Fast Evolutionary Adaptation for Monte Carlo Tree Search

Simon M. Lucas, Spyridon Samothrakis and Diego Perez
University of Essex, UK

Abstract

This paper describes a new adaptive Monte Carlo Tree Search (MCTS) algorithm that uses evolution to rapidly optimise its performance. An evolutionary algorithm is used as a source of control parameters to modify the behaviour of each iteration (i.e. each simulation or roll-out) of the MCTS algorithm; in this paper we largely restrict this to modifying the behaviour of the random default policy, though it can also be applied to modify the tree policy.

This method of tightly integrating evolution into the MCTS algorithm means that evolutionary adaptation occurs on a much faster time-scale than has previously been achieved, and addresses a particular problem with MCTS which frequently occurs in real-time video and control problems: that uniform random roll-outs may be uninformative.

Results are presented on the classic Mountain Car reinforcement learning benchmark and also on a simplified version of Space Invaders. The results clearly demonstrate the value of the approach, significantly outperforming "standard" MCTS in each case. Furthermore, the adaptation is almost immediate, with no perceptual delay as the system learns: the agent frequently performs well from its very first game.

1 Introduction

Monte Carlo Tree Search (MCTS) is a powerful selective search method that has had a profound impact on Game AI since its introduction in 2006 by a number of researchers; see the recent survey paper by Browne et al [3] for more details of its history, algorithm, variations and applications.

One of the most appealing features of MCTS is that it can operate without the need for any heuristic: it works reasonably well in its vanilla form on a variety of problems. However, it is also well known and not surprising that the appropriate use of heuristics can significantly boost performance, and all leading Go programs use these.

MCTS selectively builds an asymmetric tree. The algorithm works by following a tree policy until it finds a node to expand, at which point it performs a roll-out (also called play-out or simulation) until the end of the game (or until some other stopping condition is met). The value found at the end of the roll-out is then back-propagated up the tree, updating the mean value and the number of visits to each node. Perhaps the most popular tree policy is based on the Upper Confidence Bounds equation (UCB) which for MCTS is known as UCT (Upper Confidence bounds for Trees). This aims to optimally balance exploitation (visit the child of the current node with the best mean value, left term in equation 1) versus exploration (visit the least explored child, right term in equation 2).

$$UCB1_c = \mu_c + k\sqrt{\frac{\ln N}{n_c}} \tag{1}$$

where k is an exploration constant, N is the number of times the parent has been visited, and for child c, μ_c is the mean value and n_c the number of visits.

Although some efforts have already been made to incorporate automated learning procedures into MCTS, the current state of the art usually involves a great deal of hand-programming and leaves some important problems largely unanswered, namely:

- The action-space may be too fine-grained; it may be necessary to work in some space of macro-actions in order to perform well. Designing the macro-actions could be done by evolution.
- Uniform random roll-outs may cause insufficient exploration of the state space. They may all end in a similar or even identical degree of failure, rendering them devoid of information. In the worst case, the μ_c values for each child may be identical, meaning there is nothing to exploit.

In this paper we address the second of these problems: this is important since it will aid the development of general video game bots able to play a wide range of games to a high standard without being explicitly programmed for any particular game. This is useful for providing an automatic range of opponents for new video games, and also for evaluating automatically designed video games. Although there have been some very interesting efforts along these lines, for instance [15] [4] [5], the richness of the games that have been evolved so far has been arguably limited by the intelligence of the evolved bots [15], or the NPC rules [4] or the search algorithms used to evaluate them [5].

2 Related Research

Silver et al [13] incorporated temporal difference learning (TDL) into an MCTS algorithm, and drew the distinction between the transient values learned by the MCTS procedure and the long-term heuristic information learned by TDL.

Robles et al [11] used a similar procedure for learning in Othello, where they used TDL to learn a value function both for controlling the tree policy and for controlling the roll-outs.

Although TDL utilises more of the available information during learning than evolution [6], evolution can be more robust due to its direct emphasis on the end goal (such as winning the most games) rather than some proxy of this such as minimising the residual errors.

Evolutionary learning has also been used to tune MCTS algorithms. Benbassat and Sipper [2] used Genetic Programming, in conjunction with MCTS, for several classic games such Othello and Dodgem. In their work, each individual in the evolutionary algorithm represents a function that evaluates a board position. During the rollout step of MCTS, each move is chosen by selecting the action that maximizes the value of the next board state, according to this function.

Independently, Alhejali and Lucas [1] used evolution to tune the weights of the heuristic value function used to guide the roll-outs in an MCTS Pac-Man player. In both these approaches evolution was able to improve on the default MCTS performance, though in both cases the evolutionary algorithm was applied at the level of the individual, where each fitness evaluation involved playing one or more complete games. This approach leads to relatively expensive evolutionary runs, since for most games reasonable standard MCTS players cannot operate much faster than real-time. The approach developed in this paper is very different, since now each roll-out contributes immediately to the fitness evaluation of the policy that guided it.

The main use of the fast evolutionary adaptation used in this paper is to bias the simulation or roll-out policy after the tree policy has found a leaf node to expand. Most previous ways of doing this have relied on the information gained from the simulations; a good example is the recent work by Powley et al [10], where they learn n-gram models to bias the roll-outs. This works well when the simulations are informative, but breaks down when the simulations all terminate in identical or very similar values.

The approach developed in this paper is intended to complement the roll-out mining methods by initially hypothesizing useful “directions” for the roll-outs to take in the absence of any evidence. As the evidence accumulates, the aim is for the evolutionary algorithm to adapt the distribution of roll-out policies accordingly.

One of the most general approaches for optimizing MCTS algorithms is that of Maes et al [7] where they formulate a grammar for describing a general class of Monte Carlo Tree Search algorithms and then search the space induced by that grammar to find high performance ones. However, as with most other methods this requires relatively extensive evaluation in order to determine the fitness of each algorithm instance.

Recently MCTS has found application in video games, with Ms Pac-Man being a good example [12], supported by strong results in competitions both for controlling the Pac-Man agent [9] and the ghost team [8]. However, all these cases relied on some hand-designed heuristics such as disallowing Pac-Man reversals during roll-outs. This was found to be necessary since if the Pac-Man is allowed to reverse then it makes insufficient progress through the maze due to excessive dithering.

This is analogous to the problem observed running vanilla MCTS on the Mountain Car problem described below, and in this case is easily solved by the fast evolutionary adaptation approach.

3 Fast Evolutionary MCTS

The main contribution of this paper is the introduction of a new approach to using an evolutionary algorithm to rapidly adapt the behaviour of an MCTS algorithm. The main idea is to tightly integrate evolution’s fitness evaluation process with the MCTS algorithm. Previous evolutionary approaches (e.g. [2], [1]) have been loosely coupled in the sense that each fitness evaluation was based on the performance of the MCTS agent over an entire game or set of games, where the MCTS agent was seen as a black box with a set of tunable parameters.

In the Fast Evolutionary method each iteration (roll-out) of the MCTS algorithm contributes directly to a statistical evaluation of an individual, where each individual is characterised by a vector of parameters. As a result of this change the evolutionary algorithm has access to a much higher bandwidth of information and consequently is able to adapt more rapidly.

Within this fast evolutionary approach there are at least two distinct ways in which it could work: evaluate each individual within the same MCTS tree, or create a new MCTS tree for each individual.

The former approach aggregates the statistics of each individual within the same tree and has the advantage of throwing nothing away. The latter approach is more wasteful since each time an individual is discarded from the population all the statistics are lost; however it can also be used more flexibly and can be used to search the space of different macro-actions for example. In this paper we limit the investigation to the former approach.

Algorithm 1 outlines the main steps. The *while* loop describes the MCTS algorithm executed in order to make each decision. Here the condition is listed as being within a computational budget: this could be measured as elapsed time or as a fixed number of iterations.

For each iteration a parameter vector w is drawn from the evolutionary algorithm by calling evo.getNext() as shown on line 2. Line 3 initialises a statistics object to track the performance of this control vector. The *for* loop (line 4) is there to enable a particular MCTS control policy to be sampled K times before returning its performance statistics to the evolutionary algorithm. There are many statistics that can be used to rate how well an MCTS algorithm is performing: our basic statistics object includes calculation of the mean, standard deviation, min and max. All these can be important, though in this initial study we only use the mean. Alternatively, the K parameter can be seen as the responsibility of the evolutionary algorithm, in which case the *for* loop can be removed.

The parameter vector could be used to control both the tree policy and the default policy as indicated on lines 5 and 6 respectively, with the default policy being used to generate a roll-out that ends in a state with the value of Δ. Apart from the influence of the control parameters, the MCTS algorithm operates as normal with line 7 showing the backup of the tree statistics. Line 8 indicates the statistics object S being updated with the roll-out value Δ.

After running the MCTS algorithm for the allowed computational budget, the *while* loop exits. The algorithm returns the estimate of the best control vector found to date via a call to $evo.getBest()$ (line 12). This suggests another use case for the algorithm: to find good control vectors and then use these to *fix* the bias. In the results tables below we refer to this mode of use as *Pre-Evolved*.

Finally, the algorithm returns the selected action for the current root state using a recommendation policy (line 13), which is usually different from the tree policy. In this paper we mainly choose the action with the highest mean value, though for Space Invaders we also experimented with biasing the recommendation directly.

Algorithm 1: Fast Evolutionary MCTS. The evolutionary algorithm provides a source of parameter sets used to control the MCTS algorithm.

```
   input   : Parameter $K$, the number of roll-outs per fitness evaluation, $v_0$ is root state
   output  : weight vector $\mathbf{w}$ and action $a$
   ;                                                  // initialize evolutionary algorithm evo,
1  while within computational budget do
2      Set $\mathbf{w} \leftarrow$ EVO.GETNEXT()
3      Initialise statistics object $S$.
4      for $i$ := 1 to $K$ do
5          $v_l \leftarrow$ TREEPOLICY($v_0, T(w)$) ;          // Tree policy is influenced by $T(w)$
6          $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l), D(w)$) ;   // Default policy is influenced by $D(w)$
7          BACKUP($v_l, \Delta$)
8          UPDATESTATS($S, \Delta$)
9      end
10     EVO.SETFITNESS($\mathbf{w}, S$)
11 end
12 Return $\mathbf{w} \leftarrow evo.getBest()$
13 Return $a \leftarrow recommend(v_o)$
```

3.1 Biasing Rollouts

The main idea here is to use features associated with a given state to bias the action selection process. The biasing process works as follows: we map from state space S to feature space F with N features and then from feature space to a probability distribution over the set of actions. This is currently implemented using a hand-coded feature space for each problem. There are A actions available and the relative strength a_i of each action i is then calculated as a weighted sum of feature values. The weights are stored in a matrix W where entry w_{ij} is the weighting of feature j for action i:

$$a_i = \sum_{j=1}^{N} w_{ij} f_j \tag{2}$$

These relative action values then feed into a softmax function in order to calculate the probability $P(a_i)$ of taking each action.

$$P(a_i) = \frac{e^{-a_i}}{\sum_{j=1}^{A} e^{-a_j}} \tag{3}$$

The bias is therefore controlled by two things: the features and the weight matrix W. As previously mentioned, for the moment the features are hand-coded though in future they could be evolved using GP or auto-constructed in some other way. The weight matrix is evolved: every roll-out is biased using a W drawn from the evolutionary algorithm.

4 Test Problems

For proof of concept we choose two initial test problems: Mountain Car and Space Invaders. The first one is a simple reinforcement learning problem, but one that MCTS with uniform roll-outs fails on badly. Space Invaders is a more interesting challenge, and even the simplified version used in this paper involves precise shooting of fast moving targets (the aliens move quickly when there are only a few left), and strategic considerations regarding the order in which to shoot the aliens. In each case the MCTS tree policy was UCB1 with the exploration constant k set to 0.3 after some experimentation. The algorithm ran for 200 iterations per action selection. When calculating the mean values of each child in the UCT tree we tried scaling the scores to be in a smaller range, but this tended to degrade performance.

We used a $(1 + 1)$ Evolution Strategy (ES) for the evolutionary algorithm (i.e. the source of roll-out control vectors). This is the simplest possible choice, and most likely far from optimal. A better choice might be to use a bandit-based algorithm in order to maintain a multi-modal distribution of roll-out policies. Nonetheless, even the (1+1) ES is able to produce some interesting results.

4.1 Mountain Car

The mountain car problem is a classic reinforcement learning benchmark problem; here we use a version identical to that described by Sutton and Barto [14] (page 214) apart from limiting the number of steps per episode to 500 instead of 2,500. The problem is illustrated in Figure 1: the aim is to reach the line at the top of the hill on the right, but the engine has insufficient force to overcome gravity. The state of the system is fully specified by two scalar values: position s and velocity v. The state space is small but continuous and there are many ways of constructing features for this. For these experiments we take the most direct approach and simply use s and v scaled to be in the same range from -1 to $+1$. The three possible actions are accelerate left, neutral and accelerate right.

The difficulty of any particular instance of this problem depends on the initial state. For example, if the car starts close to the goal with a large velocity towards the goal then many action sequences will lead to success. All experiments in this paper used a start state of $(s = -0.3, v = 0)$. Starting in this way, close to the centre of the valley and with zero velocity, is relatively hard and a few oscillations are required in order to reach the goal. We limit the number of steps in each episode to 500, and the score (to be minimised) is simply the number of steps taken to reach the goal, or 500 if the goal was never reached. Configured in this way MCTS with uniform random roll-outs reaches the goal around 1 in 30 episodes.

4.1.1 Analysing Trajectories

Figure 2 shows 20 random roll-outs using (a) uniform random actions and (b) random actions biased by Equations 2 and 3, where the weights of matrix W were drawn from a Gaussian distribution with zero mean and a standard deviation of 5. Each illustrative roll-out lasted for $1,000$ steps (though for the experiments, we limited episode length to 500). This clearly illustrates the value of the approach. When taking uniform random moves none of the roll-outs reached the goal and therefore, in the standard mountain car reward scheme, would each have a value of -1000 (-1 for each step).

The biases introduce a more directed policy: sometimes this is even worse than the uniform policy but sometimes it is much better, and plot (b) shows several trajectories reaching the goal.

4.1.2 Results

Table 1 shows three sets of results based on the roll-out bias. Each roll-out ran until a terminal state was reached. Uniform roll-outs perform worst, with a mean of 497 and only 4 successes out of 100. The fast evolutionary method (Fast-Evo) reaches the goal in all but one case. From the 100 fast evolutionary runs we saved the bias matrix W with the best result and performed 100 trials with this Pre-Evolved bias. This gave the best result with a mean of 99 and no failures.

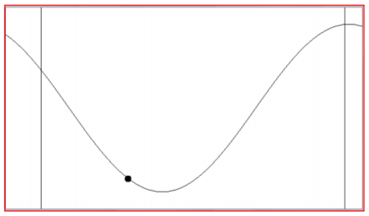

Figure 1: A depiction of the mountain car reinforcement learning benchmark. The objective is to get to the top of the hill on the right, but the force of the engine is insufficient to directly overcome gravity. To solve the problem (depending on the start state) it is usually necessary to accelerate away from the goal and up the left hill before accelerating toward the goal.

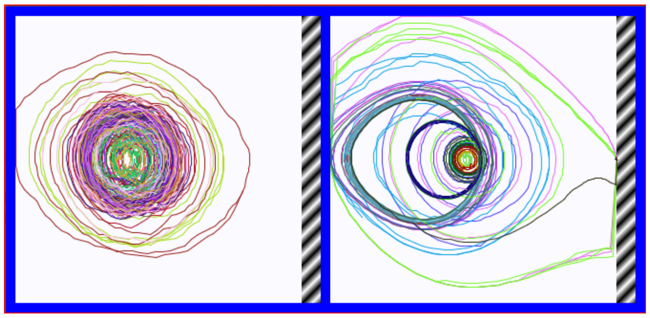

Figure 2: Random roll-outs through the two-dimensional state space (position: horizontal, velocity: vertical) of the Mountain Car problem: (a) uniform random roll-outs are unlikely to reach the goal and wander through the state-space with no purpose. (b) biased roll-outs encourage more purposeful trajectories through state space, some of which may reach the goal. The set of goal states is shown as the hatched area to the right of each plot.

.

Table 1: Mean scores and standard errors for each method based on 100 trials each. The score is the number of steps taken to reach the goal state, so lower scores are better. Each episode was terminated after 500 steps, so the worst possible score is 500. An episode was deemed successful if it found the goal in under 500 steps.

Roll-out	Mean Score (s.e.)	Successes
Pre-Evolved	99 (2.8)	100
Fast-Evo	233 (13)	99
Uniform Random	497 (1.8)	4

4.2 Space Invaders

Space invaders was released by Taito in 1978 and is one of the classic arcade games of all time, taking gameplay to new levels. There is still significant interest in developing better versions of this type of game, as evidenced by the highly playable and commercially successful Space Invaders Extreme published by SquareEnix for the Sony PlayStation Portable (PSP). The original ROM code is available on line and can be played using the Multi Arcade Machine Emulator (MAME). We encourage the interested reader to try this: the original game is superior to all of the clones we have found on the Web.

Suitable MCTS agents could be used to play-test variations of these games to assess the difficulty of each level and also to feed into the fitness function when automatically evolving new variants. However, in this paper we use the game as an initial benchmark.

Figure 3 shows a screenshot with an MCTS software agent playing the game. This version has the following features:

- The same number of aliens as the original game: 55 arranged in 11 columns, 5 rows.
- Similar movement patterns. On each tick of the game loop just one alien is moved, each missile is moved, and the player cannon is moved. This leads to the dog-legged movement pattern that can be observed in the original game, and naturally leads to the effect of the aliens moving more quickly as more are shot - with extremely fast movement when just one alien is left. Note that many clones of the game ignore this feature and move the aliens together in lock-step.
- Currently there are no alien missiles: the game is over either when an alien lands (reaches the bottom of the screen) or when all the aliens have been shot.
- The aliens are of three types (as with the original game) differing only in the score for shooting each one: scores are 10, 20, 30 for cyan (bottom two rows), magenta (next two rows) and blue (top row) respectively.
- No alien flying saucers along the top. In the original game these were worth between 50 and 300 points, and one strategy involved shooting out some middle columns in order to ensure a clear shot at the flying saucers. Our version is currently missing these.

Despite the limitations compared to the original game, the version used in this paper is nonetheless an appropriate challenge for the MCTS players under study. Actually, the game required some tuning in order to make the difficulty suitable for clearly distinguishing between weak and strong players. We did this by slowing down the speed of the player's missiles[1], and by lowering the starting point of the block of aliens. The latter difficulty adjustment happens in the original game, with the aliens starting lower down as the levels progress. This means they have to be cleared in a more constrained order to prevent them from landing.

Here the problem of constructing suitable features is much more complicated than for the mountain car problem. There are many elements to good Space Invaders strategy, and sometimes it is desirable to shoot away the end columns, but on other occasions emergency measures are needed and to avoid immediate death it is necessary to shoot away the aliens closest to landing. After some agonising over the best choice of features we made some initial experiments with just a single feature! We call this *nearest edge column displacement* and calculate it as follows. First, we find the minimum (leftmost) and maximum (rightmost) x-coordinates of the set of aliens. We then pick the one closest to the player's missile cannon and subtract the x-coordinate of the cannon.

The fact that this worked rather well was a surprise, but provides interesting insight into the nature of biasing roll-outs for MCTS. The fact is that MCTS is already a powerful adaptive algorithm, and the roll-out bias is just needed to nudge it into more

[1]Only one player missile can be fired at a time so this limits the rate at which aliens can be shot, and increases the punishment for missing, since the player must wait until the missile has left the top of the screen before firing the next one.

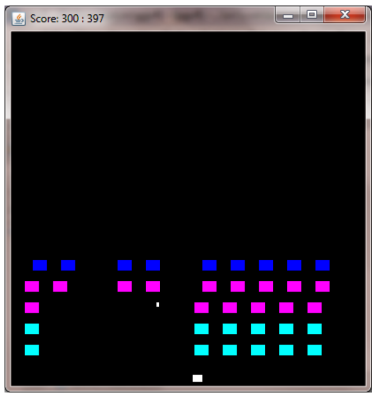

Figure 3: A Space Invaders game in progress. The aim of this is to shoot all the alien invaders before they land. In this cut-down version there are no bases and the invaders do not drop missiles. Nonetheless when play-testing the game we found that clearing the level required a reasonable level of shooting skill and also that a suitable strategy be employed such as shooting away the end columns first. In the depicted game the AI player has made the mistake of shooting away too many of the central aliens, and the aliens look set to land.

interesting regions of the search space. It may be unnecessary for the roll-out bias to be especially clever.

There are six possible actions, formed by the cartesian product of the movement actions $\{$left, dontMove, right$\}$ and the firing action $\{$dontShoot, shoot$\}$. Since there is only one feature this leads to 6 weights to learn in the matrix W.

4.2.1 Results

We tested a number of approaches. Given the simplicity of the parameters to learn, we were able to include a hand-coded set of parameters. The intuition behind these is to bias the roll-outs in order to move to the closest end column most of the time, firing occasionally.

When designing the roll-out bias by hand we also observed a frustrating aspect of this process: although the roll-outs were biased, the behaviour of the agent very often failed to reflect this. The reason for this is that the actions involving more movement may not necessarily lead to better scores, and hence may not be selected at the root level.

In order to force the effect of the bias we also created an option of adding the bias directly into the recommendation policy (i.e. the move actually chosen to play). We refer to this as $\mu + Q$ action selection. We were also interested to see the effects of not using MCTS at all, but simply playing uniform random moves, or random moves according to the hand-coded bias.

Each roll-out ran to a maximum depth of 50 from the root or until the end of the game, whichever condition was met first. This meant that every move in the game required a maximum of 10,000 game-ticks to be simulated; in our simulator this achieves real-time performance at 50 frames (actions) per second.

Table 2 shows the mean and standard error of these variations. The difference in scores between methods is significant (t-test, p = 0.01) if separated by a horizontal line. The MCTS approaches are described by the roll-out policy and the recommendation policy. The biases are: Q_{hand}: hand-designed, Q_{evo}: evolved for each of the 100 trials using Algorithm 1 and Q_{prevo}: a fixed high scoring bias matrix selected from the 100 trials of the Q_{evo} method. The Q_{evo} approach sometimes (about 5 - 10% of the time) obtains the maximum score of 990; we just selected an arbitrary one of these solutions to fix the Q_{prevo} bias.

The results are interesting. The first thing to note is that the non-MCTS methods perform poorly: clearly it is not enough just to make uniform or biased random moves. Secondly, the best MCTS approach was the hand-coded one with action selection bias. Interestingly, evolution was able to find some good solutions, but not on every run (remember here that an evolutionary run corresponds to a single game being played). The high performance of Q_{prevo} is very encouraging.

5 Conclusions

This paper introduced a novel fast evolutionary algorithm for adapting Monte Carlo Tree Search. The algorithm has an important role to play in real-time control problems

Table 2: Mean scores and standard errors for each method based on 100 games each. The maximum possible score is 990. The minimum possible score is zero.

Roll-out	Action selection	Mean Score (s.e.)
Q_{hand}	$\mu + Q_{hand}$	953 (20)
Q_{prevo}	μ	885 (11)
Q_{hand}	μ	877 (17)
Q_{evo}	μ	683 (19)
Uniform Random	μ	674 (16)
Q_{evo}	$\mu + Q_{evo}$	593 (23)
—	Uniform Random	127 (5.1)
—	Biased Random Q_{hand}	119 (6.2)

and video games where uniform random roll-outs may be uninformative. To counter this the evolutionary algorithm is used as a source of roll-out policy control vectors to encourage more decisive simulations that explore more diverse parts of the state space. When it works this enables the MCTS algorithm to work with more informative statistics.

We tested the algorithm on the Mountain Car RL benchmark, and on a reduced but interesting version of space invaders. The algorithm learns extremely quickly and can adapt the roll-outs to great effect during the playing of a single game. The estimated best control-vectors can also be used to fix the bias for a set of runs, a process we call pre-evolving the bias, and this led to good results on both problems under test.

So far the algorithm has been learning a small number of parameters — just six in each case, yet appropriate setting of these was sufficient to significantly improve performance on both test problems. Future work includes more thorough testing of the method, including cases involving complex feature sets with large numbers of parameters to tune.

Given the fact that simple features can lead to significant performance boosts, and the fact that they can be evaluated so rapidly, this suggests that GP could work well for automated feature construction.

References

[1] A. Alhejali and S. Lucas, "Using Genetic Programming to Evolve Heuristics for a MonteCarlo Tree Search Ms Pac-Man Agent," in *IEEE Conference on Computational Intelligence and Games*, 2013, pp. 65 – 72.

[2] A. Benbassat and M. Sipper, "EvoMCTS: Enhancing MCTS-Based Players through Genetic Programming," in *IEEE Conference on Computational Intelligence and Games*, 2013, pp. 57 – 64.

[3] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte

Carlo Tree Search Methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[4] M. Cook and S. Colton, "Multi-faceted Evolution Of Simple Arcade Games," in *Computational Intelligence in Games (CIG), IEEE Conference on*, 2011, pp. 289 – 296.

[5] M. Cook, S. Colton, A. Raad, and J. Gow, "Mechanic Miner: Reflection-Driven Game Mechanic Discovery and Level Design," in *Computational Intelligence in Games (CIG), IEEE Conference on*, 2013, pp. 284 – 293.

[6] S. Lucas, "Investigating learning rates for evolution and temporal difference learning," in *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, Dec 2008, pp. 1–7.

[7] F. Maes, D. St-Pierre, and D. Ernst, "Monte Carlo Search Algorithm Discovery for Single-Player Games," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 5, no. 3, pp. 201–213, 2013.

[8] K. Q. Nguyen and R. Thawonmas, "Monte Carlo Tree Search for Collaboration Control of Ghosts in Ms. Pac-Man," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 5, no. 1, pp. 57–68, 2013.

[9] T. Pepels and M. Winands, "Enhancements for Monte-Carlo Tree Search in Ms Pac-Man," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 2012, pp. 265–272.

[10] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Bandits all the way down: UCB1 as a simulation policy in Monte Carlo Tree Search," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 2013, pp. 81–88.

[11] D. Robles, P. Rohlfshagen, and S. M. Lucas, "Learning Non-Random Moves for Playing Othello: Improving Monte Carlo Tree Search," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 305–312.

[12] S. Samothrakis, D. Robles, and S. Lucas, "Fast Approximate Max-n Monte Carlo Tree Search for Ms Pac-Man," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 2, pp. 142–154, 2011.

[13] D. Silver, R. S. Sutton, and M. Müller, "Sample-Based Learning and Search with Permanent and Transient Memories," in *Proc. 25th Annu. Int. Conf. Mach. Learn.*, Helsinki, Finland, 2008, pp. 968–975.

[14] R. Sutton and A. Barto, *Introduction to Reinforcement Learning*. MIT Press, 1998.

[15] J. Togelius and J. Schmidhuber, "An Experiment in Automatic Game Design," in *IEEE Symposium on Computational Intelligence and Games*, 2008, pp. 111 – 118.