Combining Online and Offline Knowledge in UCT

Sylvain Gelly, David Silver

▶ **To cite this version:**

Combining Online and Offline Knowledge in UCT

Sylvain Gelly   SYLVAIN.GELLY@LRI.FR
Univ. Paris Sud, LRI, CNRS, INRIA, France

David Silver   SILVER@CS.UALBERTA.CA
University of Alberta, Edmonton, Alberta

Abstract

The UCT algorithm learns a value function online using sample-based search. The $TD(\lambda)$ algorithm can learn a value function offline for the on-policy distribution. We consider three approaches for combining offline and online value functions in the UCT algorithm. First, the offline value function is used as a default policy during Monte-Carlo simulation. Second, the UCT value function is combined with a rapid online estimate of action values. Third, the offline value function is used as prior knowledge in the UCT search tree. We evaluate these algorithms in 9×9 Go against GnuGo 3.7.10. The first algorithm performs better than UCT with a random simulation policy, but surprisingly, worse than UCT with a weaker, handcrafted simulation policy. The second algorithm outperforms UCT altogether. The third algorithm outperforms UCT with handcrafted prior knowledge. We combine these algorithms in *MoGo*, the world's strongest 9×9 Go program. Each technique significantly improves *MoGo's* playing strength.

1. Introduction

Value-based reinforcement learning algorithms have achieved many notable successes. For example, variants of the $TD(\lambda)$ algorithm have learned to achieve a master level of play in the games of Chess (Baxter et al., 1998), Checkers (Schaeffer et al., 2001) and Othello (Buro, 1999). In each case, the value function is approximated by a linear combination of binary features. The weights are adapted to find the relative

contribution of each feature to the expected value, and the policy is improved with respect to the new value function. Using this approach, the agent learns knowledge that applies across the on-policy distribution of states.

UCT is a sample-based search algorithm (Kocsis & Szepesvari, 2006) that has achieved remarkable success in the more challenging game of Go (Gelly et al., 2006). At every time-step the UCT algorithm builds a search tree, estimating the value function at each state by Monte-Carlo simulation. After each simulated episode, the values in the tree are updated online and the simulation policy is improved with respect to the new values. These values represent local knowledge that is highly specialised to the current state.

In this paper, we seek to combine the advantages of both approaches. We introduce two new algorithms that combine the general knowledge accumulated by an offline reinforcement learning algorithm, with the local knowledge found online by sample-based search. We also introduce a third algorithm that combines two sources of knowledge found online by sample-based search: one of which is unbiased, and the other biased but fast to learn.

To test our algorithms, we use the domain of 9×9 Go (Figure 1). The program *MoGo*, based on the UCT algorithm, has won the KGS Computer Go tournament at 9×9, 13×13 and 19×19 Go, and is the highest rated program on the Computer Go Online Server (Gelly et al., 2006; Wang & Gelly, 2007). The program *RLGO*, based on a linear combination of binary features and using the $TD(0)$ algorithm, has learned the strongest known value function for 9×9 Go that doesn't incorporate significant prior domain knowledge (Silver et al., 2007). We investigate here whether combining the online knowledge of *MoGo* with the offline knowledge of *RLGO* can achieve better overall performance.

2. Value-Based Reinforcement Learning

Value-based reinforcement learning methods use a *value function* as an intermediate step for computing a policy. In episodic tasks the *return R_t* is the total reward accumulated in that episode from time t until it terminates at time T. The action value function $Q^\pi(s,a)$ is the expected return after action $a \in \mathcal{A}$ is taken in state $s \in \mathcal{S}$, using policy π to select all subsequent actions,

$$R_t = \sum_{k=t+1}^{T} r_k$$
$$Q^\pi(s,a) = E_\pi[R_t | s_t = s, a_t = a]$$

The value function can be estimated by a variety of different algorithms, for example using the $TD(\lambda)$ algorithm (Sutton, 1988). The policy can then be *improved* with respect to the new value function, for example using an ϵ-greedy policy to balance exploitation (selecting the maximum value action) with exploration (selecting a random action). A cyclic process of evaluation and policy improvement, known as *policy iteration*, forms the basis of value-based reinforcement learning algorithms (Sutton & Barto, 1998).

If a model of the environment is available or can be learned, then value-based reinforcement learning algorithms can be used to perform *sample-based search*. Rather than learning from real experience, simulated episodes can be sampled from the model. The value function is then updated using the simulated experience. The Dyna architecture provides one example of sample-based search (Sutton, 1990).

3. UCT

The UCT algorithm (Kocsis & Szepesvari, 2006) is a value-based reinforcement learning algorithm that focusses exclusively on the start state and the tree of subsequent states.

The action value function $Q_{UCT}(s,a)$ is approximated by a partial tabular representation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A}$, containing a subset of all (state, action) pairs. This can be thought of as a search tree of visited states, with the start state at the root. A distinct value is estimated for each state and action in the tree by Monte-Carlo simulation.

The policy used by UCT is designed to balance exploration with exploitation, based on the multi-armed bandit algorithm UCB (Auer et al., 2002).

If all actions from the current state s are represented in the tree, $\forall a \in \mathcal{A}(s), (s,a) \in \mathcal{T}$, then UCT selects the action that maximises an upper confidence bound on the action value,

$$Q_{UCT}^{\oplus}(s,a) = Q_{UCT}(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}}$$
$$\pi_{UCT}(s) = \underset{a}{\arg\max}\, Q_{UCT}^{\oplus}(s,a)$$

where $n(s,a)$ counts the number of times that action a has been selected from state s, and $n(s)$ counts the total number of visits to a state, $n(s) = \sum_a n(s,a)$.

If any action from the current state s is not represented in the tree, $\exists a \in \mathcal{A}(s), (s,a) \notin \mathcal{T}$, then the uniform random policy π_{random} is used to select an action from all unrepresented actions, $\tilde{\mathcal{A}}(s) = \{a | (s,a) \notin \mathcal{T}\}$.

At the end of an episode $s_1, a_1, s_2, a_2, ..., s_T$, each state action pair in the search tree, $(s_t, a_t) \in \mathcal{T}$, is updated using the return from that episode,

$$n(s_t, a_t) \leftarrow n(s_t, a_t) + 1 \tag{1}$$
$$Q_{UCT}(s_t, a_t) \leftarrow Q_{UCT}(s_t, a_t) \tag{2}$$
$$+ \frac{1}{n(s_t, a_t)}[R_t - Q_{UCT}(s_t, a_t)]$$

New states and actions from that episode, $(s_t, a_t) \notin \mathcal{T}$, are then added to the tree, with initial value $Q(s_t, a_t) = R_t$ and $n(s_t, a_t) = 1$. In some cases, it is more memory efficient to only add the first visited state and action such that $(s_t, a_t) \notin \mathcal{T}$ (Coulom, 2006; Gelly et al., 2006). This procedure builds up a search tree containing n nodes after n episodes of experience.

The UCT policy can be thought of in two stages. In the beginning of each episode it selects actions according to knowledge contained within the search tree. But once it leaves the scope of its search tree it has no knowledge and behaves randomly. Thus each state in the tree estimates its value by Monte-Carlo simulation. As more information propagates up the tree, the policy improves, and the Monte-Carlo estimates are based on more accurate returns.

If a model is available, then UCT can be used as a sample-based search algorithm. Episodes are sampled from the model, starting from the actual state $\hat{s}$. A new representation $\mathcal{T}(\hat{s})$ is constructed at every actual time-step, using simulated experience. Typically, thousands of episodes can be simulated at each step, so that the value function contains a detailed search tree for the current state $\hat{s}$.

In a two-player game, the opponent can be modelled using the agent's own policy, and episodes simulated by self-play. UCT is used to maximise the upper confidence bound on the agent's value and to minimise the lower confidence bound on the opponent's. Under certain assumptions about non-stationarity, UCT converges on the minimax value (Kocsis & Szepesvari, 2006). However, unlike other minimax search algorithms such as alpha-beta search, UCT requires no prior domain knowledge to evaluate states or order moves. Furthermore, the UCT search tree is non-uniform and favours the most promising lines. These properties make UCT ideally suited to the game of Go, which has a large state space and branching factor, and for which no strong evaluation functions are known.

4. Linear Value Function Approximation

UCT is a tabular method and does not generalise between states. Other value-based reinforcement learning methods offer a rich variety of function approximation methods for state abstraction in complex domains (Schraudolph et al., 1994; Enzenberger, 2003; Sutton, 1996). We consider here a simple approach to function approximation that requires minimal prior domain knowledge (Silver et al., 2007), and which has proven successful in many other domains (Baxter et al., 1998; Schaeffer et al., 2001; Buro, 1999).

We wish to estimate a simple reward function: $r = 1$ if the agent wins the game and $r = 0$ otherwise. The value function is approximated by a linear combination of binary features ϕ with weights θ,

$$Q_{RLGO}(s, a) = \sigma \left(\phi(s, a)^T \theta \right)$$

where the sigmoid squashing function σ maps the value function to an estimated probability of winning the game. After each time-step, weights are updated using the $TD(0)$ algorithm (Sutton, 1988). Because the value function is a probability, we modify the loss function so as to minimise the cross entropy between the current value and the subsequent value,

$$\delta = r_{t+1} + Q_{RLGO}(s_{t+1}, a_{t+1}) - Q_{RLGO}(s_t, a_t)$$
$$\Delta \theta_i = \frac{\alpha}{|\phi(s_t, a_t)|} \delta \phi_i$$

where δ is the TD-error and α is a step-size parameter.

In the game of Go, the notion of *shape* has strategic importance. For this reason we use binary features
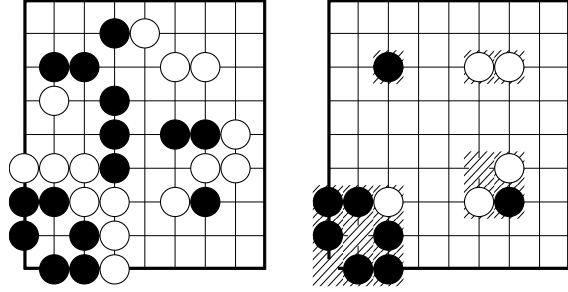

Figure 1. (Left) An example position from a game of 9×9 Go. Black and White take turns to place down stones. Stones can never move, but may be captured if completely surrounded. The player to surround most territory wins the game. (Right) Shapes are an important part of Go strategy. The figure shows (clockwise from top-left) 1×1, 2×1, 2×2 and 3×3 local shape features which occur in the example position.

$\phi(s, a)$ that recognise local patterns of stones (Silver et al., 2007). Each *local shape feature* matches a specific configuration of stones and empty intersections within a particular rectangle on the board (Figure 1). Local shape features are created for all configurations, at all positions on the board, from 1×1 up to 3×3. Two sets of weights are used: in the first set, weights are shared between all local shape features that are rotationally or reflectionally symmetric. In the second set, weights are also shared between all local shape features that are translations of the same pattern.

During training, two versions of the same agent play against each other, both using an ϵ-greedy policy. Each game is started from the empty board and played through to completion, so that the loss is minimised for the on-policy distribution of states. Thus, the value function approximation learns the relative contribution of each local shape feature to winning, across the full distribution of positions encountered during self-play.

5. UCT with a Default Policy

The UCT algorithm has no knowledge beyond the scope of its search tree. If it encounters a state that is not represented in the tree, it behaves randomly. Gelly et al. (Gelly et al., 2006) combined UCT with a *default policy* that is used to complete episodes once UCT leaves the search tree. We denote the original UCT algorithm by $UCT(\pi_{random})$ and this extended algorithm by $UCT(\pi)$ for a default policy π.

The world's strongest 9×9 Computer Go program *MoGo* uses UCT with a hand-crafted default policy π_{MoGo} (Gelly et al., 2006). However, in many domains
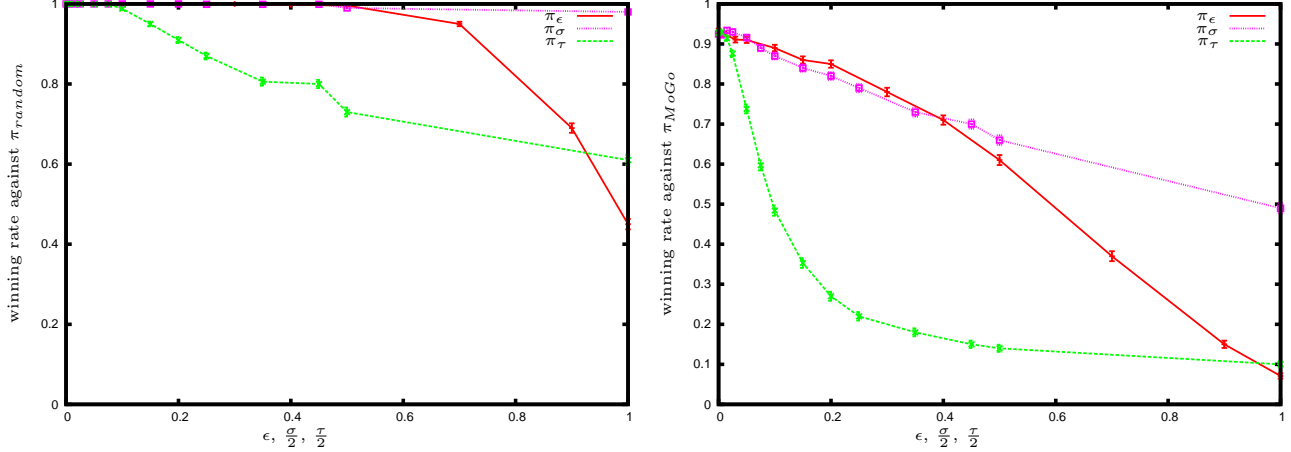
Figure 2. The relative strengths of each class of default policy, against the random policy π_{random} (left) and against a handcrafted policy π_{MoGo} (right). The x axis represents the degree of randomness in each policy.

it is difficult to construct a good default policy. Even when expert knowledge is available, it may be hard to interpret and encode. Furthermore, the default policy must be fast, so that many episodes can be simulated, and a large search tree built. We would like a method for learning a high performance default policy with minimal domain knowledge.

A linear combination of binary features provides one such method. Binary features are fast to evaluate and can be updated incrementally. The representation learned by this approach is known to perform well in many domains (Baxter et al., 1998; Schaeffer et al., 2001). Minimal domain knowledge is necessary to specify the features (Silver et al., 2007; Buro, 1999).

We learn a value function Q_{RLGO} for the domain of 9×9 Go using a linear combination of binary features (see Section 4). It is learned offline from games of self-play. We use this value function to generate a default policy for UCT. As Monte-Carlo simulation works best with a stochastic default policy, we consider three different approaches for generating a stochastic policy from the value function Q_{RLGO}.

First, we consider an ϵ-greedy policy,

$$
\pi_\epsilon(s,a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|} & \text{if } a = \text{argmax}_{a'} Q_{RLGO}(s,a') \\ \frac{\epsilon}{|A(s)|} & \text{otherwise} \end{cases}
$$

Second, we consider a greedy policy over a noisy value function, corrupted by Gaussian noise $\eta(s,a) \sim N(0, \sigma^2)$,

$$
\pi_\sigma(s,a) = \begin{cases} 1 & \text{if } a = \text{argmax}_{a'} Q_{RLGO}(s,a') + \eta(s,a') \\ 0 & \text{otherwise} \end{cases}
$$

Third, we select moves using a softmax distribution with temperature parameter τ,

$$
\pi_\tau(s,a) = \frac{e^{Q_{RLGO}(s,a)/\tau}}{\sum_{a'} e^{Q_{RLGO}(s,a')/\tau}}
$$

We compared the performance of each class of default policy π_{random}, π_{MoGo}, π_ϵ, π_σ, and π_τ. Figure 2 assesses the relative strength of each default policy (as a Go player), in a round-robin tournament of 6000 games between each pair of policies. With little or no randomisation, the policies based on Q_{RLGO} outperform both the random policy π_{random} and *MoGo's* handcrafted policy π_{MoGo} by a margin of over 90%. As the level of randomisation increases, the policies degenerate towards the random policy π_{random}.

Next, we compare the accuracy of each default policy π for Monte-Carlo simulation, on a test suite of 200 hand-labelled positions. 1000 episodes of self-play were played from each test position using each policy π. We measured the MSE between the average return (i.e. the Monte-Carlo estimate) and the hand-labelled value (see Figure 3). In general, a good policy for $UCT(\pi)$ must be able to evaluate accurately in Monte-Carlo simulation. In our experiments with *MoGo*, the MSE appears to have a close relationship with playing strength.

The MSE improves from uniform random simulations when a stronger and appropriately randomised default policy is used. If the default policy is too deterministic, then Monte-Carlo simulation fails to provide any benefits and the performance of π drops dramatically. If the default policy is too random, then it becomes equivalent to the random policy π_{random}.
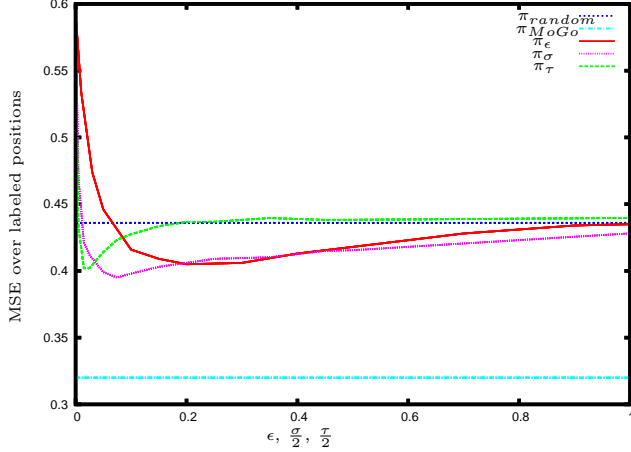
Figure 3. The MSE of each policy π when Monte Carlo simulation is used to evaluate a test suite of 200 hand-labelled positions. The x axis indicates the degree of randomness in the policy.

Algorithm	Wins .v. GnuGo
$UCT(\pi_{random})$	8.88 ± 0.42 %
$UCT(\pi_\sigma)$	$9.38 \pm 1.9\%$
$UCT(\pi_{MoGo})$	$48.62 \pm 1.1\%$

Table 1. Winning rate of the UCT algorithm against GnuGo 3.7.10 (level 0), given 5000 simulations per move, using different default policies. The numbers after the $\pm$ correspond to the standard error from several thousand complete games. π_σ is used with $\sigma = 0.15$.

Intuitively, one might expect that a stronger, appropriately randomised policy would outperform a weaker policy during Monte-Carlo simulation. However, the accuracy of π_ϵ, π_σ and π_τ never come close to the accuracy of the handcrafted policy π_{MoGo}, despite the large improvement in playing strengths for these default policies. To verify that the default policies based on Q_{RLGO} are indeed stronger in our particular suite of test positions, we reran the round-robin tournament, starting from each of these positions in turn, and found that the relative strengths of the default policies remain similar. We also compared the performance of the complete UCT algorithm, using the best default policy based on Q_{RLGO} and the parameter minimising MSE (see Table 1). This experiment confirms that the MSE results apply in actual play.

It is surprising that an objectively stronger default policy does not lead to better performance in UCT. Furthermore, because this result only requires Monte-Carlo simulation, it has implications for other sample-based search algorithms. It appears that the nature of the simulation policy may be as or more important than its objective performance. Each policy has its own bias, leading it to a particular distribution of episodes during Monte-Carlo simulation. If the distribution is skewed towards an objectively unlikely outcome, then the predictive accuracy of the search algorithm may be impaired.

6. Rapid Action Value Estimation

The UCT algorithm must sample every action from a state $s \in \mathcal{T}$ before it has a basis on which to compare

values. Furthermore, to produce a low-variance estimate of the value, each action in state s must be sampled multiple times. When the action space is large, this can cause slow learning. To solve this problem, we introduce a new algorithm UCT_{RAVE}, which forms a *rapid action value estimate* for action a in state s, and combines this online knowledge into UCT.

Normally, Monte-Carlo methods estimate the value by averaging the return of all episodes in which a is selected immediately. Instead, we average the returns of all episodes in which a is selected at *any* subsequent time. In the domain of Computer Go, this idea is known as the *all-moves-as-first* heuristic (Bruegmann, 1993). However, the same idea can be applied in any domain where action sequences can be transposed.

Let $Q_{RAVE}(s, a)$ be the rapid value estimate for action a in state s. After each episode $s_1, a_1, s_2, a_2, ..., s_T$, the action values are updated for every state $s_{t_1} \in \mathcal{T}$ and every subsequent action a_{t_2} such that $a_{t_2} \in \mathcal{A}(s_{t_1})$, $t_1 \le t_2$ and $\forall t < t_2, a_t \ne a_{t_2}$,

$$
\begin{aligned}
m(s_{t_1}, a_{t_2}) &\leftarrow m(s_{t_1}, a_{t_2}) + 1 \\
Q_{RAVE}(s_{t_1}, a_{t_2}) &\leftarrow Q_{RAVE}(s_{t_1}, a_{t_2}) \\
&+ 1/m(s_{t_1}, a_{t_2})[R_{t_1} - Q_{RAVE}(s_{t_1}, a_{t_2})]
\end{aligned}
$$

where $m(s, a)$ counts the number of times that action a has been selected at any time following state s.

The rapid action value estimate can quickly learn a low-variance value for each action. However, it may introduce some bias, as the value of an action usually depends on the exact state in which it is selected. Hence we would like to use the rapid estimate initially, but use the original UCT estimate in the limit. To achieve this, we use a linear combination of the two estimates, with a decaying weight β,
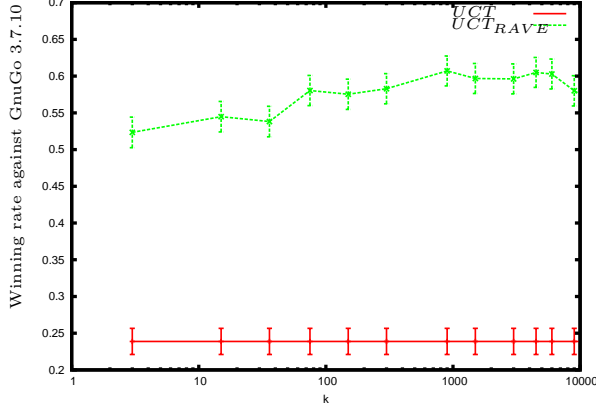
Figure 4. Winning rate of $UCT_{RAVE}(\pi_{MoGo})$ with 3000 simulations per move against GnuGo 3.7.10 (level 8), for different settings of the equivalence parameter k. The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

$$Q^{\oplus}_{RAVE}(s,a) = Q_{RAVE}(s,a) + c\sqrt{\frac{\log m(s)}{m(s,a)}}$$

$$\beta(s,a) = \sqrt{\frac{k}{3n(s)+k}}$$

$$Q^{\oplus}_{UR}(s,a) = \beta(s,a)Q^{\oplus}_{RAVE}(s,a)$$
$$+ (1-\beta(s,a))Q^{\oplus}_{UCT}(s,a)$$
$$\pi_{UR}(s) = \underset{a}{\text{argmax}}\, Q^{\oplus}_{UR}(s,a)$$

where $m(s) = \sum_a m(s,a)$. The *equivalence parameter* k controls the number of episodes of experience when both estimates are given equal weight.

We tested the new algorithm $UCT_{RAVE}(\pi_{MoGo})$, using the default policy π_{MoGo}, for different settings of the equivalence parameter k. For each setting, we played a 2300 game match against GnuGo 3.7.10 (level 8). The results are shown in Figure 4, and compared to the $UCT(\pi_{MoGo})$ algorithm with 3000 simulations per move. The winning rate using UCT_{RAVE} varies between 50% and 60%, compared to 24% without rapid estimation. Maximum performance is achieved with an equivalence parameter of 1000 or more. This indicates that the rapid action value estimate is worth about the same as several thousand episodes of UCT simulation.

7. UCT with Prior Knowledge

The UCT algorithm estimates the value of each state by Monte-Carlo simulation. However, in many cases we have prior knowledge about the likely value of a state. We introduce a simple method to utilise offline knowledge, which increases the learning rate of UCT without biasing its asymptotic value estimates.

We modify UCT to incorporate an existing value function $Q_{prior}(s,a)$. When a new state and action (s,a) is added to the UCT representation $\mathcal{T}$, we initialise its value according to our prior knowledge,

$$n(s,a) \leftarrow n_{prior}(s,a)$$
$$Q_{UCT}(s,a) \leftarrow Q_{prior}(s,a)$$

The number n_{prior} estimates the *equivalent experience* contained in the prior value function. This indicates the number of episodes that UCT would require to achieve an estimate of similar accuracy. After initialisation, the value function is updated using the normal UCT update (see equations 1 and 2). We denote the new UCT algorithm using default policy π by $UCT(\pi, Q_{prior})$.

A similar modification can be made to the UCT_{RAVE} algorithm, by initialising the rapid estimates according to prior knowledge,

$$m(s,a) \leftarrow m_{prior}(s,a)$$
$$Q_{RAVE}(s,a) \leftarrow Q_{prior}(s,a)$$

We compare several methods for generating prior knowledge in 9×9 Go. First, we use an *even-game* heuristic, $Q_{even}(s,a) = 0.5$, to indicate that most positions encountered on-policy are likely to be close. Second, we use a *grandfather* heuristic, $Q_{grand}(s_t,a) = Q_{UCT}(s_{t-2},a)$, to indicate that the value with player P to play is usually similar to the value of the last state with P to play. Third, we use a handcrafted heuristic $Q_{MoGo}(s,a)$. This heuristic was designed such that greedy action selection would produce the best known default policy $\pi_{MoGo}(s,a)$. Finally, we use the linear combination of binary features, $Q_{RLGO}(s,a)$, learned offline by $TD(\lambda)$ (see section 4).

For each source of prior knowledge, we assign an equivalent experience $m_{prior}(s,a) = M_{eq}$, for various constant values of M_{eq}. We played 2300 games between $UCT_{RAVE}(\pi_{MoGo}, Q_{prior})$ and GnuGo 3.7.10 (level 8), alternating colours between each game. The UCT algorithms sampled 3000 episodes of experience at each move (see Figure 5), rather than a fixed time per move. In fact the algorithms have comparable execution time (Table 4).

The value function learned offline, Q_{RLGO}, outperforms all the other heuristics and increases the winning rate of the UCT_{RAVE} algorithm from 60% to 69%.

Algorithm	Wins .v. GnuGo
$UCT(\pi_{random})$	1.84 ± 0.22 %
$UCT(\pi_{MoGo})$	23.88 ±0.85%
$UCT_{RAVE}(\pi_{MoGo})$	60.47 ± 0.79 %
$UCT_{RAVE}(\pi_{MoGo}, Q_{RLGO})$	69 ± 0.91 %

Table 2. Winning rate of the different UCT algorithms against GnuGo 3.7.10 (level 8), given 3000 simulations per move. The numbers after the ± correspond to the standard error from several thousand complete games.

Maximum performance is achieved using an equivalent experience of $M_{eq} = 50$, which indicates that Q_{RLGO} is worth about as much as 50 episodes of UCT_{RAVE} simulation. It seems likely that these results could be further improved by varying the equivalent experience according to the variance of the prior value estimate.

8. Conclusion

The UCT algorithm can be significantly improved by three different techniques for combining online and offline learning. First, a default policy can be used to complete episodes beyond the UCT search tree. Second, a rapid action value estimate can be used to boost initial learning. Third, prior knowledge can be used to initialise the value function within the UCT tree.

We applied these three ideas to the Computer Go program *MoGo*, and benchmarked its performance against GnuGo 3.7.10 (level 8), one of the strongest 9×9 programs that isn't based on the UCT algorithm. Each new technique increases the winning rate significantly from the previous algorithms, from just 2% for the basic UCT algorithm up to 69% using all three techniques. Table 2 summarises these improvements, given the best parameter settings for each algorithm, and 3000 simulations per move. Table 4 indicates the CPU requirements of each algorithm.

These results are based on executing just 3000 simulations per move. When the number of simulations increases, the overall performance of *MoGo* improves correspondingly. For example, using the combined algorithm $UCT_{RAVE}(\pi_{MoGo}, Q_{MoGo})$, the winning rate increases to 92% with more simulations per move. This version of *MoGo* has achieved an Elo rating of 2320 on the Computer Go Online Server, around 500 points higher than any program not based on UCT, and over 200 points higher than other UCT-based programs[1] (see Table 3).

[1] The Elo rating system is a statistical measure of playing strength, where a difference of 200 points indicates an expected winning rate of approximately 75% for the higher rated player

Simulations	Wins .v. GnuGo	CGOS rating
3000	69%	1960
10000	82%	2110
70000	92%	2320*

Table 3. Winning rate of $UCT_{RAVE}(\pi_{MoGo}, Q_{MoGo})$ against GnuGo 3.7.10 (level 8) when the number of simulations per move is increased. The asterisked version used on CGOS modifies the simulations/move according to the available time, from 300000 games in the opening to 20000.

Algorithm	Speed
$UCT(\pi_{random})$	6000 g/s
$UCT(\pi_{MoGo})$	4300 g/s
$UCT(\pi_{\epsilon}), UCT(\pi_{\sigma}), UCT(\pi_{\tau})$	150 g/s
$UCT_{RAVE}(\pi_{MoGo})$	4300 g/s
$UCT_{RAVE}(\pi_{MoGo}, Q_{MoGo})$	4200 g/s
$UCT_{RAVE}(\pi_{MoGo}, Q_{RLGO})$	3600 g/s

Table 4. Number of simulations per second for each algorithm on a P4 3.4Ghz, at the start of the game. $UCT(\pi_{random})$ is faster but much weaker, even with the same time per move. Apart from $UCT(\pi_{RLGO})$, all the other algorithms have a comparable execution speed

The domain knowledge required by this version of *MoGo* is contained in the default policy π_{MoGo} and in the prior value function Q_{MoGo}. By learning a value function offline using $TD(\lambda)$, we have demonstrated how this requirement for domain knowledge can be removed altogether. The learned value function outperforms the heuristic value function when used as prior knowledge within the tree. Surprisingly, despite its objective superiority, the learned value function performs worse than the handcrafted heuristic when used as a default policy. Understanding why certain policies perform better than others in Monte-Carlo simulations may be a key component of any future advances.

We have focussed on the domain of 9×9 Go. It is likely that larger domains, such as 13×13 and 19×19 Go, will increase the importance of the three algorithms. With larger branching factors it becomes increasingly important to simulate accurately, accelerate initial learning, and incorporate prior knowledge. When these ideas are incorporated, the UCT algorithm may prove to be successful in many other challenging domains.

References

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, *47*, 235–256.

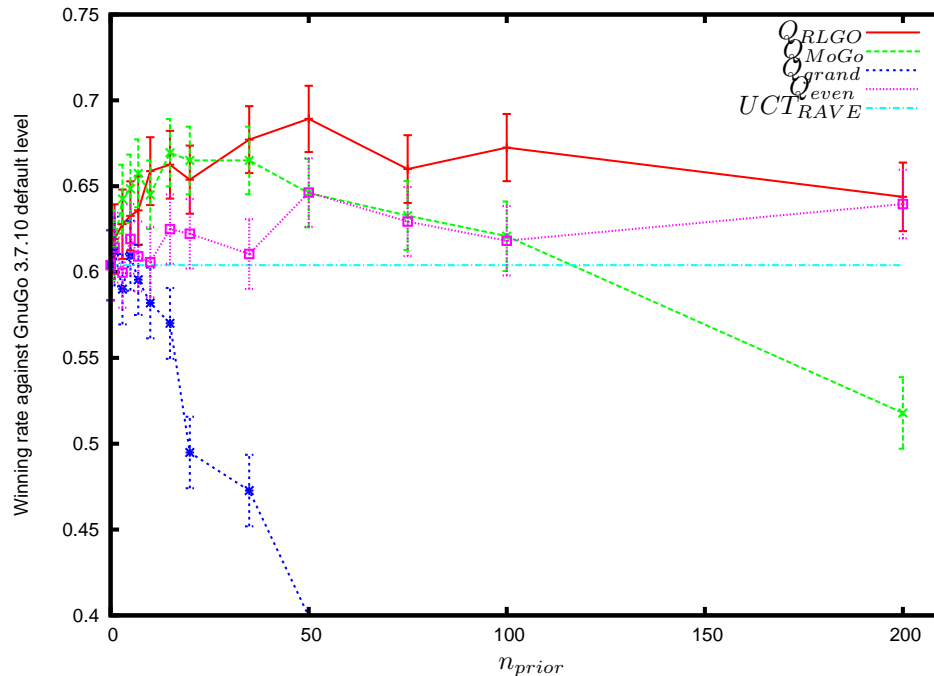Baxter, J., Tridgell, A., & Weaver, L. (1998). Experiments in parameter learning using temporal differ-

Figure 5. Winning rate of $UCT_{RAVE}(\pi_{MoGo})$ with 3000 simulations per move against GnuGo 3.7.10 (level 8), using different prior knowledge as initialisation. The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

ences. *International Computer Chess Association Journal, 21*, 84–99.

Bruegmann, B. (1993). Monte-Carlo Go. *http://www.cgl.ucsf.edu/go/Programs/Gobble.html*.

Buro, M. (1999). From simple features to sophisticated evaluation functions. *1st International Conference on Computers and Games* (pp. 126–145).

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. *5th International Conference on Computer and Games, 2006-05-29*. Turin, Italy.

Enzenberger, M. (2003). Evaluation in Go by a neural network using soft segmentation. *10th Advances in Computer Games Conference* (pp. 97–108).

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). *Modification of UCT with patterns in Monte-Carlo Go* (Technical Report 6062). INRIA.

Kocsis, L., & Szepesvari, C. (2006). Bandit based Monte-Carlo planning. *15th European Conference on Machine Learning* (pp. 282–293).

Schaeffer, J., Hlynka, M., & Jussila, V. (2001). Temporal difference learning applied to a high-performance game-playing program. *17th International Joint Conference on Artificial Intelligence* (pp. 529–534).

Schraudolph, N., Dayan, P., & Sejnowski, T. (1994). Temporal difference learning of position evaluation in the game of Go. *Advances in Neural Information Processing Systems 6* (pp. 817–824). San Francisco: Morgan Kaufmann.

Silver, D., Sutton, R., & Müller, M. (2007). Reinforcement learning of local shape in the game of Go. *20th International Joint Conference on Artificial Intelligence* (pp. 1053–1058).

Sutton, R. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3*, 9–44.

Sutton, R. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *7th International Conference on Machine Learning* (pp. 216–224).

Sutton, R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems 8* (pp. 1038–1044).

Sutton, R., & Barto, A. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press.

Wang, Y., & Gelly, S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii* (pp. 175–182).