An Investigation into the Effectiveness of Heavy Rollouts in UCT

Steven James[1], **Benjamin Rosman**[1,2] **& George Konidaris**[3]
[1]University of the Witwatersrand, Johannesburg, South Africa
[2]Council for Scientific and Industrial Research, Pretoria, South Africa
[3]Duke University, Durham NC 27708, USA
steven.james@students.wits.ac.za, brosman@csir.co.za, gdk@cs.duke.edu

Abstract

Monte Carlo Tree Search (MCTS) is a family of directed search algorithms that has gained widespread attention in recent years, with its domain-independent nature making it particularly attractive to fields such as General Game Playing. Despite the vast amount of research into MCTS, the dynamics of the algorithm are still not yet fully understood. In particular, the effect of using knowledge-heavy or biased rollouts in MCTS still remains largely unknown, with surprising results demonstrating that better-informed rollouts do not necessarily result in stronger agents. We show that MCTS is well-suited to a class of domains possessing a smoothness property, and that any error due to incorrect bias is compounded in non-smooth domains, particularly for low-variance simulations.

1 Introduction

Monte Carlo Tree Search (MCTS) has found great success in a number of seemingly unrelated applications, ranging from Bayesian reinforcement learning [Guez *et al.*, 2013] to *Ms Pac-Man* [Pepels *et al.*, 2014]. Originally developed to tackle the game of Go [Coulom, 2007] , it is often applied to domains for which only low-quality heuristics exist. MCTS combines a traditional tree search with Monte Carlo simulations (also known as rollouts), and uses the outcome of these simulations to evaluate states in a lookahead tree. That MCTS requires neither expert knowledge nor heuristics makes it a powerful general-purpose approach, particularly relevant to tasks such as General Game Playing [Genesereth *et al.*, 2005]. It has also shown itself to be a flexible planner, recently combining with deep neural networks to achieve super-human performance in the game of Go [Silver *et al.*, 2016].

While many variants of MCTS exist, the UCT (Upper Confidence bound applied to Trees) 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 is known about the simulation phase.

UCT calls for this phase 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 provides a fairly accurate measure of the strength of the initial state, despite the fact that the simulation is completely 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].

While UCT is indeed domain-independent, it cannot be simply seen as a panacea—Ramanujan *et al.* [2011] demonstrates its poor performance in chess, for example. Coupled with the above results regarding heavy playouts, this raises two questions: when is UCT a good choice of algorithm and what is the effect of non-uniformly random rollouts?

There are a number of conflating factors that make analysing UCT in the context of games difficult, especially in the multi-agent case. These include the strength of our opponents and whether they adapt their policies in response to our own, as well as the additional complexity of the rollout phase, which now requires 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 winning rate that should be expected. Given all of the above, we choose to simplify matters by restricting our investigation to the single-agent case.

We propose that a key characteristic of a domain is its smoothness, and then demonstrate that UCT is well-suited to domains possessing this property. We provide results which show that biased rollouts can indeed improve performance, but identify high-bias, low-variance rollout policies as potentially dangerous choices that can lead to worse performance. This is further compounded in non-smooth domains.

2 Background

2.1 Markov Decision Process

A Markov Decision Process (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].

Suppose that after time step t we observe the sequence of rewards $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$ For episodic tasks, a finite number of rewards will be observed. In general, we wish to maximise our expected return $\mathbb{E}[R_t]$, where $R_t = \sum_{i=1}^{N} \gamma^{i-1} r_{t+i}$ represents the discounted sum of the rewards.

A *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a mapping that specifies the probability of executing an action in a given state. For a policy π, the value of a state is the expected reward gained by following π:

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

A policy π^* is optimal if $\forall s \in \mathcal{S}, V^{\pi^*}(s) = \max_{\pi} V^{\pi}(s)$.

2.2 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 is reached.

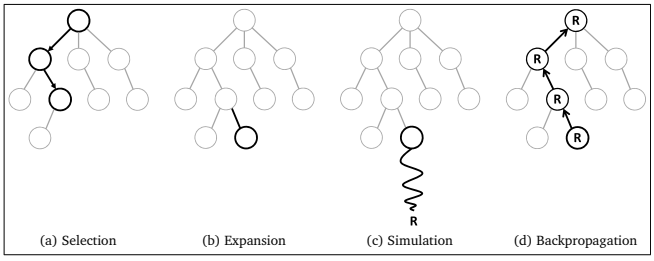


Figure 1: Phases of the Monte Carlo tree search algorithm. A search tree, rooted at the current state, is grown through repeated application of the above four phases.

UCT uses a policy known as UCB1, a well-known solution to the multi-armed bandit problem [Auer *et al.*, 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 playout 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 (usually that which leads to the most visited state) is selected.

2.3 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 *et al.* [2011], advance the notion of *trap states*, which occur when the value of two sibling nodes differs greatly. It is thought 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 vector space.

3 Function Optimisation

Reasoning about the smoothness (or lack thereof) of an MDP is difficult for all but the vaguest of statements. 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 by considering a state to be terminal when $b - a \leq 10^{-5}$.

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

At the completion of the algorithm, we calculate the score by descending the game 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 $f(x) = 4x(1 - x)$ and $g(x) = \max(3.6x(1 - x), 1 - 10|0.9 - x|)$. f has a single global

Function	Number of Simulations						
	500	1000	5000	10000	20000	50000	100000
f	1 ± 0.001	1 ± 0.001	1 ± 0.0	1 ± 0.0	1 ± 0.0	1 ± 0.0	1 ± 0.0
g	0.89 ± 0.002	0.90 ± 0.002	0.94 ± 0.002	0.95 ± 0.002	0.96 ± 0.001	0.96 ± 0.001	0.97 ± 0.001
h	1 ± 0.0	1 ± 0.0	1 ± 0.0	1 ± 0.0	1 ± 0.0	1 ± 0.0	1 ± 0.0
j	0.60 ± 0.012	0.62 ± 0.007	0.65 ± 0.003	0.66 ± 0.003	0.67 ± 0.002	0.67 ± 0.002	0.68 ± 0.002

Table 1: Average maximum value found by UCT for the functions f, g, h and j, averaged over 100 runs.

maximum at $x = 0.5$, while g has a local maximum at the same point, and a global maximum at $x = 0.9$. Despite the fact that both functions are relatively simple, UCT occasionally fails to find the true optimal value, as illustrated by the first two rows of Table 1.

To see why this occurs, consider an additional two functions which are far more complex: $h(x) = |\sin \frac{1}{x^5}|$ and $j(x) = \begin{cases} \frac{1}{2} + \frac{1}{2}|\sin \frac{1}{x^5}| & \text{if } x < \frac{1}{2} \\ \frac{7}{20} + \frac{1}{2}|\sin \frac{1}{x^5}| & \text{if } x \geq \frac{1}{2} \end{cases}$.

Notice that the frequency of the function h decreases as x increases. 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 (Figure 2a), with the leaf nodes concentrating around this smooth area despite there being many optima at $x \leq 0.5$.

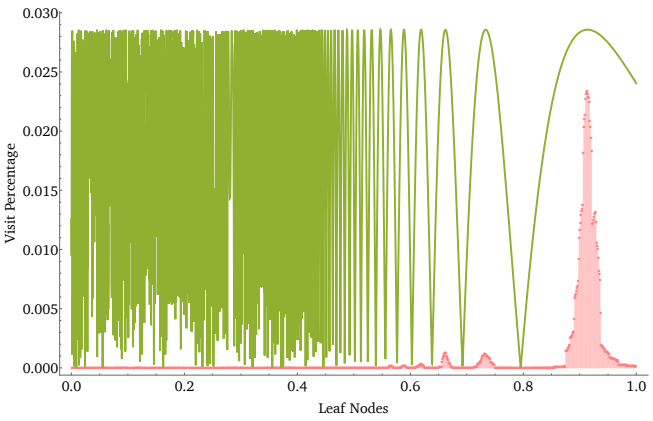
On the other hand, the function j is a tougher proposition, despite having the same number of critical points as h. Here, the "safer" interval of the function's domain (at $x \geq 0.5$) preferred by UCT is now suboptimal. In this case, UCT finds it difficult to make the transition to the true optimal 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 2b). Since the value of nearby states in this region changes rapidly, robust estimates are required to find the true optimum. For function j, UCT achieves an average score 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, which results in very poor returns.

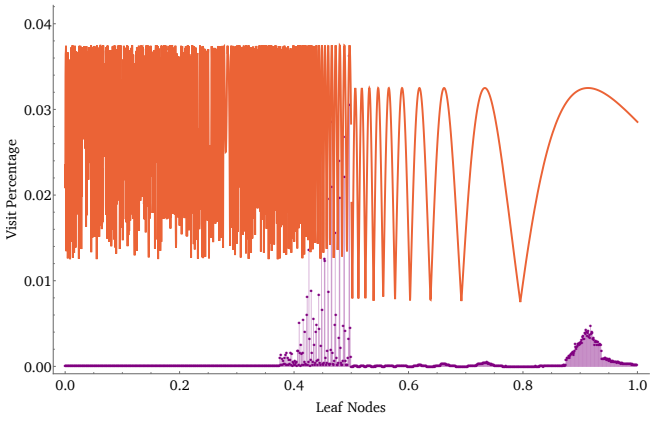
4 Bias in the Simulation Phase

Having demonstrated the effect of the domain's smoothness on 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, 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



(a) Percentage of visits to leaves after 50 000 iterations of UCT for the function h.



(b) Percentage of visits to leaves after 50 000 iterations of UCT for the function j.

Figure 2: The percentage of total visits assigned to each leaf node, averaged over 100 runs. A scaled version of h and j is overlaid for reference. For illustration purposes, leaves are grouped into 1000 buckets, with the sum of the visits to leaves in each bucket plotted.

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 need to be observed to change it significantly. Thus, a prior with low entropy can effectively overwhelm the evidence received from the environment. If such a prior is incorrect, this can result in a posterior with a large degree of bias.

Uncertainty in a domain arises from the fact that we are unaware of the true policy being used by an agent. 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 extreme 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 biasing 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 its value.

The choice of rollout policy affects the initial view MCTS has of the function to be optimised. The figures in Figure 3 demonstrate this phenomenon for the random, biased and inversely-biased policies when optimising the function $y(x) = \frac{|\sin(5\pi x) + \cos(x))|}{2}$.

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 last 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.

5 Risky Simulation Policies

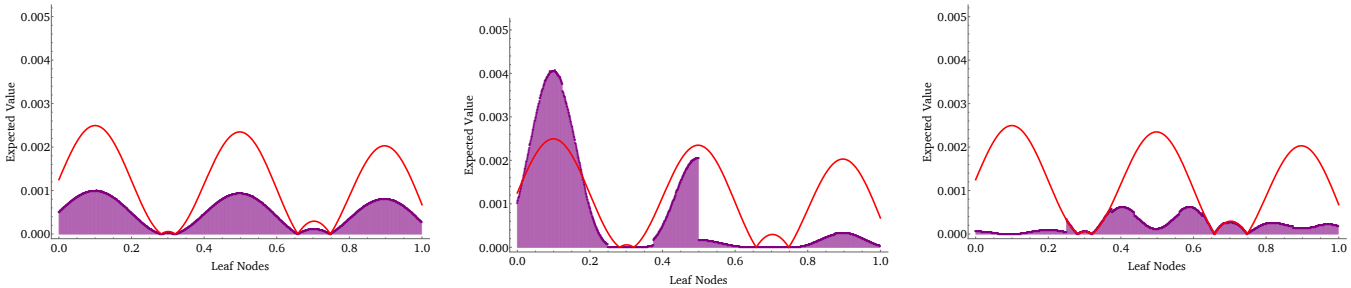
To illustrate the possible risk in selecting the incorrect simulation policy, consider a perfect k-ary tree which represents a generic extensive-form game of perfect information. Vertices represent the state-space, and edges the action-space, so that $\mathcal{A}(s) = \{0, 1, \ldots, k-1\}$. Rewards in the range $[0, 1]$ are assigned to each leaf node such that $\forall s, \pi^*(s, \lfloor \frac{k}{2} \rfloor) = 1$. 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 π_{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

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

Figure 4 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 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.

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



(a) Expected value under a uniformly random policy.

(b) Expected value under a biased policy.

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

Figure 3: The view of the overlaid function under the different policies. The expected value is calculated by multiplying the probability of reaching each leaf by its value.

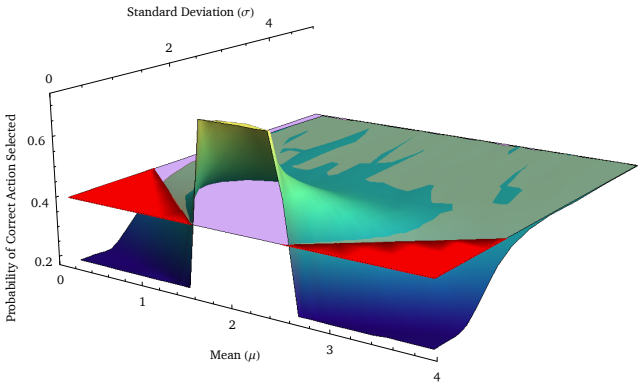


Figure 4: Results of rollout policies averaged over 10 000 $[5, 5]$ games. The x and y axes represent the mean and standard deviation of the rollout policy used by the UCT agent, while the z-axis denotes the percentage of times the correct action was returned. The performance of a uniformly random rollout policy (which returned the correct move 39.4% of the time) is represented by the plane, while the red region indicates policies whose means are more than one standard deviation from the optimal policy.

is, with small standard deviations), their performance dropping below even that of random. When the rollouts have too great a variance, their performance degenerates to that of random. There is thus only a small window for improvement, which requires the correct bias and low variance. One should be certain of the correct bias, however, as the major risk of failure occurs for low-variance, high-bias distributions.

6 Heavy Rollouts in Non-Smooth Domains

One interesting question is the manner in which the rollouts allow UCT to handle noise or unexpected encounters in a domain. To investigate this, we consider a maze domain with deterministic transition dynamics in which an agent navigates a grid (the start and end squares are randomly assigned). A number of obstacles may be placed on the grid according to two strategies. The first is to simply place obstacles randomly, while the second is to form a cluster of obstacles. Informally, clustered obstacles only affect an isolated region of the state-space, while the randomly placed ones affect the entire space. The clustered obstacles therefore create a localised non-smooth region, whereas the randomly placed obstacles make the entire space non-smooth.

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

In order to create a policy for this domain, each action is first assigned a base value of 1. Actions that lead to states

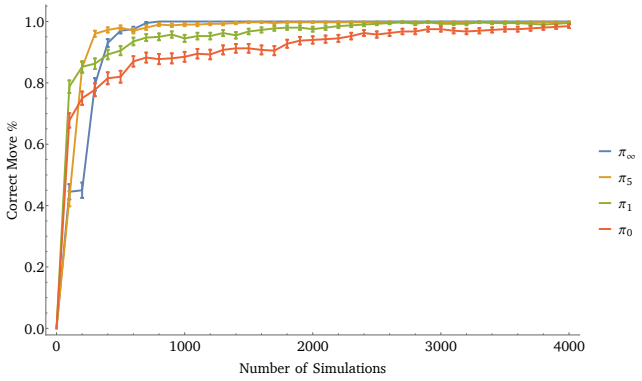


Figure 5: Results of various rollout policies in a 10×10 maze with no obstacles, averaged over 400 different instances of the domain.

closer to the goal (ignoring obstacles) have some additional weight added to them. An action is then selected proportionally to its assigned value.

We parameterise the policies by the value that is added to these actions. For instance, π_0 represents a uniformly random policy, while π_∞ is a deterministic greedy policy. We test the performances of four policies (π_0, π_1, π_5, π_∞) in a 10×10 grid with no obstacles, 15 obstacles and 15 clustered obstacles, with results presented in Figures 5, 6 and 7 respectively.

With no obstacles, the results are fairly straightforward. The greedy policy, which in this case is also the optimal policy, is the most successful, the random the least and the others in between. When obstacles are added randomly, the situation changes completely. Since the rollout policies were constructed to head towards the goal without knowledge of any obstacles, their presence damages the performance of UCT. The worst performing agent in this case is the deterministic policy, while the more conservatively biased policy is the best choice. Random also remains unaffected by the obstacles, with little difference between it and the biased policies.

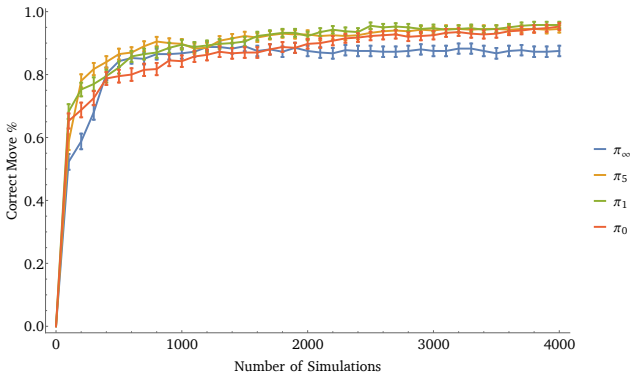


Figure 6: Results of various rollout policies in a 10×10 maze with 15 obstacles randomly scattered, averaged over 400 different instances of the domain.

When the obstacles are clustered together, they only affect a single region of the state-space. The results under these con-

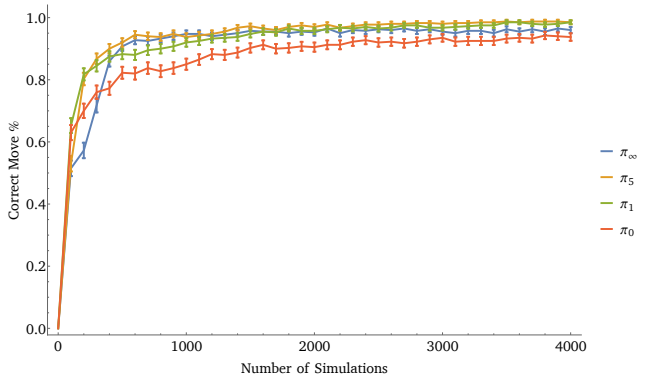


Figure 7: Results of various rollout policies in a 10×10 maze with 15 obstacles clustered together, averaged over 400 different instances of the domain.

ditions are therefore not as drastic as the randomly placed obstacles. Although the deterministic policy again suffers somewhat, it is not to the same extent as previously.

These results suggest that the random and more conservatively biased policies are resilient to unexpected events or noise. Plotting the performance of the policies with an increase in randomly placed obstacles reveals just that (Figure 8). Random rollouts are unaffected by the presence of any number of obstacles, while the dropoff in performance of the other policies is inversely proportional to their level of stochasticity. This speaks to the dangers of a high-bias, low-variance policy. In domains where a good policy cannot be constructed, these results suggest using a higher-variance policy to mitigate against any noise or unforeseen pitfalls.

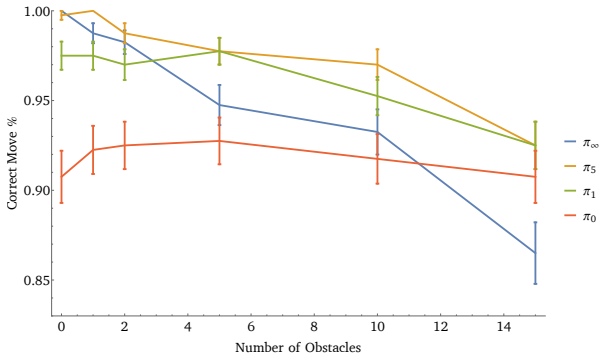
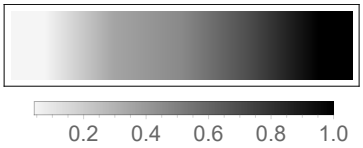
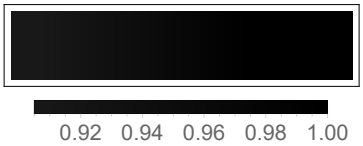


Figure 8: Results of various rollout policies in a 10×10 maze, averaged over 400 different instances of the domain. Agents were restricted to 2000 iterations per move.

One additional point of interest is the poor initial performance of π_∞, even when it is indeed the optimal policy. This occurs because the optimal policy actually provides less information than the random rollout, which better disambiguates the action values. To illustrate, consider a 1×5 grid, where the goal state is the rightmost square and the only available actions are `LEFT` and `RIGHT`. Since the values of states are so close to one another under the optimal policy, UCT is required to explore for a longer period of time, resulting in poor performance at the beginning. The random policy, on the other hand, clearly differentiates between adjacent states, allowing UCT to begin exploiting much earlier. Figure 9 demonstrates this phenomenon.



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



(b) Value of states under the optimal policy.

Figure 9: Value of states under uniformly random and optimal policies, linearly mapped from $[-30, 0]$ to $[0, 1]$. The differentiation between states under the random policy is clearly evident, but not so in the optimal policy's case. Note, too, the difference in the scales of the figures.

As a final experiment, consider the Taxi domain. Here the agent has two additional actions (`PICKUP` and `DROPOFF`), which need to be executed at the appropriate state (the agent incurs a penalty of -10 otherwise). The agent's aim is to navigate to some state and execute the `PICKUP` action, before proceeding to a final state and executing `DROPOFF`.

Initially, it may seem as if we can expect similar results, since this domain can be seen as two sequential instances of the maze task. However, the key difference is the critical requirement of executing a single action in a single state. Thus while the obstacles provide some additional difficulty, they pale in comparison with the bottleneck of having to execute these critical actions at the correct time.

Adopting the previous approach, Figure 10 illustrates that the number of randomly placed obstacles has minimal effect on the lower variance policies when compared with Figure 8—executing `PICKUP` and `DROPOFF` at the proper time is evidently far more important than the presence of the obstacles.

The domain is therefore indicative of many games where selecting the correct action at critical times is more important than playing well at all other times. This also supports the approach of rollout policies such as that of the Go agent MoGo, which is hardcoded to make critical moves when necessary, but plays randomly otherwise [Wang and Gelly, 2007].

7 Conclusion and Future Work

We have shown that the smoothness of a domain is key to deciding whether to apply the UCT algorithm to the problem. This supports prevailing theories regarding the reason for its poor performance in chess, despite its strong showing in Go.

We have also demonstrated that selecting a low-variance policy can markedly improve the performance of UCT in

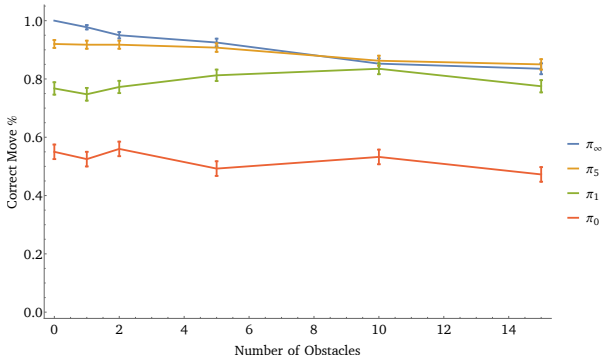


Figure 10: Results of various rollout policies in a 10×10 Taxi domain, averaged over 400 different instances of the domain. Agents were restricted to 2000 iterations per move.

the correct situation, especially in those situations that require a critical action to be selected at the correct state, but can also result in extremely poor performance exacerbated by non-smooth domains. In situations of uncertainty, a higher-variance rollout policy may thus be the better, less-risky choice. When it comes to learning simulation policies, this suggests that learning a distribution over policies may be superior to any kind of pointwise approach.

Future work should also investigate methods and heuristics for quantifying the smoothness of a particular domain. One approach may be first to construct a game tree using uniformly random rollouts. The tree could then be analysed by estimating the smoothness of sibling nodes' values throughout. This can be achieved by assigning a smoothness score to a node, calculated as a linear combination of the lag-one autocorrelation of its children's values and respective smoothness scores. This value would indicate the smoothness of the tree (or subtree) and provide information as to UCT's likely performance beforehand.

Acknowledgements

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

References

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

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

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

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

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

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

M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62, 2005.

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

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

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

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

T. Pepels, M.H.M. Winands, and M. Lanctot. Real-time Monte Carlo tree search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, 2014.

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

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

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

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

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