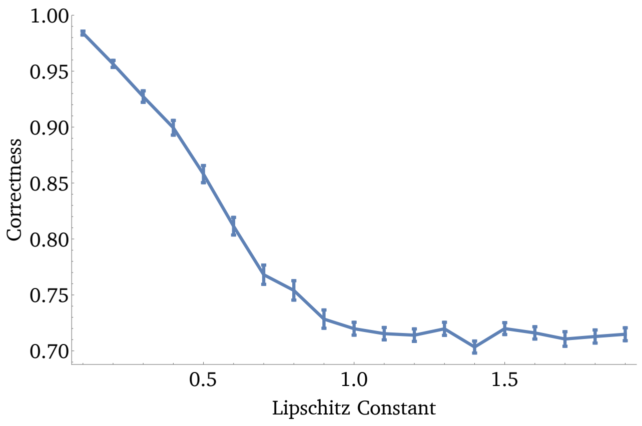


Figure 2: The strength of the policy calculated by a single step of policy iteration as the smoothness in the transition function decreases. Results were averaged over 100 runs.

## Effect of Rollouts on UCT

Having investigated only the Monte Carlo evaluation of states, we now focus on the full UCT procedure. It seems intuitive that if the preferences over all leaf nodes across the entire tree are computed correctly, then the algorithm will produce a correct ordering of actions at the root.

To test this hypothesis, we construct a ternary tree of height 10 which functions as an extensive form game. Rewards in the range $[0, 1]$ are assigned to each leaf node such that the optimal action for any node is to select the left child.

We execute UCT on different instantiations of the domain, allowing 5000 iterations per move. However, instead of simulating actions until a terminal state is reached, our rollout phase directly samples the reward from a Gaussian distribution centred about the true value with varying standard deviation $\sigma$. For small values of $\sigma$, the "simulation" returns a near-optimal estimate for states and actions, and thus preserves the correct action preferences with high probability. With larger standard deviations, the probability of the correct ordering decreases.

Figure 3 plots the percentage of times UCT selects the correct action, as well as how often it maintained the correct action ordering at the root. For illustrative purposes, we plot the probability that a random variable drawn from $\mathcal{N}(1, \sigma)$ is greater than one drawn from $\mathcal{N}(\frac{1}{2}, \sigma)$ in order to show that an increase in standard deviation leads to an increase in the probability of an incorrect ranking (dotted orange line).

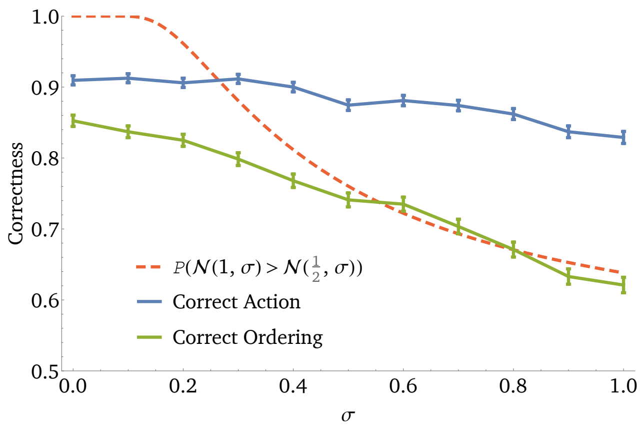


Figure 3: The probability that UCT maintains the correct action ranking at the root node in a ternary tree, averaged over 2000 runs.

## Function Optimisation

Reasoning about the smoothness (or lack thereof) of an MDP is difficult for all but the simplest of domains. To develop some method of controlling and visualising the smoothness, we consider the task of finding the global maximum of a function. Simple, monotonic functions can be seen as representing smooth environments, while complicated ones represent non-smooth domains.

For simplicity, we constrain the domain and range of the functions to be in the interval $[0, 1]$. Each state represents some interval $[a, b]$ within this unit square, with the starting state representing $[0, 1]$. We assume that there are two available actions at each state: the first results in a transition to the new state $[a, \frac{a+b}{2}]$, while the second transitions to $[\frac{a+b}{2}, b]$. This approach forms a binary tree that covers the entire state-space. As this partitioning could continue *ad infinitum*, we truncate the tree at a fixed height by considering a state to be terminal when $b - a \leq 10^{-5}$.

In the simulation phase, actions are executed uniformly randomly until a terminal state is encountered, at which point some reward is received. Let $f$ be the function and $c$ be the midpoint of the state reached by the rollout. At iteration $t$, a binary reward $r_t$, drawn from a Bernoulli distribution $r_t \sim \text{Bern}(f(c))$, is generated.

At the completion of the algorithm, we calculate the score by descending the lookahead tree from root to leaf (where a leaf node is a node that has not yet been fully expanded), selecting at each state its most visited child. The centre of the leaf node's interval represents UCT's belief of the location of the global maximum.

To illustrate UCT's response to smoothness, consider two functions: $f(x) = |\sin \frac{1}{x^5}|$ and

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

Notice that the frequency of the function $f$ decreases as $x$ increases (see Figure 4). Since the function attains a maximum at many points, we can expect UCT to return the correct answer frequently. Visiting an incorrect region of the domain here is not too detrimental, since there is most likely still a state that attains the maximum in the interval.

With that said, there is clearly a smoother region of the space that can be searched. In some sense this is the more conservative space, since a small perturbation does not result in too great a change in value. Indeed, UCT prefers this region, with the leaf nodes concentrating around this smooth area despite there being many optima at $x \leq 0.5$.

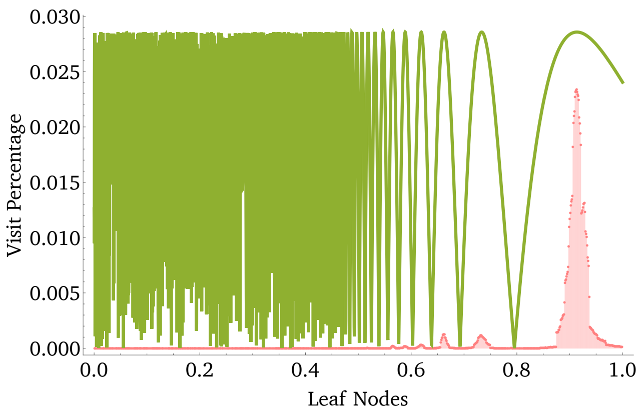


Figure 4: Percentage of visits to leaves after 50 000 iterations of UCT for the function $f$.

On the other hand, the function $g$ is a tougher proposition, despite having the same number of critical points as $f$. Here the "safer" interval of the function's domain (at $x \geq 0.5$) preferred by UCT is now suboptimal. In this case, UCT finds it difficult to make the transition to the true optimal value, since it prefers to exploit the smoother, incorrect region.

After a sufficient number of simulations, however, UCT does indeed start to visit the optimal region of the graph (Figure 5). Since the value of nearby states in this region changes rapidly, robust estimates are required to find the true optimum. For the function $g$, UCT achieves an average score of $0.67 \pm 0.002$—lower than even that of the local maxima. This suggests that the search spends time at the suboptimal maxima before switching to the region $x < 0.5$. However, because most of the search had not focused on this space previously, its estimates are inadequate, resulting in very poor returns.

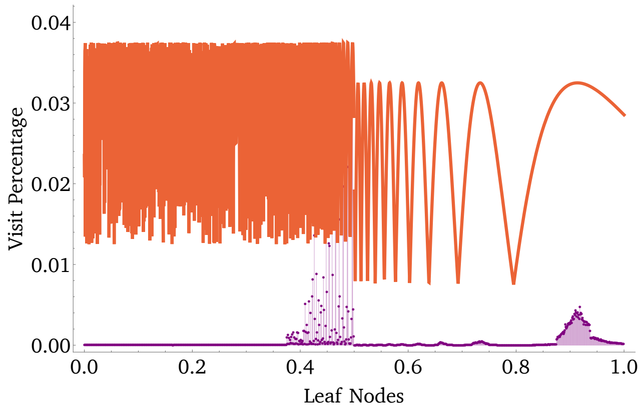


Figure 5: Percentage of visits to leaves after 50 000 iterations of UCT for the function $g$.

These results also suggest that, in practice, a globally smooth assumption is not completely necessary. Instead, a value function that is locally Lipschitz about the optima appears to be sufficient for good performance.

## Bias in the Simulation Phase

Having demonstrated the effect of smoothness on the uniformly random rollouts of UCT, we now turn our attention to heavy rollouts. Oftentimes rollouts that are not uniformly random are referred to as biased rollouts. Since the simulation phase is a substitute for the value function of an as-yet-unknown future policy, almost all policies suffer from some bias, even uniformly random ones. As this applies to both deterministic and random rollouts—policies at opposite ends of the spectrum—it is important to differentiate between the two.

To draw an analogy, consider Bayesian inference. Here a prior distribution, which represents the knowledge injected into the system, is modified by the evidence received from the environment to produce a posterior distribution. Arguments can be made for selecting a maximal entropy prior—that is, a prior that encodes the minimum amount of information. Based on this *principle of indifference*, the posterior that is produced is directly proportional to the likelihood of the data.

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

Uncertainty in a domain arises from the fact that we are unaware of the policy that the agent will utilise in the future. This is especially true beyond the search tree's horizon, where there exist no value estimates. The simulation phase is thus responsible for managing this uncertainty. The choice of rollout policy can therefore be viewed as a kind of prior distribution over the policy space—one which encodes the user's knowledge of the domain, with uniformly random rollouts representing maximal entropy priors, and deterministic rollouts minimal ones.

To illustrate the advantage of selecting a high-entropy simulation policy, we consider adding knowledge to the simulations for the function optimisation task by performing a one-step lookahead and selecting an action proportional to the value of the next state. We also consider an inversely-biased policy which selects an action in inverse proportion to the successor state's value.

The choice of rollout policy affects the initial view MCTS has of the function to be optimised. Figure 6 demonstrates this phenomenon for the random, biased and inversely-biased policies when optimising the function

$$y(x) = \frac{|\sin(5\pi x) + \cos(x))|}{2}.$$

Random rollouts perfectly represent the function, since their expected values depend only on the function's value itself, while the biased policy assigns greater importance to the region about the true maximum, but does not accurately represent the underlying function. This serves to focus the search in the correct region of the space, as well as effectively prune some of the suboptimal regions. This is not detrimental here since the underestimated regions do not contain the global maximum. Were the optimal value to exist as an extreme outlier in the range $[0.5, 1]$, then the policy would hinder the ability of MCTS to find the true answer, as it would require a large number of iterations to correct this error. A sufficiently smooth domain would preclude this event from occurring.

Finally, the rightmost figure demonstrates how an incorrectly biased policy can cause MCTS to focus initially on a completely suboptimal region. Many iterations would thus be required to redress the serious bias injected into the system. Thus while heavy rollouts can offer performance advantages, they can also degrade the accuracy of UCT.

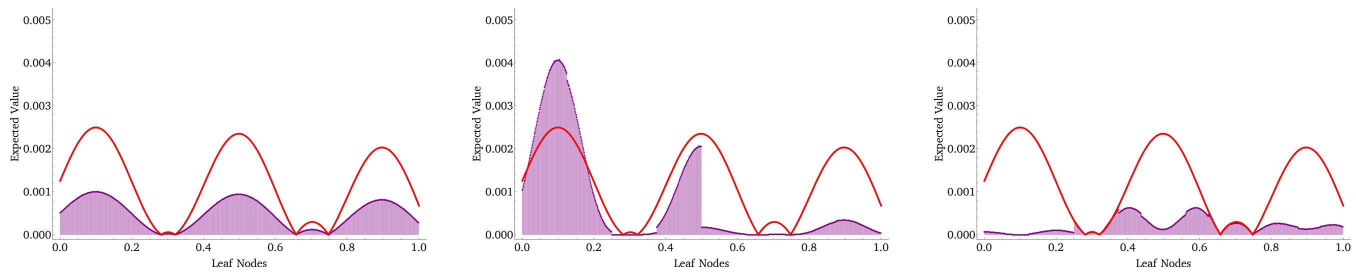
To demonstrate the possible risk in selecting the incorrect simulation policy, consider a perfect $k$-ary tree which represents a generic extensive-form game of perfect information. Vertices represent the states, and edges the actions, so that $\mathcal{A}(s) = \{0, 1, \ldots, k-1\}$. Rewards in the range $[0, 1]$ are assigned to each leaf node such that $\forall s \in \mathcal{S}, \pi^*(s, \lfloor \frac{k}{2} \rfloor) = 1$. For non-optimal actions, rewards are distributed randomly. A $k$-ary tree of height $h$ is referred to as a $[k, h]$ tree henceforth.

A uniformly random rollout policy $\pi_{rand}$ acts as a baseline with which to compare the performance of other simulation policies. These policies sample an action from normal distributions with varying mean and standard deviation—that is, policies are parameterised by $\beta \sim \mathcal{N}(\mu, \sigma)$ such that

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

where the operator $\lfloor \cdot \rceil$ rounds to the nearest integer, away from zero.

Figure 7 presents the results of an experiment conducted on a $[5, 5]$ instance of the domain. We limit the MCTS algorithm to 30 iterations per move to simulate an environment in which the state-space is far greater than what would



(a) Expected value under a uniformly random policy.

(b) Expected value under a biased policy.

(c) Expected value under an inversely-biased policy.

Figure 6: The view of the overlaid function under the different policies, with the expected value calculated by multiplying the probability of reaching each leaf by its value.

be computable given the available resources. Both the mean and standard deviation are incrementally varied from $0$ to $4$, and are used to parameterise a UCT agent. The agent is then tested on 10 000 different instances of the tree.

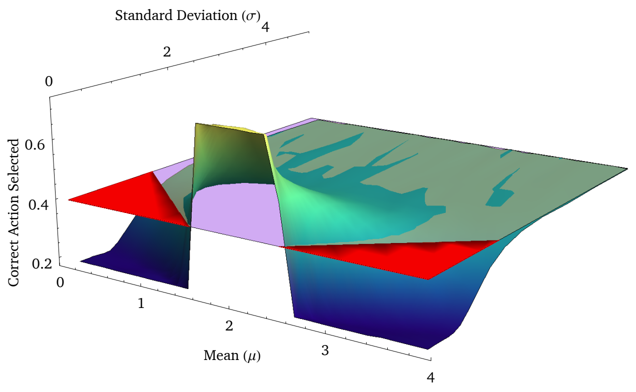


Figure 7: Results of rollout policies averaged over 10 000 $[5, 5]$ games. The $x$ and $y$ axes represent the mean and standard deviation of the rollout policy used by the UCT agent, while the $z$-axis denotes the percentage of times the correct action was returned. The performance of a uniformly random rollout policy is represented by the plane, while the red region indicates policies whose means are more than one standard deviation from the optimal policy.

The results demonstrate that there is room for bettering random rollouts. Quite naturally, the performance of the UCT agent is best when the distribution from which rollout policies are sampled are peaked about the optimal action. However, the worst performance occurs when the rollouts have incorrect bias and are over-confident in their estimation (that is, with small standard deviations), their performance dropping below even that of random. When the rollouts have too great a variance, however, their performance degenerates to that of random. There is thus only a small window for improvement, which requires the correct bias and low variance. Similar results have been demonstrated by Veness, Lanctot, and Bowling (2011), where a reduction in variance can dramatically decrease the overall error when the bias is not too great. One should be certain of the correct bias, however, as the major risk of failure occurs for low-variance, high-bias distributions.

While the $k$-ary tree is fairly synthetic—the same bias occurs throughout the state-space—we have observed a similar phenomenon in more realistic domains where this is not the case (James, Rosman, and Konidaris 2016).

The above results suggest that the best course of action may be to select uniformly random rollouts and forego the associated risks that come with executing heavy playouts. However, the large variance of uniformly random rollouts renders them ineffective in domains with extremely large state-spaces, which may be as a result of stochasticity or hidden information in the game. In these domains, random trajectories can practically cover only a vanishingly small part of the overall state-space. With so large a variance, these rollouts provide almost no information, and performing a sufficient number of them is simply infeasible. Informed rollouts are therefore sometimes unavoidable.

To show that heavy rollouts can indeed be of great benefit in the correct situation, consider a $[5, 10]$ tree. We compare the uniformly random and optimal rollout policies with a heavy rollout policy, which selects the optimal action with probability $0.6$, and all others with probability $0.1$. The percentage of times UCT returned the optimal action was recorded, with the results given by Figure 8.

Viewing policies as distributions over actions, the heavy rollout biases the uniform distribution by placing greater weight on the optimal action. This leads to performance greater than that of the uniformly random policy for any number of simulations, and convergence to optimal play in a shorter timespan. Thus as long as the error due to the bias of a heavy rollout is less than that of uniformly random, the heavy rollout will offer improved performance.

## Conclusion and Future Work

We have demonstrated that a key consideration of performing rollouts in UCT is not in how accurately it evaluates states, but how well it preserves the correct action ranking. In smooth domains, this preservation often appears to be the case. This explains UCT's remarkably strong performance, despite its use of uninformed simulations to evaluate states.

We have also described how both uniformly random and

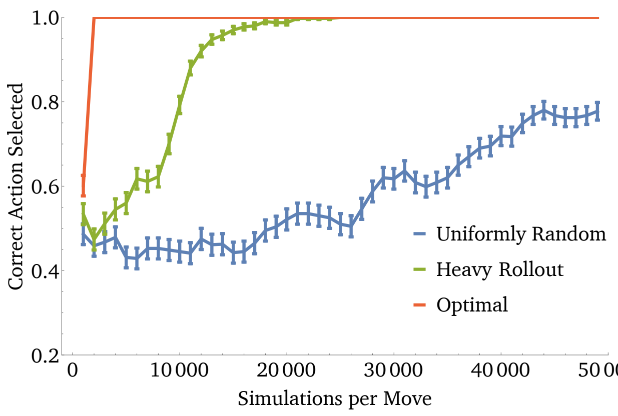


Figure 8: The percentage of correct action decisions made by a UCT agent with various rollout policies in a 5-ary game tree averaged over 400 runs.

heavy rollouts are biased, and identified high-bias, low-variance rollouts as dangerous choices which can result in extremely poor performance. In situations of uncertainty, a higher-variance rollout policy may thus be the better, less-risky choice.

Future work should focus on extending this work to adversarial settings, as well as investigating methods and heuristics for quantifying the smoothness of a particular domain. This would allow us to decide on the type of rollout policy to employ, or even whether UCT is at all appropriate.

## Acknowledgements

The authors wish to thank the anonymous reviewers for their thorough feedback and helpful comments.

## References

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

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

Chaslot, G.; Winands, M.; Herik, H. v. d.; Uiterwijk, J.; and Bouzy, B. 2008. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation* 4(3):343 – 357.

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

Coulom, R. 2007. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, 72–93. Turin, Italy: Springer.

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

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. *Association for the Advancement of Artificial Intelligence* 8:259 – 264.

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

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

Howard, R. 1960. Dynamic programming and Markov processes.

James, S.; Rosman, B.; and Konidaris, G. 2016. An investigation into the effectiveness of heavy rollouts in UCT. In *The IJCAI-16 Workshop on General Game Playing*, 55–61.

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

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

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

Pazis, J., and Parr, R. 2011. Non-parametric approximate linear programming for MDPs. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 459–464. AAAI Press.

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

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

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

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

Veness, J.; Lanctot, M.; and Bowling, M. 2011. Variance reduction in Monte-Carlo tree search. In *Advances in Neural Information Processing Systems*, 1836–1844.