ECAI 2012
Luc De Raedt et al. (Eds.)

doi:10.3233/978-1-61499-098-7-384

Towards Generalizing the Success of Monte-Carlo Tree Search beyond the Game of Go

António Gusmão and **Tapani Raiko** [1]

Abstract. Monte-Carlo Tree Search and specifically the variants of the UCT algorithm have been a break-through in AI of the board game Go. However, UCT has had limited applicability to other domains. We study the limitations of some of the existing variants of UCT in a small-scale Markov decision process (MDP), and propose new variants that can reduce those limitations. Our experiments show great improvements in performance against traditional UCT and comparable performance to estabilished reinforcement learning algorithm, thus opening possibilities for applying UCT in other problem domains.

1 Introduction

Abramson [1] proposed the expected-outcome heuristic for game state evaluation. The value of a game state is the expected outcome of the game given random play from that moment on. The value can be used to evaluate leaf nodes in a game-tree that has the current state as the root node and possible moves as children of the node. The expected-outcome heuristic applies to a large number of games with or without randomness or hidden information and with one or more players. Expected outcome heuristic is also known as roll-out analysis or play-out analysis.

The use of the stochastic play-out analysis in tree search became really useful when the stochasticity of the evaluation function was taken into account in a bandit based tree search known as the UCT (Upper Confidence bound in Trees) algorithm [8]. Its application in computer Go [7] revolutionalized the whole field. Monte-Carlo tree search has seen extensive use in games, but its application to other problem domains has been limited.

The game of Go has some properties that fit the UCT algorithm well. First, the number of available moves at a time is large, which prevents the use of brute-force search to explore all the possibilities as the search tree grows exponentially with respect to depth. In UCT, the number of visits is constant with respect to search depth, so the searches become deeper. Second, evaluation of a non-terminal state in Go is notoriously difficult [14] but UCT only needs to evaluate terminal states. In Go, the game tends to lead into a terminal state regardless of the used policy. Third, loops in the game are forbidden by rules and actions are determenistic, so the tree structure representation of the state space used in UCT fits it well. This paper will consider problems that arise when loops are common.

In the next section we will review existing work on the topic and discuss their limitations. In Section 3, we will propose new variants of the UCT algorithm, and in Section 4 we will make experimental comparisons.

[1] Aalto University School of Science, Finland, email: antoniogusmao@gmail.com, tapani.raiko@tkk.fi

2 Monte Carlo Tree Search and the UCT Algorithm

Monte Carlo Tree Search (MCTS) is a family of experience-based algorithms that learn by sampling event sequences and store the statistics in tree structures. These statistics are used to repeatedly solve a one-state Markov decision process (MDP) problem called *stochastic multi-armed bandit*. First we discuss the multi-armed bandit problem and its solution by using the asymptotically optimal algorithms UCB and UCB-V. Then we review UCT, a tree algorithm based on repeated application of UCB that is able to solve full MDPs and describes how these ideas have been applied to game playing, particularly in the board game Go.

2.1 Stochastic Multi-Armed Bandit

The stochastic multi-armed bandit problem has been extensively studied in statistics and is one of the simplest problem instances involving the exploration versus exploitation dilemma. Consider an agent that, at each time step, is repeatedly given the same K action possibilities. Each action incurs an immediate numerical reward taken from an unknown stationary probability distribution that depends on the action selected, i.e. rewards are independent and identically distributed [4]. The agent's goal is to select actions as to maximize the expected sum of rewards over a possibly infinite number of time steps. This is equivalent to a one-state Markov decision process with undiscounted rewards.

Denote the expected reward obtained from taking an action as the value of that action. Clearly, solving a multi-armed bandit problem is trivial if one knows the value of each of the actions - simply keep taking the action with best value. Because the values are unknown, the agent needs to estimate them by repeatedly trying each of the actions - the agent needs to *explore*. On the other hand, to achieve its main goal, the agent needs to *exploit* its current knowledge by taking the actions with best values. Exploration and exploitation are balanced by an allocation policy that defines the next action to take given a sequence of past actions and rewards obtained. The expected regret of an allocation policy allows one to compare different policies and, after T action selections, is given by

$$L_T = \max_a E\left\{\sum_{t=1}^{T} R(a,t)\right\} - E\left\{\sum_{t=1}^{T} R(a(t),t)\right\}, \quad (1)$$

where $R(a,t)$ is the reward obtained from taking action a at time t. Action $a(t)$ indicates which action was actually taken at time t. The first term in (1) is the reward the agent would get if he always picked the optimal action, whereas the second term is the reward from the actions $a(t)$ he actually took. Thus, the regret is the loss caused by

the policy not always picking the best action. It has been proven that, for a large class of reward distributions, there is no policy whose regret grows slower than $O(\log(T))$ [10].

Many algorithms have been proposed to solve the multi-armed bandit problem, such as ϵ-greedy, softmax strategy, ϵ_n-greedy ($\log(T)$ regret bound [4]) and UCB. Next, the UCB algorithm is described, which asymptotically achieves the optimal $O(\log(T))$ regret bound.

2.2 UCB Algorithms

UCB [4] stands for Upper Confidence Bounds and is a class of bandit algorithms that has been gaining popularity due to its theoretical guarantees, together with practical efficiency and ease of implementation. One of the simplest UCB algorithms is UCB1, a deterministic policy which, at each decision step, picks the action a with the largest $\rho(a)$

$$\rho(a) = \underbrace{\frac{1}{T}\sum_{t=1}^{T} R(a,t)}_{\text{average reward}} + \underbrace{\sqrt{\frac{2b^2\ln(T)}{N(a)}}}_{\text{bias factor}}, \tag{2}$$

where T is the total number of actions taken so far and $N(a)$ is the number of times action a was taken ($\sum_a N(a) = T$). Rewards are assumed to lie almost surely on the $[0, b]$ range, with $b > 0$.

The regret of UCB1 was proven to grow asymptotically at $O(\ln(T))$ rate. Additionally, Auer *et al.* [4] presented UCB2 and UCB-Tuned. UCB2 achieves smaller constants in the asymptotically regret bound but obtained worse empirical results. UCB-Tuned includes in its bias factor a measure of the empirical variance of rewards and outperformed both UCB1 and UCB2. This is intuitive since actions with low value variance do not require so much exploration as few samples are needed to get an accurate measure of their expected value. Although regret bounds for UCB-Tuned were not shown by Auer *et al.*, later Audibert *et al.* [3] laid theoretical foundations for a more general form of UCB using variance estimates in the bias factor, UCB-V.

Let $c \geq 0$ be an arbitrary constant. Denote $\overline{R}(a,T)$ as the empirical expected reward of action a after T episodes and $V(a,T)$ as the empirical variance, defined respectively by:

$$\overline{R}(a,T) = \frac{1}{N(a)} \sum_{\{t|a(t)=a\}} R(a,t) \tag{3}$$

$$V(a,T) = \frac{1}{N(a)} \sum_{\{t|a(t)=a\}} \left[R(a,t) - \overline{R}(a,T)\right]^2, \tag{4}$$

where $\{t|a(t) = a\}$ is the set of times t at which the action a was chosen. UCB-V plays the arm with maximal $B(a,T)$:

$$B(a,T) = \overline{X}(a,T) + \sqrt{\frac{2V(a,T)\xi(T)}{N(a)}} + c\frac{3b\xi(T)}{N(a)}, \tag{5}$$

where $\xi(T)$ are non-negative and non-decreasing real numbers for all $T \geq 0$. The function $\xi(T)$ is called *exploration function* and it determines how the bias evolves with subsequent plays. A common choice is $\xi(T) = q\log(T)$, with an arbitrary constant q.

2.3 UCT Algorithm

UCT [8] stands for upper confidence trees and is a Monte-Carlo tree search (MCTS) method that sets up a multi-armed bandit problem for each state and picks actions using UCB algorithms. Its applications include learning in episodic MDPs and minimax game tree search.

Consider a reinforcement learning agent. Denote $Q(s,a)$ as the value of action a in state s and $\mathcal{A}_s$ as the set of available action at state s. Let $\beta(s,a)$ be a real-valued bias factor. The value under UCT is given by:

$$Q_{UCT}(s,a) = Q(s,a) + \beta(s,a). \tag{6}$$

If actions are selected based on the UCB1 bandit algorithm, according to Equation (2), the bias factor is

$$\beta(s,a) = \sqrt{\frac{2b^2\ln(N(s))}{N(s,a)}},$$

where $N(s)$ it the number of times state s was visited and $N(s,a)$ is the number of times action a was taken in state s.

Essentially, UCT adds a bonus to each action value, the bonus being determined by the bias factor $\beta(s,a)$. When the bias factor is given by UCB1, taking action a at state s results in a decrease of the bonus given to a and an increase to the one given to all other actions that could have been taken, i.e. $\beta(s,a)$ is decreased and $\beta(s,a')$ is increased for all $a' \neq a$, $a' \in \mathcal{A}_s$. This helps understand the reason why UCB1 never stops exploring. It follows directly from Equation (2) that, in an infinite number of action selections at state s, any action will have its value increased to infinity if it is not picked infinitely often.

UCT was described by Kocsis and Szepesvári [8] using UCB1. The authors proved that action values $Q(s,a)$ converge to their expected value at a rate of $O(\log(N(s))/N(s))$. It has been proven that UCB1 picks the best action exponentially more times than all other actions [4].

2.4 UCT and Game AI

The UCT algorithm is a best-first search method that is able to learn an evaluation function by repeatedly playing or simulating game episodes. Most Monte-Carlo tree search algorithms, in particular UCT, can be applied to game AI under a simple common structure. A game played from start to finish constitutes one episode. A tree is built incrementally from each experienced or simulated episode, and the whole procedure consists in repeating the following four steps:

1. **Selection**: starting at the root (either the starting state or the current game state), recursively pick actions until a previously unseen state is encountered.
2. **Simulation**: from the new state onward, take actions according to a default policy (e.g. random policy) until a terminal state is reached (end of the game).
3. **Expansion**: add one or more newly found states to the tree.
4. **Backpropagation**: propagate the result back to all visited states during the episode.

Each game episode, e, consists of a sequence of state-action pairs:

$$e = \{(s_1,a_1),(s_2,a_2),...,(s_m,a_m)\}$$

The final pair (s_m, a_m) must result in either a win or a loss. In UCT using UCB1, each tree node z stores:

- s: the state corresponding to the tree node.
- $Q(z,a)$: the action values for all actions $a \in \mathcal{A}_s$.
- $N(z,a)$: the number of times action a was taken in node z

Actions values, $Q(z, a)$, are an empirical estimate of the probability of winning from node z (and state s) when action a is taken.

The selection process follows the UCB1 policy. At each node z (and corresponding state s), the agent picks action a with maximal:

$$Q_{UCT}(z, a) = Q(z, a) + c\sqrt{\frac{\ln(\sum_{a'} N(z, a'))}{N(z, a)}}. \quad (7)$$

Note that compared to Equation (2), a real-valued constant c was introduced to balance the importance of exploration and exploitation. This constant should reflect the agent's prior knowledge regarding the amount of exploration required.

The selection process progresses down the tree until a new node is found. At this point, a default policy commences and is used to take the game into a terminal state, thus completing the episode.

From this episode, one or more new nodes can be added to the tree. In Go programs it is usual to only add the first non-visited state (e.g. [6]), whereas some authors add states from the full episode [8, 5].

The final backpropagation step is straightforward. Denote the set of all nodes in the current tree as S. When an episode e terminates, all state-action pairs in the episode that are also in the tree, all $(s, a) \in e \cap S$ are updated. Let $R(e)$ be the final result of the game episode, with $R(e) = 1$ if the episode ended in a win, and $R(e) = 0$ otherwise. Then, the updates are:

$$N(z, a) \leftarrow N(z, a) + 1 \quad (8)$$

$$Q(z, a) \leftarrow Q(z, a) + \frac{1}{N(z, a)}[R(e) - Q(z, a)]. \quad (9)$$

The update for $Q(z, a)$ is an incremental implementation of the average operator.

Considering reinforcement learning formalism, this is equivalent to an undiscounted MDP where all states have zero rewards except terminal states that result in a win, which are assigned value 1. UCT qualifies as a on-policy Monte-Carlo algorithm since $Q(z, a)$ is the expected reward of taking action a in node z directly obtained from following the UCT policy. The UCT policy is, in essence, a union of two policies: a policy based on UCB, valid for states in the search tree; and a default policy for states outside the tree. As episodes are fed to UCT, the search tree grows and the default policy becomes less important.

Many of the applications of UCT deal with minimax trees and deterministic games. For minimax game trees, UCT can be applied directly using the negamax convention for MIN decision nodes. An example of this is UCT in computer Go (see e.g. [7, 6, 11]), where professional level of play was achieved for 9x9 boards. Other game AI implementations include computer Hex [2], Settlers of Catan[2] [13] and Poker [12], the last involving the earlier discussed UCB-Tuned algorithm. Additionally, UCT was applied in RTS games with moderate success [9].

2.5 Trans-UCT

For connected graphs, UCT may store multiple nodes for the same state, resulting in the exploration of a considerably larger search space, possibly exponential in regards to average branching factor. Childs *et al.* [5] detect state transpositions in computer-Go and store the statistics $Q(s, a)$ and $N(s, a)$ for each $\{s, a\}$, as opposed to storing them for each tree node. This helps UCT obtain reliable estimates quicker. However, a tree structure is still mantained whose size keeps increasing even if all $\{s, a\}$ pairs are already represented in the tree. The authors also assume that the game is a directed acyclic graph (DAG). To help differentiate the several UCT variants we denote this approach as Trans-UCT.

2.6 Limitations of UCT and Trans-UCT

An important issue that has not seen enough attention in the UCT literature is the fact that, in most games, states can be revisited. Thus, a game should be represented as a connected graph (a MDP), not as a tree nor as a DAG.

In MDPs the value of a state-action pair independent of the path that originated it, due to the Markov property. Optimal behavior can be achieved by determining the optimal action values $Q(s, a)$ for all states and actions. Therefore statistics only need to be stored for each state-action pair, and need not consider the episode history. We can conclude that for MDPs the tree-based formalism of UCT has an immense degree of redundancy, storing values for each node of the tree, nodes that can represent the same states. Consequently, UCT requires more episodes to get accurate estimates of action values.

An additional drawback of the tree formalism is that it heightens the reliance of UCT on the random policy. UCT takes the random policy every time a new node is discovered and this happens very often since every possible path taken results in additional nodes of the tree. To alleviate this issue, in the realm of computer Go, Gelly *et al.* [6] combine online learning via UCT with offline learning of a default policy using TD(0) and linear function approximation. Whenever UCT visits a new state, both $Q(z, a)$ and $N(z, a)$ are initialized with prior values $Q_{prior}(z, a)$ and $N_{prior}(z, a)$, where the former is the offline-learned value function and the latter is a designer-selected integer value which corresponds to the equivalent experience contained in $Q_{prior}(z, a)$.

The use of tree data structures is not devoid of reason. Recall that UCT is composed of two policies, the UCB policy and a random policy. The UCB policy enforces exploration but only across different episodes, as it is a deterministic policy which will repeatedly pick the same action given the same action value and visit counts. Hence, an agent that constantly chooses actions according to UCB may get stuck in loops if the value function has (local) maxima at non-terminal states. UCT avoids such loops by switching from UCB to a random policy based on the history of the executing episode. The following section introduces alternative approaches to avoid loops that do not require tree data structures.

3 Proposed Improvements for UCT in MDPs

To understand how to make better use of the statistics collected in each episode, first we examine the approach taken by Childs *et al.* [5], Trans-UCT, which stores values $Q(s, a)$ and $N(s, a)$ but maintains an explicit tree representation. Although the values $Q(s, a)$ and $N(s, a)$ are not duplicated in different nodes of the tree, the tree structure is not purposeless as it determines whether the policy taken is the UCB policy or the random policy. However, storing a full tree is infeasible in all but the most simple environments[3]. Similarly to

[2] Settlers of Catan is a modern multi-player board game. An interesting difference compared to classical games (such as Chess) is an initial randomized board state.

[3] The reader may be reminded that UCT had success in the realm of computer-Go, which is certainly not a small-scale environment. However,

Childs *et al.*, we ideally wish to store unique $Q(s,a)$ and $N(s,a)$ values for each state-action pair. Unfortunately, as discussed previously, UCT must not rely on UCB alone as its deterministic nature results in infinite loops if the algorithm does not switch to a policy which enforces exploration within an episode. In other words, any proposed policy must be soft, i.e. it must guarantee that any action a at any state s has a non-zero probability of being taken, at any episode. Note that if we exclude the random policy and consider only the UCB part of UCT, then UCT is not a soft policy. It always picks the action with highest value and it only updates the exploration biases at the end of each episode. We propose several alternative improvements to the UCT algorithm.

3.1 Revisit-UCT

Revisit-UCT stores $N(s,a,u)$ and $Q(s,a,u)$, where u is the number of times action a was taken in state s in the current episode.

Similar to traditional tree-based UCT, separate values for the same state-action pair are stored. However, Revisit-UCT makes better use of collected experience and reduces the amount of redundancy in the information stored. First, whereas the trees built by UCT are only meaningful for a fixed starting state, Revisit-UCT is valid whichever state the environment starts in. Second, the values stored are considerably less dependent on the paths taken, making better use of collected experience and resulting in faster learning. Finally, Revisit-UCT is able to execute the UCB policy in any state where the current number of visits to any available action is smaller than u. In contrast, as soon as UCT reaches a leaf state, the random policy is executed until the episode terminates, independently of what states happen to be visited. Thus, one should expect Revisit-UCT to clearly outperform classic UCT, particularly in highly connected environments where exploratory behavior leads to numerous visits to the same states, within an episode. To limit memory requirements, we bound the maximum depth to $u = 10$, i.e. Revisit-UCT considers $N(s,a,u) = 0$ for $u > 10$.

3.2 Revisit-UCT-V

Revisit-UCT-V is identical to Revisit-UCT except that actions are picked according to UCB-V instead of UCB. A variance estimate is required for each (s,a,u) triplet and is stored in variable $V(s,a,u)$. The action-value function becomes

$$Q_{UCT}(s,a,u) = Q(s,a,u) \tag{10}$$

$$+ c\left(\sqrt{2\frac{V(s,a,u)\xi(s,u)}{N(s,a,u)}} + 3\frac{\xi(s,u)}{N(s,a,u)}\right) \tag{11}$$

with

$$\xi(s,u) = \ln\left(\sum_a N(s,a,u)\right).$$

Variance estimates are computed using an incremental algorithm to avoid storing the whole history of value updates.

UCT when applied to computer-Go does not construct full search trees. At each decision step, it builds a limited-depth search tree relying on the random policy to estimate the values of leaf nodes. Thus, UCT in computer-Go does not learn a permanent value function, limiting itself to learning an ephemeral value function which is discarded after each move.

3.3 Stochastic UCT

Stochastic UCT (SUCT) imposes within-episode exploration through the use of a stochastic policy. In regular UCT, UCB always picks the action with the highest $Q_{UCB}(s,a)$ value. In contrast, SUCT relies on a stochastic policy to do that selection. Thus, the purely random policy is never required, resulting in an algorithm that explores considerably less when few episodes have been experienced. The disadvantage comes from having to select an appropriate stochastic policy. The within-episode exploration depends on the randomness of the stochastic policy, but we must require that the policy converges to the greedy policy to guarantee that SUCT converges to the greedy policy as well. The exploratory behavior across different episodes can be controlled independently from within-episode exploration by selecting the constant c in Equation (7).

For SUCT we chose a dynamic ϵ-greedy policy where the value of epsilon is:

$$\epsilon = \frac{w_n\epsilon_n + w_r\epsilon_r}{w_n + w_r} \tag{12}$$

with

$$w_n = \sum_{a'} T(s,a'), \qquad w_r = e^{\sum_{a'} M(s,a')} - 1$$

$$\epsilon_r = 1/|\mathcal{A}_s| \quad \text{and} \quad \epsilon_n = \frac{\epsilon_0}{\sqrt{\sum_{a'} T(s,a')}}.$$

Thus, the value of ϵ is a weighted average of ϵ_r and ϵ_n. The former value represents a uniform policy whereas the latter is an epsilon-greedy policy which converges to the greedy policy. The weight w_n increases with the experience collected in state s whereas w_r increases with the number of visits to a state within the same episode. Intuitively, the purpose of these weights is to quickly shift to a random policy when a state is visited several times in the same episode. Thus, SUCT assumes that an optimal policy should not visit the same state repeatedly. This is a valid assumption for many problem instances such as games and navigation tasks. Finally, $\epsilon_0 \in [0,1[$ is a free parameter that determines the amount of experience required to reduce the randomness of the policy.

3.4 Online UCT

In online UCT (OUCT), exploration within an episode is enforced directly by the UCB algorithm. This is achieved by considering an exploration bias which accounts for the visits to a state-action pair during execution of an episode, in an online manner. Let $M(s,a)$ represent the number of times action a was taken in state s during the current executing episode. Let $T(s,a)$ be the number of episodes where $\{s,a\}$ was visited. Then, the value for a state-action pair is

$$Q_{OUCT}(s,a) = Q(s,a) + c\sqrt{\frac{\ln(\sum_{a'} M(s,a') + T(s,a'))}{M(s,a) + T(s,a)}}. \tag{13}$$

With the exploration enforced by the modified value function, the necessity for a random policy is greatly reduced. In OUCT, the random policy is only triggered when no statistical information is available for a state-action pair. The policy becomes

$$\mathcal{I}_{OUCT}(s) = \begin{cases} \arg\max_{a'} Q_{OUCT}(s,a') & \text{if all } T(s,a') > 0 \\ \sim \text{Uniform}(unexplored(s)) & \text{otherwise} \end{cases} \tag{14}$$

where $unexplored(s)$ represents the set of all actions available at state s for which there is no statistical information, i.e. all $a' \in \mathcal{A}_s$ for which $T(s,a') = 0$.

4 Experiments

4.1 Testing Environment

The environment is a discounted-MDP that mimics some of the characteristics found in adversarial games. There is a unique starting state and two types of terminal states, winning states and losing states. The rewards given to the agent are zero at all states except winning states, for which the reward is 1. The rewards are discounted with discount factor $\gamma = 0.99$.

At each state there are at most four actions available, the moves to adjacent cells (North, South, East, West) that are not walls. Moves are not deterministic; they will not always move to their intended target states because terminal loss states of the environment act as attractors that, with some probability, force the agent to move in the direction of the closest losing state. Let s' be the state resulting from taking action a at state s. In addition, let s_c represent the losing state which is closest to state s, measured by euclidean distance. Then, the model of each action has the following form:

$$s' = \begin{cases} \text{target state} & \text{with probability } p(s) \\ \text{state closer to } s_c & \text{with probability } 1 - p(s) \end{cases} \quad (15)$$

Notice that both s_c and $p(s)$ are determined regarding the origin of the action, s, thus being independent of the action taken by the agent. The value of $p(s)$ is

$$p(s) = e^{-\|s - s_c\|^2/\sigma},$$

where σ is a map-dependent constant used to design and tune map environments. Thus, $p(s)$ is higher the closer s is to a losing state.

The environment possesses three fundamental differences when compared to Go: (1) the actions are stochastic; (2) the reward is discounted; and (3) there is a high probability of revisiting states when exploring the state space.

Three maps, shown in Figure 1, were used to test the algorithms in distinct situations. Each grid cell is a state of the environment and the color represents its optimal value computed by dynamic programming. Note that the *9x9* map is a deterministic environment since it has no losing states. The maps picked are particularly small-scale problems in order to illustrate the difficulties the algorithms show even when learning in such simple environments.

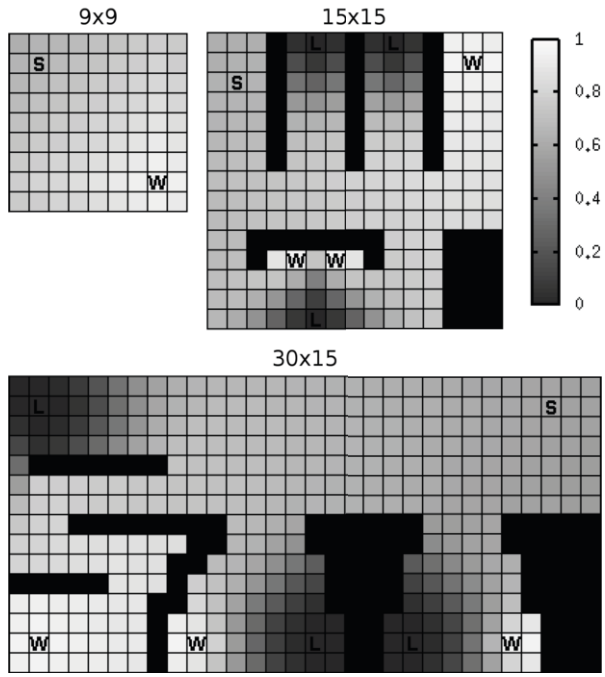

Figure 1. Legend: Black-colored cells are walls; S - start; L - losing state; W - winning state. The grey-scale values on the floor show the value of the state assuming optimal policy.

4.2 Results

Revisit-UCT is compared with UCT and Trans-UCT, both in which the random policy is executed after the first non-UCB action selection. The first (leftmost) column in Figure 2 presents results for three different maps, averaged over 10 independent runs. The value of c for all three algorithms was empirically hand-picked. Results show that Revisit-UCT achieves higher rewards than both UCT and Trans-UCT in all maps. We confirmed that our implementation of UCT and Trans-UCT were able to solve the small map by re-running the experiment with 100 000 episodes (not shown).

Not only was Revisit-UCT the only algorithm able to obtain a reasonably accurate estimate of the optimal value function, it did so without a significant impact on earned rewards. At first glance it might appear that UCT and Trans-UCT did not explore the full action space. However, this is not true. Both algorithms explored the action space (confirmed by looking at the visit counter), but their exploration was not effective as they switch to a uniform random policy which is completely unguided and is likely to result in small rewards due to discounting. Revisit-UCT makes better use of its existing knowledge which allows it to obtain higher rewards given the same experience.

The second column in Figure 2 compares the rewards for the three proposed methods, Revisit-UCT, Stochastic UCT (SUCT), and Online UCT (OUCT). SUCT lacks an effective mechanism to escape local maxima. OUCT does a better job at that since it is not random and its exploration bias is updated during the execution of an episode, forcing it to explore a larger region of state-action space. Revisit-UCT is clearly superior in all maps, fact which is not easily justified. We suspect that storing different action-values at each depth improves the capacity of the algorithm to deal with the non-stationary policy, since estimates at higher depths will be based on experience collected from a better performing policy. In addition, (local) maxima of the action-value function have to appear at several different depths to pose a significant problem to Revisit-UCT.

The third column in Figure 2 compares Revisit-UCT to established reinforcement learning algorithms, namely Q-learning and on-policy MC. While Q-learning works very well in the smallest problem, it seems that Revisit-UCT is competitive in the large ones.

Finally, the addition of a variance estimate to the bias factor of Revisit-UCT proved to have minimal impact in the algorithm as the last column in Figure 2 shows. It is unclear whether Revisit-UCT-V has any advantage when compared to Revisit-UCT. Revisit-UCT-V seems to have a negative effect of increasing the variance of rewards obtained, but the algorithm appears to be stable nonetheless.

Table 1 presents a summary of the main characteristics of the UCT algorithms analyzed. All the proposed variants of UCT surpass the original algorithm and Revisit-UCT achieves the highest rewards. OUCT and SUCT have trouble leaving (local) maxima of the action-value function which is the main reason for their low performance in the *15x15* and *30x15* maps.

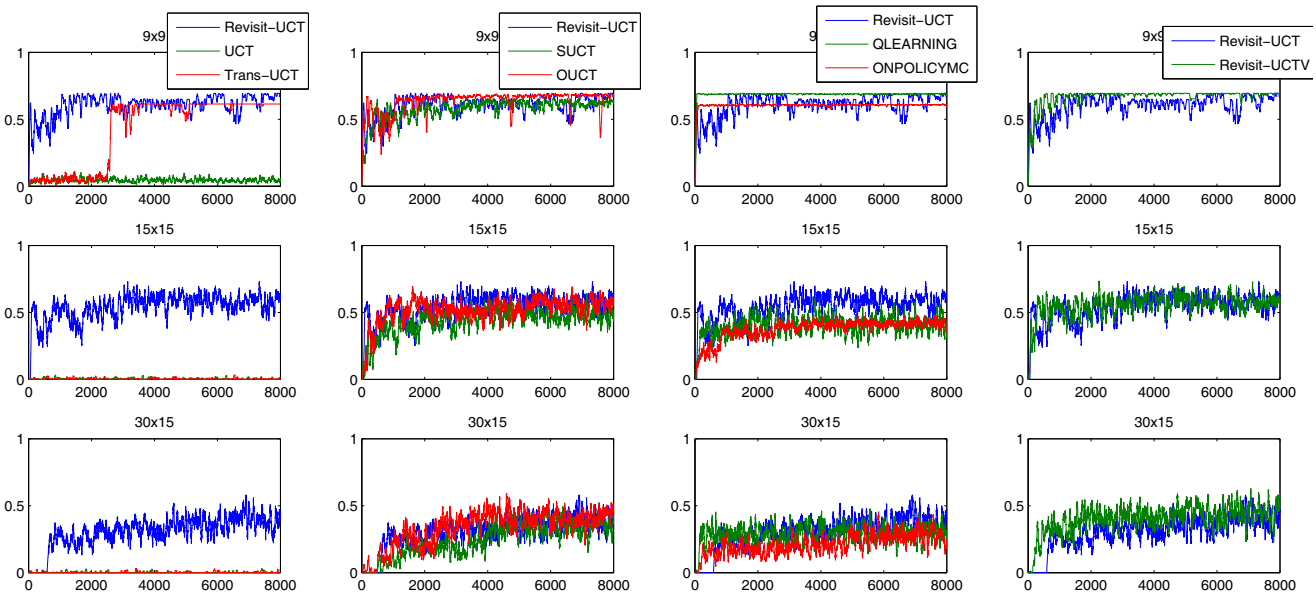

Figure 2. Obtained rewards as a function of the episode count for diffent problems and methods. See text for explanation.

Algorithm	Learned Function	Exploitation Stage(UCB)	Rank
UCT	$Q(s,a,\text{TreeNode})$	if TreeNode $\in$ tree	4
Trans-UCT	$Q(s,a)$	if TreeNode $\in$ tree	3
Revisit-UCT	$Q(s,a,u)$	if $N(s,\bar{a},M_{s\bar{a}})>0$	1
Revisit-UCT-V	$Q(s,a,u)$ $W(s,a,u)$	Revisit-UCT with UCB-V	1
SUCT	$Q(s,a)$	always (stochastic)	2
OUCT	$Q(s,a)$	if $N(s,\bar{a})>0$	2

Table 1. Summary of the main characteristics of UCT algorithms. Classical UCT has two stages: exploitation with UCB; and exploration with a uniform random policy. Rank is a subjective order (smaller is better) based on the experimental results presented. Legend: M_{sa}: number of visits to (s,a) in episode; d: depth of (s,a); Note: $N(s,\bar{a})>0 \Leftrightarrow N(s,a')>0, \forall a' \in \mathcal{A}_s$.

5 Discussion

Traditional variants of the UCT algorithm do not perform well in general Markov decision process problems despite their huge success in computer Go. We proposed three variants of UCT that are all much better in the studied problem. Revisit-UCT performed the best, but still, it has two drawbacks. First, it requires storage of the value function at different revisit numbers u, which might correspond to a prohibitive amount of memory. More importantly, representing $Q(s,a,u)$ through function approximation is a considerably harder problem than representing $Q(s,a)$. Also the other new variants, SUCT and OUCT, introduce interesting concepts which can be transferred to the continuous case with little additional effort. As a conclusion, we have provided tools that make UCT a strong alternative for solving MDP problems.

One of the main benefits of using UCB is its asymptotically optimal regret bound. In future, it would be important to show similar results for our variants. For instance, in SUCT, we assumed that an optimal policy for the map environment should not visit the same state repeatedly, which of course limits its optimality in the general case.

REFERENCES

[1] Bruce Abramson, 'Expected-outcome: A general model of static evaluation', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **12**(2), 182–193, (1990).

[2] Broderick Arneson, Ryan Hayward, and Philip Henderson. MoHex wins Hex tournament, 2009.

[3] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári, 'Tuning bandit algorithms in stochastic environments', in *Algorithmic Learning Theory*, eds., Marcus Hutter, Rocco Servedio, and Eiji Takimoto, volume 4754 of *Lecture Notes in Computer Science*, 150–165, Springer Berlin / Heidelberg, (2007). 10.1007/978-3-540-75225-7_15.

[4] Peter Auer and Jyrki Kivinen, 'Finite-time analysis of the multiarmed bandit problem', in *Machine Learning*, pp. 235–256, (2002).

[5] Benjamin E. Childs, James H. Brodeur, and Levente Kocsis, 'Transpositions and Move Groups in Monte Carlo Tree Search', in *IEEE Symposium on Computational Intelligence and Games*, eds., Philip Hingston and Luigi Barone, pp. 389–395. IEEE, (December 2008).

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

[7] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go, December 2006.

[8] Levente Kocsis and Csaba Szepesvári, 'Bandit based Monte-Carlo planning', in *In: ECML-06. Number 4212 in LNCS*, pp. 282–293. Springer, (2006).

[9] Radha krishna Balla and Alan Fern, 'UCT for tactical assault planning in real-time strategy games', (2009).

[10] T.L. Lai and H. Robbins, 'Asymptotically efficient adaptive allocation rules', *Advances in Applied Mathematics*, 4–22, (1985).

[11] Chang-Shing Lee, Mei-Hui Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong, 'The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments', *Computational Intelligence and AI in Games, IEEE Transactions on*, **1**(1), 73–89, (2009).

[12] Raphaël Maîtrepierre, Jérémie Mary, and Rémi Munos, 'Adaptive play in Texas Hold'em Poker', in *Proceeding of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, pp. 458–462, Amsterdam, The Netherlands, The Netherlands, (2008). IOS Press.

[13] István Szita, Guillaume Chaslot, and Pieter Spronck, 'Monte-Carlo tree search in Settlers of Catan', in *Proceedings of Advances in Computer Games*, (2009).

[14] Lin Wu and Pierre Baldi, 'A scalable machine learning approach to go', in *in Advances in Neural Information Processing Systems 19*, pp. 1521–1528. MIT Press, (2007).