Monte Carlo Tree Search in Continuous Action Spaces with Execution Uncertainty

Timothy Yee, Viliam Lisý, Michael Bowling
Department of Computing Science University of Alberta
Edmonton, AB, Canada T6G 2E8
{tayee, lisy, bowling}@ualberta.ca

Abstract

Real world applications of artificial intelligence often require agents to sequentially choose actions from continuous action spaces with execution uncertainty. When good actions are sparse, domain knowledge is often used to identify a discrete set of promising actions. These actions and their uncertain effects are typically evaluated using a recursive search procedure. The reduction of the problem to a discrete search problem causes severe limitations, notably, not exploiting all of the sampled outcomes when evaluating actions, and not using outcomes to help find new actions outside the original set. We propose a new Monte Carlo tree search (MCTS) algorithm specifically designed for exploiting an execution model in this setting. Using kernel regression, it generalizes the information about action quality between actions and to unexplored parts of the action space. In a high fidelity simulator of the Olympic sport of curling, we show that this approach significantly outperforms existing MCTS methods.

1 Introduction

Many real world problems involve selecting sequences of actions from a continuous space of actions. Examples include choosing target motor velocities in robot navigation; choosing the angle, offset, and speed to hit a billiard ball; or choosing the angle, velocity, and rotation to throw a curling stone. Execution of these actions is often fundamentally uncertain due to limited human or robot skill and the stochastic nature of physical reality. In this paper, we focus on algorithms for choosing good actions in continuous action, continuous state, stochastic planning problems when a model of the execution uncertainty is known.

One often-used approach [Smith, 2007; Archibald *et al.*, 2009; Yamamoto *et al.*, 2015] for such challenging planning problems is to address the continuous action space by using domain knowledge to identify a small, discrete set of candidate actions. Then, the continuous space of stochastic outcomes is sampled for each action. Finally, for each sampled outcome, a heuristic function (possibly preceded by a very shallow search) is used to evaluate the outcomes and thus the original candidates. This approach reveals a tension between exploring a larger set of candidate actions to increase the probability that a good action is considered, and more accurately evaluating promising candidates through deeper search or more execution outcomes, increasing the probability the best candidate is selected. Monte Carlo tree search (MCTS) methods, such as UCT [Kocsis and Szepesvari, 2006], are well suited for balancing this sort of tradeoff, however, many of the successful variants and enhancements are designed for finite, discrete action spaces. A number of recent advances have sought to address this shortcoming. The classical approach of progressive widening (or unpruning) [Coulom, 2007; Chaslot *et al.*, 2008] can handle continuous action spaces by considering a slowly growing discrete set of sampled actions. cRAVE [Couëtoux *et al.*, 2011] combines this with a modification of the RAVE heuristic [Gelly and Silver, 2011] to do generalization from similar (but not exactly the same) actions. HOOT [Mansley *et al.*, 2011] replaces the UCB algorithm in UCT with HOO [Bubeck *et al.*, 2011], an algorithm with theoretical guarantees in continuous action spaces. However, none of these methods make use of one critical insight: *samples of execution uncertainty from a particular action provide information about any action that could have generated that execution outcome.*

We use this insight to propose a novel variant of Monte Carlo tree search, KR-UCT (Kernel Regression UCT), designed specifically for reasoning about continuous actions with execution uncertainty. Instead of evaluating only a discrete set of candidate actions, the algorithm considers the entire continuous space of actions, with candidates acting only as initialization. The core of our approach is in the use of kernel regression to generalize action value estimates over the entire parameter space, with the execution uncertainty model as its generalization kernel. KR-UCT distinguishes itself in a number of key ways. First, it allows information sharing between all actions under consideration. Second, it can identify actions outside of the initial candidates for further exploration by combining kernel regression and kernel density estimation to optimize an exploration-exploitation balance akin to the popular UCB formula [Auer *et al.*, 2002]. Third, it can ultimately select actions outside of the candidate set allowing it to improve on less-than-perfect domain knowledge.

We evaluate KR-UCT in a high fidelity simulation of the Olympic sport of curling. Curling is an example of a challenging action selection problem with continuous actions, continuous stochastic outcomes, sequential decisions, execution uncertainty, and the added challenge of an adversary. We show that the proposed algorithm significantly outperforms existing MCTS techniques. The improvement is apparent not only at short horizons, which allows exploring a large number of different shots, but also at long horizons when evaluating only tens of samples of execution outcomes. Furthermore, we show that existing MCTS improvements, such as RAVE and progressive widening do not improve standard UCT as significantly as KR-UCT in this domain.

2 Background

We begin by describing the core algorithms that KR-UCT will build upon, along with the main building blocks of the competitors used our evaluation.

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a simulation-based search approach to planning in finite-horizon sequential decision-making settings. The core of the approach is to iteratively simulate executions from the current state to a terminal state, incrementally growing a tree of simulated states (nodes) and actions (edges). Each simulation starts by visiting nodes in the tree, selecting which actions to take based on a *selection function* and information maintained in the node. Consequently, it transitions to a successor state. When a node is visited whose immediate children are not all in the tree, the node is *expanded* by adding a new leaf to the tree. Then, a *rollout policy* (e.g., random action selection) is applied from the new leaf to a terminal state. The value of the terminal state is then returned as the value for that new leaf and the information stored in the tree is updated. In the simplest case, a tree with height 1, MCTS starts with an empty tree and adds a single leaf each iteration.

The most common selection function for MCTS is Upper Confidence Bounds Applied to Trees (UCT) [Kocsis and Szepesvari, 2006]. Each node maintains the mean of the rewards received for each action, $\bar{v}_a$, and the number of times each action has been used, n_a. It first uses each of the actions once and then decides what action to use based on the size of the one-sided confidence interval on the reward computed based on the Chernoff-Hoeffding bound as:

$$\underset{a}{\operatorname{argmax}}\, \bar{v}_a + C\sqrt{\frac{\log \sum_b n_b}{n_a}} \tag{1}$$

The constant C controls the exploration-exploitation tradeoff and is typically tuned for the specific domain.

The success of MCTS across a wide range of domains has inspired many modifications and improvements. One of the most notable is Rapid Action Value Estimation (RAVE) [Gelly and Silver, 2011]. It allows learning about multiple actions from a single simulation, based on the intuition that in many domains, such as Go, an action that is good when taken later in the sequence is likely to be good right now as well. RAVE maintains additional statistics about the quality of actions regardless of where they have been used in a subtree. These statistics are then added to action values in the selection function (e.g., UCT) as an additional term with relative weight decreasing with more simulations.

2.2 Progressive Widening

Most selection functions in MCTS, including UCT, require trying every action once. So, obviously, they are not directly applicable in continuous action spaces. Even if the action space is finite, but very large, having too many options can result in a very shallow lookahead. The same solution to this problem was independently introduced in [Coulom, 2007] as *progressive widening* and [Chaslot *et al.*, 2008] as progressive unpruning. It artificially limits the number of actions evaluated in a node by MCTS based on the number of visits to the node. Only after the quality of the best available action is estimated sufficiently well, additional actions are taken into consideration. The order of adding the actions could be done randomly or by exploiting domain knowledge.

If a domain includes stochastic outcomes, such as being the result of execution uncertainty, the outcomes are commonly represented by chance nodes in the search tree. If the set of outcomes is finite and small, then the next state can be sampled from the known probability distribution over the outcomes. If the possible outcomes are large or even continuous then one can simply sample a small number of outcomes [Kearns *et al.*, 2002] or slowly grow the number of sampled outcomes as the node is repeatedly visited, in the same way that progressive widening grows the number of actions [Couëtoux *et al.*, 2011].

UCT assures that the tree grows deeper more quickly in the promising parts of the search tree. The progressive widening strategies add that it also grows wider in the same parts of the search tree.

2.3 Kernel Regression

Kernel regression is a nonparametric method for estimating the conditional expectation of a real-valued random variable from data. In its simplest form [Nadaraya, 1964; Watson, 1964], it estimates the expected value of a point as an average of the values of all points in the data set, weighted based on a typically non-linear function of the distance from the point. The function defining the weight given a pair of points is called the kernel and further denoted K. For a data set $(x_i, y_i)_{i=0}^{n}$, the estimated expected values is:

$$\mathbb{E}(y|x) = \frac{\sum_{i=0}^{n} K(x, x_i) y_i}{\sum_{i=0}^{n} K(x, x_i)}. \tag{2}$$

The kernel is typically a smooth symmetric function, such as the Gaussian probability density function. However, asymmetric kernel functions can also be used in well-motivated cases (e.g., [Michels, 1992]). An important quantity related to kernel regression is kernel density, which quantifies the amount of relevant data available for a specific point in the domain. For any x, the kernel density is defined to be the denominator in Equation 2:

$$W(x) = \sum_{i=0}^{n} K(x, x_i). \tag{3}$$

Algorithm 1 Kernel Regression UCT

```
1:  procedure KR-UCT(state)
2:      if $state$ is terminal then
3:          return utility(state), false
4:      expanded ← false
5:      $A \leftarrow$ actions considered in state
6:      action ← $\text{argmax}_{a \in A} \mathbb{E}(v|a) + C\sqrt{\frac{\log \sum_{b \in A} W(b)}{W(a)}}$
7:      if $\sqrt{\sum_{a \in A} n_a} < |A|$ then
8:          newState ← child of state by taking action
9:          $rv$, expanded ← KR-UCT(newState)
10:     if not expanded then
11:         newAction ≈ $\text{argmin}_{K(action,a) > \tau} W(a)$
12:         add $newAction$ to $state$
13:         newState ← child of state by taking newAction
14:         add initial actions to newState
15:         $rv \leftarrow$ ROLLOUT(newState)
16:     Update $\bar{v}_{action}, n_{action}$, and KR with $rv$
17:     return $rv$, true
```

3 Kernel Regression UCT

The main idea behind KR-UCT is to increase the number of children (i.e., actions) of a node as it is repeatedly visited, while enabling information sharing between similar actions through kernel regression. All evaluated actions, both initially provided candidates and progressively added actions, together with their estimated values, form a single dataset at each node for kernel regression. For the kernel function we use the probability density function of the known model of execution uncertainty, i.e., $K(x, x')$ is the model's probability that action x' is executed when x is the intended action. Equation 2 then can be viewed as a simple Monte Carlo estimate of the integral corresponding to that evaluated action's expected value. This particular kernel regressor is then used to estimate the two key quantities required by UCT. For the average of the action, $\bar{v}_a$, we use the kernel regression estimate. For the number of visits, n_a, we use the kernel density estimate of data coverage. Note that if we had a discrete action space and used a Dirac delta as the kernel (so $K(x, x') = 1_{x=x'}$), then this approach is identical to vanilla UCT.

The pseudocode for KR-UCT is presented as Algorithm 1. The structure of the algorithm follows the standard four steps of MCTS (selection, expansion, simulation, and backpropogation). The main recursive procedure, KR-UCT, is called on the root of the search tree a fixed number of times determined by a per-decision computation budget. It returns the value of the terminal node reached and a boolean flag indicating if the expansion step has already occurred. We explore some algorithmic details of each step below.

Selection. At each decision node, our algorithm uses an adaptation of the UCB formula to select amongst already explored outcomes to be further refined (line 6). Each node maintains the number of visits, n_b, and the mean utility, $\bar{v}_b$, for each outcome (child node) b. As outcomes are revisited throughout the iterations their mean and visit counts are updated. We incorporate such multiple samples of the same outcome into our kernel regression and kernel density estimates by effectively counting each sample as its own data point.

$$\mathbb{E}(v|a) = \frac{\sum_{b \in A} K(a,b)\bar{v}_b n_b}{\sum_{b \in A} K(a,b) n_b} \quad (4)$$

$$W(a) = \sum_{b \in A} K(a,b) n_b. \quad (5)$$

These estimates are then plugged into their respective roles of the UCB formula, $\mathbb{E}(v|a)$ for $\bar{v}_a$ and $W(a)$ for n_a. The UCB formula is then used to select the action to refine further. As usual, the UCB exploration bias is considered infinite if the denominator in the term is 0 (i.e., $W(a) = 0$). The scaling constant C serves the same role as in vanilla UCT, controlling the tradeoff between exploring less visited actions and refining the value of more promising actions. It should be experimentally tuned for each specific domain.

Expansion. After selecting the action with the highest UCB value, the algorithm continues by either improving the estimated value of the action by recursing on its outcome (lines 8-9), or improving the estimated value of the action by adding a new outcome as a new child node (lines 11-15). This decision is based on keeping the number of outcomes in a node bounded by some sublinear function of the number of visits to the node (line 7), just as in progressive widening. In the event of reaching a terminal node within the tree, a new outcome is always added at the leaf's parent (line 10). When adding a new outcome, we want to choose an outcome b that will be heavily weighted by the kernel (i.e., $K(a, b)$ is large), yet is not well represented by the current set of outcomes (i.e., $W(b)$ is small). We achieve this balance by finding an outcome that minimizes the kernel density estimate within the set of all outcomes whose kernel weight is at least a threshold τ (line 11). For computational efficiency, we approximate this optimization by choosing k outcome samples from the execution uncertainty given the chosen action, and select from these outcomes the one with minimal kernel density. After identifying a new outcome, this state is added as a child of the current node with some generated set of initial actions, possibly determined using domain-specific knowledge (lines 12-14).

Simulation and Backpropagation. When a new outcome is added to the tree, a complete simulation is executed to reach a terminal state using some rollout policy (line 15). As in vanilla MCTS, the rollout policy can involve random action selection, a fast decision policy learned from data [Silver *et al.*, 2016] or based on hand-coded rules [Gelly *et al.*, 2006], or even a static heuristic evaluation. The value of the resulting terminal state is then used to update $\bar{v}_b$ and n_b of the outcomes along the selected path through the tree. Finally, for the sake of efficiency, the kernel density and value estimates at actions can be updated incrementally during the backpropagation step.

Final Selection. After the computational budget is exhausted, we still need to use the results of the search to select an action to execute. It is common to use the most sampled action or the action with the highest sampled mean at the root.

Instead, we select the action at the root with the greatest *lower confidence bound* (LCB) using our kernel regression and kernel density estimates:

$$\operatorname*{argmax}_{a \in A} \mathbb{E}(v|a) - C\sqrt{\frac{\log \sum_{b \in A} W_b}{W_a}}. \qquad (6)$$

The constant C acts as a tradeoff between choosing actions that have been sampled the most (i.e., are the most certain) and actions that have high estimated value. This constant does not have to be the same as the constant used in the upper confidence bound calculation.

Computational Complexity. Maintaining the value estimates $E(v|a)$ and kernel densities $W(a)$ for the set of actions A tracked by a node can be done incrementally in time linear in $|A|$ in each iteration. If we store also the sum $\sum_{b \in A} W_b$ in the nodes, selection can be performed in a single pass over the set A. The most expensive operation is expansion, namely finding the minimal kernel density close to an action. With a naïve implementation, evaluating each point in the parameter space takes $O(|A|)$ operations; hence if the optimization evaluates k points, it takes $O(k|A|)$ operations. This complexity can be in practice substantially reduced by storing the actions in a spatial index, such as R-trees [Guttman, 1984], but if all A actions are close to the selected shot, the worst case complexity stays the same.

4 Experimental Evaluation

We validate our algorithm using the Olympic sport of curling.

4.1 Curling

Curling is a two team game, where the goal is to slide (throw) stones, also known as rocks, down a sheet of ice towards the scoring area or *house*, a circular area centered on the *button*. Games are divided into rounds called *ends*, where teams alternate in throwing a total of eight stones each. When each end finishes, the team with the stone closest to the button scores points equal to the number of their stones in the house closer than any opponent stone. If the state in Figure 1 is the end of an end, yellow would score one point. If the yellow rock closest to the button were missing, then red would instead score two points. After an end, all stones are removed, and the next end begins. The winner is the team with the highest score after all ends are played. A key aspect of curling, is that rocks do not follow a straight trajectory. Instead, due to the friction with the ice stones curl left or right, depending on the stone's rotation as demonstrated by the dashed line in Figure 1. This fact allows teams to place stones behind other stones and make them harder to remove from the house.

An important strategic consideration in the game is execution uncertainty. A team may intend to deliver a stone at particular angle and velocity, but inevitably human precision has its limitations. In addition, the ice and stones create additional unpredictability as different parts of the ice or even different stones might see different amounts of friction and lateral forces. Debris on the ice can also cause drastic trajectory changes. When deciding what shot to attempt, the players have to carefully consider possible negative consequences of execution errors.

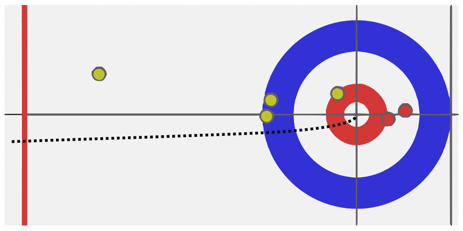
Figure 1: Sample curling game state. The yellow player scores 1 point if this is the end of an end. The dashed line indicates a typical rock trajectory.

4.2 Curling Simulator

The physics of a curling stone is not fully understood and is a surprisingly active area of research [Denny, 2002; Jensen and Shegelski, 2004; Nyberg *et al.*, 2012; 2013]; so, a simulation based on first principles is not possible. The curling simulator used in this paper is implemented using the Chipmunk 2D rigid body physics library with a heuristic lateral force that recreates empirically observed stone trajectories and modified collision resolution to match empirically observed elasticity and energy conservation when rocks collide. A rock's trajectory is simulated from its initial linear velocity and angular velocity (which due to the trajectory's insensitivity to the speed of rotation is determined only by its sign).

The curling simulation also needs a model of the execution uncertainty. We model it based on Olympic-level players. Their executed shot is usually quite close to the intended shot, with decreasing probability of being increasingly different. However, the stone can move over debris drastically altering the shot's trajectory from that which was intended. Therefore, we use the heavy-tailed Student-t distribution as the basis of our execution model. The amount of execution noise added to a shot depends on the *weight* (speed) of the shot. The faster shots tend to be more precise, since the ice conditions have smaller effect on their trajectory. The noise added to a specific shot is defined by three parameters: the variance of the Student-t distribution that is added to the intended weight, the variance of the Student-t distribution that is added to the aim, and the number of degrees of freedom of the distributions. We have fit these parameters to match the *curling percentage* statistics of men from the 2010 and 2014 Olympic games. The number of degrees of freedom for the Student-t distribution was fit to 5, the weight variance is approximately 9.5 mm/s, and the aim variance varies with the weight and is between 1.16×10^{-3} and 3.65×10^{-3} radians.

The simulator does not explicitly model *sweeping*, where the non-throwing members of a team can vigorously brush the ice in front of a travelling rock to reduce friction and keep a rock's trajectory straighter. While sweeping is critical to the game of curling, we model its effect as a reduction in the execution uncertainty, i.e., an increase in the likelihood the actual rock's trajectory is close to the planned trajectory.[1]

[1]Sweeping does have other effects beyond reducing execution

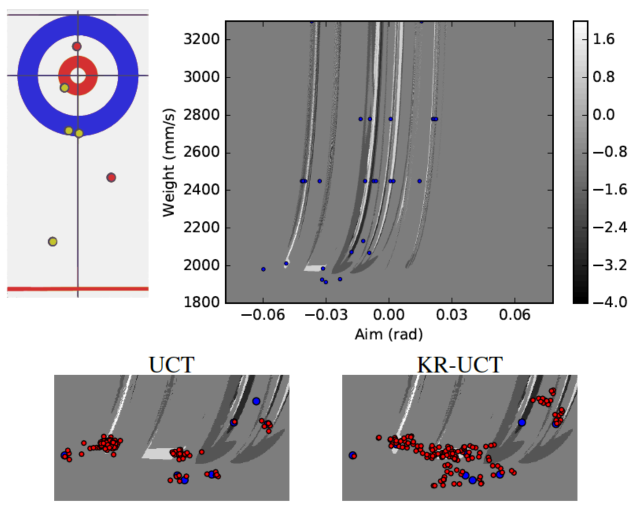

Figure 2: The game state of the final shot of the fifth end in the 2014 Olympics men's gold medal match. The scores of different shot parameters with an out-turn (counter-clockwise spin), the shots generated by the domain knowledge and the shots evaluated by the analyzed algorithms after 300 samples.

4.3 Domain Knowledge

All evaluated algorithms use the same domain knowledge: a curling-specific shot generator that creates a list of potential shots to take from a particular state. The generator works using a list of hand-defined types of shots, such as drawing to the button, removing an opponent's rock, or guarding its own rock, and generates the needed weight, aim, and rotation to make the shot, if it is possible. It also stores the purpose of this shot such as ("takeout yellow 3"), which can be used with RAVE to learn quickly about shots in a particular state. The generator typically generates between 7 and 30 different candidate shots. Figure 2 shows a game state in which red throws the last shot of the end. The grayscale heatmap shows the resulting score of executing a shot as the aim (x-axis) and weight vary (y-axis) when using a counter-clockwise rotation. Note that execution uncertainty makes achieving the exact values shown in the heatmap impossible in expectation. Overlaid on this heatmap are blue points showing the candidates shots from the generator. The shot generator does not always produce the best shot in expectation, as seen by the fact that there is no generated shot in the center of large light area with weight 2000 mm/sec and aim -0.035 radians.

The other domain knowledge in use is a hand-coded rollout policy. The policy involves a hand-coded set of rules to select an action from any state. Even though the rules are deterministic, the shots are executed with execution uncertainty and therefore the rollout policy outcomes vary. Because the physical simulator is computationally expensive, we did not rollout full ends. Instead, rollouts would be restricted to at most five simulated shots, after which the end was scored as if it were complete. Many shots in the middle of an end can result in a back-and-forth of placing and removing one's own/opponent stones, which this approach dispenses with.

error. Sweeping focused on the end of a rock's trajectory can allow it to reach a location on the ice not possible without sweeping. Furthermore, a shot far from an intended shot can be swept to achieve an entirely different purpose, such as rolling under a different guard if the executed angle is off. The primary effect of sweeping, though, is to compensate for execution error.

	KR-UCT	PW	RAVE+PW	RAVE	UCT
KR-UCT		**0.086**	**0.084**	**0.105**	**0.110**
PW	**-0.086**		0.010	0.014	**0.037**
RAVE + PW	**-0.084**	-0.010		0.015	**0.033**
RAVE	**-0.105**	-0.014	-0.015		0.006
UCT	**-0.110**	**-0.037**	**-0.033**	-0.006	

Table 1: The average point differences by the algorithms in the rows against the one in the column over one-end games. Bold values denote statistical significance (p=0.95; upper-tailed t-test).

4.4 Algorithms

We compare our proposed KR-UCT algorithm with four other variants of MCTS. (1) UCT: available actions are limited to generated shots, but the number of random outcomes sampled for each action is kept below the square root of the number of its visits. (2) RAVE: like UCT but with RAVE using the generator's action descriptions to identify actions. (3) PW: like UCT but with progressive widening on the generated actions; adding random actions when the number of actions is below the square root of the number of visits. (4) RAVE+PW: combination of RAVE and PW; added action has the highest value (UCT + RAVE).

All algorithms used 1600 samples and evaluated the final shot selection with a lower confidence bound estimate ($C_{LCB} = 0.001$). For each algorithm, we ran a round robin tournament to identify a good UCB constant from the set $\{0.01, 0.1, 1.0, 10, 100\}$. For all algorithms, $C_{UCB} = 1.0$ was the best constant. For the weighting in RAVE, we did a similar round robin tournament to select the β parameter from the set $\{0.01, 0.1, 1.0, 10.0, 100.0\}$, and found $\beta = 1.0$ to be the best for RAVE and RAVE+PW. For KR-UCT, we defined $\tau = 0.02$ and $k = 10$. We found that these values were a good trade off of runtime and exploration.

4.5 One-End Games

We first present results of head-to-head performance in a single end of curling. Each algorithm played against each other in 16000 one-end games. Because having the last shot (the *hammer*) is advantageous, algorithms played an equal number of games with and without the hammer against each opponent. The algorithms aimed to optimize the score differential in the game (as opposed to the probability of being ahead) and so our results are presented as average (differences in) points.

Table 1 shows the results of every pair of algorithms. KR-UCT had a positive expected point differential against every other algorithm by statisticially significant margins. Not surprisingly, UCT appears to be its weakest opponent. Additionally, we can see that while RAVE does not significantly outperform UCT, which can be attributed to the limited number of samples. With 1600 samples, the algorithm rarely searches past a depth of 3, so the RAVE statistics are underutilized.

	KR-UCT	PW	RAVE+PW	RAVE	UCT
KR-UCT	**1.424**	1.302	1.301	1.303	1.302
PW	**1.452**	1.297	1.297	1.297	1.297
RAVE+PW	**1.443**	1.284	1.284	1.284	1.284
RAVE	**1.446**	1.274	1.274	1.274	1.274
UCT	**1.431**	1.264	1.264	1.265	1.265

Table 2: Average number of points gained by algorithms in the columns in all hammer shot states reached by algorithms in the rows. The standard error on these values is 0.007. Bold values denote a statistically significant maximum within the row (p = 0.99; paired t-test).

	KR-UCT	RAVE+PW
KR-UCT	1.392	1.272
RAVE+PW	1.405	1.248

	KR-UCT	PW
KR-UCT	1.414	1.281
PW	1.416	1.265

	KR-UCT	RAVE
KR-UCT	1.428	1.311
RAVE	1.400	1.242

	KR-UCT	UCT
KR-UCT	1.461	1.344
UCT	1.423	1.255

Table 3: Average number of points gained by algorithms in the columns in the hammer shot states reached by algorithms in the rows in mutual matches of the algorithms mentioned in each table. Standard error ≤ 0.011 for each entry.

However, the algorithms with Progressive Widening result in an improvement, although the gain achieved by KR-UCT is much higher with a higher statistical confidence. KR-UCT's gain of 0.110 points per end, in expectation, against UCT, is a considerable result. If we extended the result to a standard 10-end game, KR-UCT might expect to see a full point improvement per game. In the 2014 Olympics, 29% of curling games were decided by a single point.

4.6 Hammer Shot Analysis

The hammer shot is an interesting case to analyze [Ahmad *et al.*, 2016]. Being the last shot of the end, it is by far the most important. Furthermore, the search does not have to recursively evaluate long term consequences of actions, so algorithms can evaluate substantially more actions in the same computational budget.

For each game in the round robin tournament, we took the state right before the hammer shot and let other algorithms choose a shot from that state. We then found the expected value of that shot by simulating it under our execution model 100,000 times. Table 2 shows the expected points earned by averaging over the reached hammer states. Each row represents the set of hammer states reached by a particular algorithm, and the columns represent the algorithms evaluated from these states. The subtables in Table 3 show the same analysis, except the set of states considered is limited to states from the matches between the two algorithms in the subtable. These tables allow us to analyze two effects: an algorithm's ability to choose a good hammer shot and an algorithm's ability to reach a good state for the hammer shot.

A clear picture emerges when comparing the columns of Table 2. KR-UCT outperforms all other algorithms by over 0.12 points in its shot selection, regardless of the set of hammer states under consideration. Also, notice that preexisting variants of UCT have nearly identical performance. RAVE learns from actions in a subtree, which are not present in this case; PW will always explore all options removing the need to consider a smaller subset.

This performance difference highlights the main disadvantage of other algorithms: they can choose only from the generated shots, which is a severe limitation in the hammer shot, when available samples are sufficient to explore far more options. It can be seen quite dramatically in the bottom part of Figure 2. The two heatmaps are overlaid with the shots evaluated by vanilla UCT and KR-UCT in the first 300 samples from this hammer state. Notice the generated candidate shot at weight 2000 mm/s and aim -0.03 radians, which is on the edge of a high-value region. In UCT, approximately half of the noisy samples of this shot receive 1 point, but the other half receives -1 point and the shot is not selected. The algorithm uses more samples in the smaller region left of this shot. However, KR-UCT is not restricted to choose only from the generated shots. As soon as it generates a noisy shot in the good region, the next sample will be near this shot and the sampling will gradually move all over the high-value region. The final shot is selected close to the center of this region.

Additionally, we can compare a table's rows to evaluate an algorithm's ability to reach a good state through lookahead and planning, even when a different algorithm is choosing the final hammer shot. Looking at the rows in the subtables of Table 3, we see a general trend that KR-UCT generally reaches states that result in equal or greater value regardless of which algorithm is selecting the final shot. On the whole this suggests KR-UCT is an improvement over UCT when the search space is shallow as well as when it is deep. This story is only contradicted in the case where KR-UCT is the algorithm selecting the hammer shot. This discrepancy is most clear in the full results of Table 2 where the preexisting enhancements to UCT seem to result in more valuable states for KR-UCT than KR-UCT does for itself (viz., compare the different rows in the KR-UCT column). We would like to explore this curious effect further in future work.

5 Conclusion

While a number of planning approaches have been proposed for sequential decision-making in continuous action spaces, none fully exploit the situation which arises from execution uncertainty. We present a new approach, KR-UCT, based on the insight that samples of execution uncertainty from a particular action provide information about any other action that could have generated that execution outcome. KR-UCT uses kernel regression to both share information between actions with similar distributions of outcomes and, more importantly, guide the exploration of actions outside of the initially provided candidate set. We show that the algorithm makes a significant improvement over both vanilla UCT and some of its standard enhancements.

Acknowledgements

This research was funded by NSERC and Alberta Innovates Technology Futures (AITF) through Amii, the Alberta Machine Intelligence Institute. The computational resources were made possible by Compute Canada and Calcul Québec. Additionally, we thank all others who have contributed in the development of the Computer Curling Research Group.

References

[Ahmad *et al.*, 2016] Zaheen Farraz Ahmad, Robert C. Holte, and Michael Bowling. Action selection for hammer shots in curling. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)*, 2016.

[Archibald *et al.*, 2009] Christopher Archibald, Alon Altman, and Yoav Shoham. Analysis of a winning computational billiards player. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 1377–1382. Morgan Kaufmann Publishers Inc., 2009.

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

[Bubeck *et al.*, 2011] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. χ-armed bandits. *Journal of Machine Learning Research*, 12:1655–1695, 2011.

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

[Couëtoux *et al.*, 2011] Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *Learning and Intelligent Optimization*, pages 433–445. Springer, 2011.

[Coulom, 2007] Rémi Coulom. Computing "ELO ratings" of move patterns in the game of go. *ICGA Journal*, 30(4):198–208, 2007.

[Denny, 2002] Mark Denny. Curling rock dynamics: Towards a realistic model. *Canadian journal of physics*, 80(9):1005–1014, 2002.

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

[Gelly *et al.*, 2006] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo go. Research Report RR-6062, INRIA, 2006.

[Guttman, 1984] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[Jensen and Shegelski, 2004] E.T. Jensen and Mark R.A. Shegelski. The motion of curling rocks: Experimental investigation and semi-phenomenological description. *Canadian journal of physics*, 82(10):791–809, 2004.

[Kearns *et al.*, 2002] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.

[Kocsis and Szepesvari, 2006] Levente Kocsis and Csaba Szepesvari. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer Berlin / Heidelberg, 2006.

[Mansley *et al.*, 2011] Christopher R. Mansley, Ari Weinstein, and Michael L. Littman. Sample-based planning for continuous action Markov decision processes. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*, 2011.

[Michels, 1992] Paul Michels. Asymmetric kernel functions in non-parametric regression analysis and prediction. *The Statistician*, pages 439–454, 1992.

[Nadaraya, 1964] Elizbar A Nadaraya. On estimating regression. *Theory of Probability & Its Applications*, 9(1):141–142, 1964.

[Nyberg *et al.*, 2012] Harald Nyberg, Sture Hogmark, and Staffan Jacobson. Calculated trajectories of curling stones sliding under asymmetrical friction. In *Nordtrib 2012, 15th Nordic Symposium on Tribology, 12-15 June 2012, Trondheim, Norway*, 2012.

[Nyberg *et al.*, 2013] Harald Nyberg, Sara Alfredson, Sture Hogmark, and Staffan Jacobson. The asymmetrical friction mechanism that puts the curl in the curling stone. *Wear*, 301(1):583–589, 2013.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 01 2016.

[Smith, 2007] Michael Smith. Pickpocket: A computer billiards shark. *Artificial Intelligence*, 171(1617):1069 – 1091, 2007.

[Watson, 1964] Geoffrey S. Watson. Smooth regression analysis. *Sankhya: The Indian Journal of Statistics, Series A*, pages 359–372, 1964.

[Yamamoto *et al.*, 2015] Masahito Yamamoto, Shu Kato, and Hiroyuki Iizuka. Digital curling strategy based on game tree search. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 474–480. IEEE, 2015.