Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)

Achieving Master Level Play in 9×9 Computer Go

Sylvain Gelly[*]
Univ. Paris Sud, LRI, CNRS, INRIA, France

David Silver[†]
University of Alberta, Edmonton, Alberta, Canada

Abstract

The UCT algorithm uses Monte-Carlo simulation to estimate the value of states in a search tree from the current state. However, the first time a state is encountered, UCT has no knowledge, and is unable to generalise from previous experience. We describe two extensions that address these weaknesses. Our first algorithm, heuristic UCT, incorporates prior knowledge in the form of a value function. The value function can be learned offline, using a linear combination of a million binary features, with weights trained by temporal-difference learning. Our second algorithm, UCT–RAVE, forms a rapid online generalisation based on the value of moves. We applied our algorithms to the domain of 9×9 Computer Go, using the program MoGo. Using both heuristic UCT and RAVE, MoGo became the first program to achieve human master level in competitive play.

Introduction

Computers have equalled or exceeded the level of human masters in the games of Chess, Checkers and Othello. In these domains, *heuristic minimax search* has provided a silver bullet, allowing the computer to exhaustively search a huge tree of possible continuations. Each leaf position is evaluated using a heuristic function, which encodes handcrafted or machine-learned knowledge about the domain.

One type of heuristic is the *value function*, which is the expected outcome for each position. It can be estimated by a linear combination of binary features: material and pawn structures in Chess (Baxter, Tridgell, & Weaver 1998); material and mobility terms in Checkers (Schaeffer, Hlynka, & Jussila 2001); and disc configurations in Othello (Buro 1999). The weights of these features are trained offline by temporal-difference learning from games of self-play.

Heuristic minimax search has failed to achieve human levels of performance in the more challenging game of Go. The branching factor is large, and accurate heuristic functions have proven difficult to build or learn (Müller 2002). The most successful approaches in Computer Go are based instead on Monte-Carlo tree search algorithms, such as UCT (Kocsis & Szepesvari 2006). Many thousands of games are simulated, using self-play, starting from the current position. Each position in the search tree is evaluated by the average outcome of all simulated games that pass through that position. The search tree is used to guide simulations along promising paths. This results in a highly selective search that is grounded in simulated experience, rather than an external heuristic. Programs using UCT search have outperformed all previous Computer Go programs (Coulom 2006; Gelly *et al*. 2006).

Monte-Carlo tree search algorithms suffer from two sources of inefficiency. First, when a position is encountered for the first time, no knowledge is available to guide the search. Second, each position is represented individually, and there is no generalisation between related positions. We have introduced two new algorithms to address these weaknesses (Gelly & Silver 2007). The first algorithm, heuristic UCT, uses a heuristic function to initialise the values of new positions in the tree. We learn a heuristic by following the principles that have been so successful in other games: we estimate the value function, using a linear combination of a million binary features, trained offline by temporal-difference learning. The second algorithm, UCT–RAVE, forms an online generalisation between related positions, using the average outcome of each move, within a subtree of the search tree. UCT–RAVE combines this rapid but biased estimate of a move's value, with the slower but unbiased Monte-Carlo estimate.

Monte-Carlo Simulation

A policy $\pi(s, a)$ is the probability of selecting move a in position s, and represents a stochastic strategy for playing games. We define the value function $Q^{\pi}(s, a)$ to be the expected outcome of a self-play game that starts with move a in position s, and then follows policy π for both players.

Monte-Carlo simulation provides a simple method for estimating the value of positions and moves. N games are simulated using self-play, starting from position s and playing a candidate move a, and then continuing until the end of the game using policy π. The estimated value $Q(s, a)$ is the average outcome of all simulations in which move a was selected in position s,

$$Q(s, a) = \frac{1}{n(s, a)} \sum_{i=1}^{N} I_i(s, a) z_i, \qquad (1)$$

[*]Now at Google, Zurich

[†]Research supported by Alberta Ingenuity and iCore.

where z_i is the outcome of the ith simulation: $z = 1$ if the player wins the game and $z = 0$ otherwise. $I_i(s, a)$ is an indicator function returning 1 if move a was selected in position s during the ith simulation, and 0 otherwise. $n(s, a) = \sum_{i=1}^{N} I_i(s, a)$ counts the total number of simulations in which move a was selected in position s. For example, if the move a was played in 10 out of 100 simulations starting from state s, of which 7 were won and 3 were lost, then we would have $N = 100, n(s, a) = 10$ and $Q(s, a) = 0.7$.

This gives a simple search algorithm for identifying the best move in the current position s_t. Each candidate move in s_t is evaluated using Monte-Carlo simulation, and the move with the highest estimated value is played.

Monte-Carlo Tree Search

Monte-Carlo tree search builds a search tree of positions and moves that are encountered during simulations from the current position. There is one node in the tree for each position s and move a, containing a count $n(s, a)$, and a value $Q(s, a)$ that is estimated by Monte-Carlo simulation (equation 1). Each simulation starts from the current position s_t, and is divided into two stages: a *tree policy* is used while all children are contained in the search tree; and a *default policy* is used for the remainder of the simulation. The simplest version of Monte-Carlo tree search uses a greedy tree policy during the first stage, that selects the move with the highest value among all child nodes; and a uniform random default policy during the second stage, that rolls out simulations until completion. The tree is grown by one node per simulation, by adding the first position and move in the second stage.

Monte-Carlo tree search can be extended by enhancing the policy used in each stage of the simulation. A domain specific default policy can significantly boost the performance of Monte-Carlo algorithms (Gelly *et al*. 2006). The tree policy can be significantly improved by encouraging exploratory moves. The UCT algorithm treats each individual position as a multi-armed bandit (Kocsis & Szepesvari 2006), and selects the move a in position s with the greatest upper confidence bound $Q^{\oplus}(s, a)$ on its value,

$$Q^{\oplus}(s, a) = Q(s, a) + c\sqrt{\frac{\log n(s)}{n(s, a)}}. \qquad (2)$$

where $n(s)$ counts the total number of visits to position s, and c is a constant determining the level of exploration. Similarly, opponent moves are selected by minimising a lower confidence bound on the value.

Typically, many thousands of games are simulated per move, building an extensive search tree from the current position s_t. There is a cycle of policy improvement and evaluation: as the tree grows, the tree policy becomes more informed; and as the tree policy improves, the values in the tree become more accurate. 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,

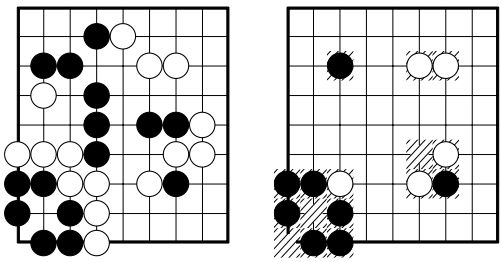

Figure 1: (a) An example position from a game of 9×9 Go. (b) Shapes are an important part of Go strategy. The figure shows (clockwise from top-left) $1 \times 1, 2 \times 1, 2 \times 2$ and 3×3 local shape features which occur in the example position.

UCT requires no prior domain knowledge to evaluate positions or order moves. Furthermore, the UCT search tree is highly 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.

Heuristic UCT

Monte-Carlo tree search evaluates positions and moves by simulating experience from the current position s_t. However, when only a few relevant simulations have been experienced, the value estimates have high variance. Furthermore, the search tree grows exponentially with depth, and thus the vast majority of nodes in the tree are experienced rarely.

One way to reduce the uncertainty for rarely encountered positions is to incorporate prior knowledge, by using a heuristic function $Q_h(s, a)$. When a node is first added to the search tree, it is initialised according to the heuristic function, $Q(s, a) = Q_h(s, a)$ and $n(s, a) = n_h(s, a)$. The quantity n_h indicates our confidence in the heuristic in terms of *equivalent experience*. This corresponds to the number of simulations that UCT would require to achieve a value of similar accuracy[1]. After initialisation, the value and count are updated using standard Monte-Carlo simulation.

Learning a Heuristic for Computer Go

We learn a *value function* as a heuristic, that estimates the expected outcome of the game. The value function is approximated by a linear combination of binary features ϕ with weights θ,

$$Q_h(s, a) = \sigma\left(\phi(s, a)^T \theta\right) \qquad (3)$$

where σ is the logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$, which squashes the linear combination into the range $[0, 1]$. We use temporal difference learning (Sutton 1988) to update the weights, trained offline from games of self-play,

$$\begin{aligned} \delta &= Q_h(s_{t+1}, a_{t+1}) - Q_h(s_t, a_t) \\ \Delta\theta_i &= \alpha\delta\phi_i(s_t, a_t) \end{aligned}$$

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

[1]This is equivalent to a beta prior if binary outcomes are used.

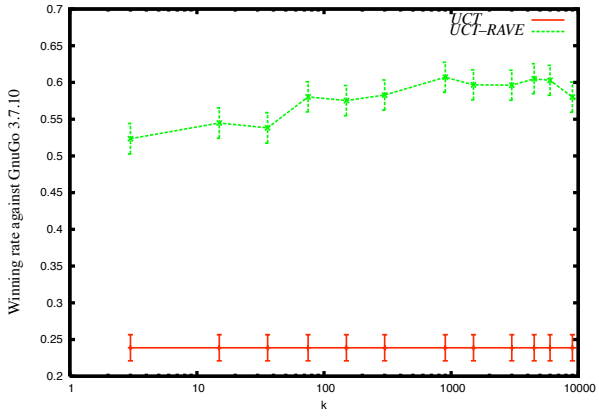

Figure 2: Winning rate of UCT–RAVE with 3000 simulations per move against GnuGo 3.7.10 (level 10), for different settings of the equivalence parameter k. The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

In the game of Go, the notion of *shape* has strategic importance. For this reason we use binary features $\phi(s,a)$ that recognise local patterns of stones (Silver, Sutton, & Müller 2007). Each *local shape feature* matches a specific configuration of stones and empty intersections within a particular square on the board (Figure 1). Local shape features are created for all configurations, at all positions on the board, from size 1×1 up to 3×3. Rotational, reflectional and translational symmetries are exploited by sharing weights between equivalent features.

Rapid Action Value Estimation (RAVE)

UCT learns a unique value for each node in the search tree, estimated online from experience simulated from the current position. However, it cannot generalise between related positions. The RAVE algorithm provides a simple way to share experience between classes of related positions, resulting in a rapid, but biased value estimate.

The RAVE value $\hat{Q}(s,a)$ is the average outcome of all simulations in which move a is selected in position s, or in any subsequent position,

$$\hat{Q}(s,a) = \frac{1}{\hat{n}(s,a)} \sum_{i=1}^{N} \hat{I}_i(s,a) z_i, \qquad (4)$$

where $\hat{I}_i(s,a)$ is an indicator function returning 1 if position s was encountered at any step k of the ith simulation, and move a was selected at any step $t >= k$, or 0 otherwise. $\hat{n}(s,a) = \sum_{i=1}^{N} \hat{I}_i(s,a)$ counts the total number of simulations used to estimate the RAVE value.

The RAVE value $\hat{Q}(s,a)$ generalises the value of move a across all positions in the subtree below s, and subsequent positions encountered during the default policy. It is closely related to the *history heuristic* in alpha-beta search (Schaeffer 1989), and the *all moves as first* heuristic in Computer Go (Bruegmann 1993).

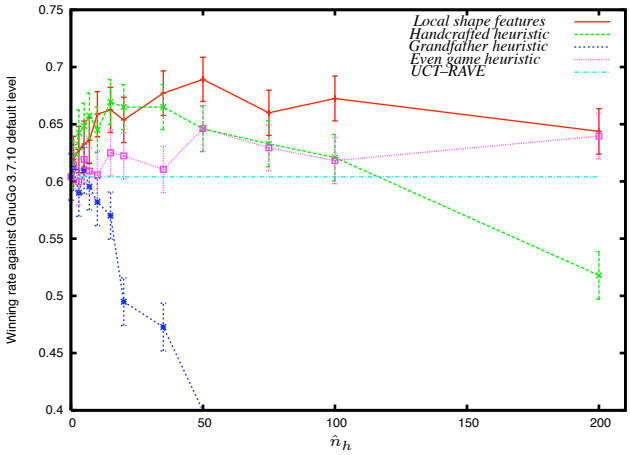

Figure 3: Winning rate of heuristic UCT–RAVE with 3000 simulations per move against GnuGo 3.7.10 (level 10), using different heuristics, and using a constant level of equivalent experience $\hat{n}_h$. The bars indicate the standard error. Each point of the plot is an average over 2300 complete games.

Consider a maze in which the agent starts on the left and searches for an exit on the right. The search quickly branches into two regions: in the top of the maze, moving right and down typically leads to success; in the bottom of the maze, moving right and up are usually most successful. The RAVE algorithm enables the agent to generalise from these observations. When the agent encounters an unseen state, it knows to try the actions that were previously successful in this region of the search tree.

Combining UCT and RAVE

The Monte-Carlo value $Q(s,a)$ is unbiased, but may be high variance if insufficient experience is available. The RAVE value $\hat{Q}(s,a)$ is biased, but lower variance; it is based on more experience, but this generalisation may not always be appropriate. Hence we rely predominantly on the RAVE value initially, but gradually shift to the Monte-Carlo value, by using a linear combination of these values with a decaying weight . An *equivalence parameter* k controls the number of simulations when both estimates are given equal weight (Gelly & Silver 2007).

The results of UCT–RAVE are shown in Figure 2. The winning rate against GnuGo (level 10) reaches 60%, compared to 24% without RAVE. Maximum performance is achieved with an equivalence parameter of 1000 or more. This indicates that each RAVE value is worth several thousand Monte-Carlo simulations.

Heuristic UCT–RAVE

The heuristic UCT and UCT–RAVE algorithms can be combined by initialising the RAVE values to the heuristic function, $\hat{Q}(s,a) = Q_h(s,a)$ and $\hat{n}(s,a) = \hat{n}_h(s,a)$. The quantity $\hat{n}_h$ indicates our confidence in the heuristic in terms of

equivalent experience for the RAVE value.

We compared the performance of four different heuristic functions: 1. A learned value function using local shape features. 2. A handcrafted heuristic based on MoGo's domain specific knowledge. This heuristic was designed such that greedy move selection would produce the best known default policy for MoGo, as described by Gelly *et al*. (2006). 3. A grandfather heuristic that assumes that the current value is similar to the overall value (after all simulations were completed) last time the agent was to play, $Q_h(s_t, a_t) = Q(s_{t-2}, a_t)$. 4. An *even-game* heuristic, $Q_h(s, a) = 0.5$, that assumes even positions are more likely than clearly won or lost positions. Figure 3 shows the performance of MoGo against GnuGo 3.7.10, using heuristic UCT–RAVE for constant values of $\hat{n}_h$, for all four heuristic functions.

Heuristic UCT–RAVE with a good heuristic significantly outperformed the basic UCT–RAVE algorithm. Local shape features provided the best performance across a wide range of equivalent experience, and appear to be worth around 50 simulations of experience[2]. However, a poor heuristic can hurt performance, particularly when it is trusted too heavily - as demonstrated by the misleading grandfather heuristic.

Conclusion

Monte-Carlo tree search algorithms such as UCT provide a new framework for search. Instead of requiring human knowledge to evaluate leaf positions, evaluation is grounded in the outcomes of simulations. Instead of considering all possible continuations, simulations are sampled selectively. Instead of explicitly computing the minimax solution, it emerges from an incremental process of policy improvement and evaluation within the search tree. In large, challenging domains such as Computer Go, these differences appear to be critical to success.

The UCT algorithm can be signficantly improved by using a heuristic to provide prior knowledge within the search tree, and by using a simple generalisation to accelerate online learning. Each improvement is cumulative (see Table 1), and scalable (see Table 2).

Using a combination of heuristic UCT and UCT–RAVE, the program MoGo achieved a rating of 3–*dan* (strong master level) in 9×9 Go games against human players on the Kiseido Go Server; became the highest rated program on both 9×9 and 19×19 Computer Go servers; won the gold medal at the 2007 Computer Go Olympiad; and became the first program to beat a human professional player at 9×9 Go.

Algorithm	Default Policy	Wins .v. GnuGo
UCT	Uniform random	1.84 ± 0.22 %
UCT	MoGo	23.88 ± 0.85%
UCT–RAVE	MoGo	60.47 ± 0.79 %
Heuristic UCT–RAVE	MoGo	69 ± 0.91 %

Table 1: Winning rate of the different UCT algorithms against GnuGo 3.7.10 (level 10), given 3000 simulations per move. The numbers after the $\pm$ correspond to the standard error from several thousand complete games.

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

Table 2: Winning rate of heuristic UCT–RAVE against GnuGo 3.7.10 (level 10) for increasing number of simulations, and MoGo's rating on the Computer Go Online Server.

References

Baxter, J.; Tridgell, A.; and Weaver, L. 1998. Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal* 21(2):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. In *First International Conference on Computers and Games*, 126–145.

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *5th International Conference on Computer and Games, 2006-05-29*, 72–83.

Gelly, S., and Silver, D. 2007. Combining online and offline learning in UCT. In *17th International Conference on Machine Learning*, 273–280.

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

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

Müller, M. 2002. Computer Go. *Artificial Intelligence* 134:145–179.

Schaeffer, J.; Hlynka, M.; and Jussila, V. 2001. Temporal difference learning applied to a high-performance game-playing program. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 529–534.

Schaeffer, J. 1989. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-11(11):1203–1212.

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

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

[2]Subsequent versions of the handcrafted heuristic outperform the learned heuristic, and are used in the competitive MoGo.