Evolving UCT Alternatives for General Video Game Playing

Ivan Bravi, Ahmed Khalifa, Christoffer Holmgård, Julian Togelius
New York University, Tandon School of Engineering
ivan.bravi@nyu.edu, ahmed.khalifa@nyu.edu, holmgard@nyu.edu, julian@togelius.com

Abstract

We use genetic programming to evolve alternatives to the UCB1 heuristic used in the standard UCB formulation of Monte Carlo Tree Search. The fitness function is the performance of MCTS based on the evolved equation on playing particular games from the General Video Game AI framework. Thus, the evolutionary process aims to create MCTS variants that perform well on particular games; such variants could later be chosen among by a hyper-heuristic game-playing agent. The evolved solutions could also be analyzed to understand the games better. Our results show that the heuristic used for node selection matters greatly to performance, and the vast majority of heuristics perform very badly; furthermore, we can evolve heuristics that perform comparably to UCB1 in several games. The evolved heuristics differ greatly between games.

1 Introduction

Monte Carlo Tree Search (MCTS) is a popular and effective algorithm for planning and game playing, which has in particular seen successes in general game playing, i.e. playing unseen games where no a priori domain information is possible [Kocsis and Szepesvári, 2006; Browne *et al.*, 2012]. In its most common formulation, the algorithm builds a search tree through exploring nodes in a best-first manner, and every time a new node is explored a random playout of the game/planning problem is performed to stochastically estimate the value of the node; values are also propagated up to all ancestors of a node. At the core of the MCTS algorithm is the UCB1 equation which allows the algorithm to balance between exploration and exploitation.

A large number of modifications to the basic MCTS algorithm have been advanced to deal with particular games and other problem domains. The modifications that make MCTS work better for a particular problem might very well make it worse for another problem, meaning that domain knowledge is necessary to invent and select the right modification for a given problem. The literature contains a large number of MCTS modifications, a handful of which are listed in a well-known survey paper [Browne *et al.*, 2012]. While some of these modifications change the way the overall MCTS algorithm works or focuses on particular aspects such as the roll-out, others change the UCB1 equation itself [Jacobsen *et al.*, 2014].

As exploring new variants of the MCTS algorithm is a currently fruitful area, one wonders whether it would be possible to automate some version of this research. In other words: automate the invention of such modifications. Furthermore, one wonders if it would be possible to automatically create MCTS variations that are specifically tailored to particular games or problems. This could be useful for example in agents based on hyper-heuristics, that would select appropriate MCTS variations for particular games/problems [Burke *et al.*, 2013]. But it could also be useful for understanding the characteristics of a particular game/problem through finding which MCTS variation performs best at it; in other words analyzing the problem through finding strategies for solving it.

In this paper we propose using genetic programming to evolve replacement equations for the UCB1 equation. The idea is that different versions of, or alternatives to, UCB1 might make MCTS more suitable for particular problems, and that we can find such alternatives or versions automatically. For our testbed problems, we use the games in the General Video Game Playing framework. We show that for several games, we can find replacements for UCB1 that makes MCTS play the game as well.

2 Background

This section reviews the background on Monte Carlo Tree Search, how it has been combined with evolution, genetic programming and general video game playing.

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a relatively recently proposed algorithm for planning and game playing. It is a tree search algorithm which selects which nodes to explore in a best-first manner, which means that unlike Minimax (for two-player games) and breadth-first search (for single-player games) Monte Carlo Tree Search focuses on promising parts of the search tree first, while still conducting targeted exploration of under-explored parts. This balance between exploitation and exploration is usually handled through the application of the Upper Confidence Bound for Trees (UCT) algorithm which applies UCB1 to the search tree.

The basic formulation of UCB1 is given in Equation 1, but many variations exist for different games [Auer *et al.*, 2002; Browne *et al.*, 2012; Park and Kim, 2015].

$$UCB1 = \overline{X_j} + 2C_p\sqrt{\frac{2\ln n}{n_j}} \quad (1)$$

These variations change UCB1 by e.g. optimizing it for single-player games or incorporating feature selection to name a few variations. However, when we use MCTS for general game playing it becomes impossible to know if we are better off using "plain UCB" or some specialized version, since we do not know which game we will be encountering.

Ideally, we need some way of searching through the different possible variations of tree selection policies to find one that is well suited for the particular game in question. We propose addressing this problem by evolving tree selection policies to find specific formulations that are well suited for specific games. If successful, this would allow us to automatically generate adapted versions of UCB for games we have never met, potentially leading to better general game playing performance.

2.2 Combinations of evolution and MCTS

Evolutionary computation is the use of algorithms inspired by Darwinian evolution for search, optimization, and/or design. Such algorithms have a very wide range of applications due to their domain-generality; with an appropriate fitness function and representation, evolutionary algorithms can be successfully applied to optimization tasks in a variety of fields.

There are several different ways in which evolutionary computation could be combined with MCTS for game playing. Perhaps the most obvious combination is to evolve game state evaluators. In many cases, it is not possible for the rollouts of MCTS to reach a terminal game state; in those cases, the search needs to "bottom out" in some kind of state evaluation heuristic. This state evaluator needs to correctly estimate the quality of a game state, which is a non-trivial task. Therefore the state evaluator can be evolved; the fitness function is how well the MCTS agent plays the game using the state evaluator. This is done routinely for Minimax search and has been done several times in the literature for MCTS [Pettit and Helmbold, 2012].

Of particular interest for the current investigation is Cazenave's work on evolving UCB1 alternatives for Go [Cazenave, 2007]. It was found that it was possible to evolve heuristics that significantly outperformed standard UCB formulations; given the appropriate primitives, it could also outperform more sophisticated UCB variants specifically aimed at Go. While successful, Cazenave's work only concerned a single game, and one which is very different from a video game.

At first sight, it would seem that evolutionary algorithms and MCTS are very different kinds of algorithms used for very different purposes. However, it has recently emerged that they can be used for very similar purposes. MCTS has been used for content generation [Browne, 2011; Browne *et al.*, 2012] and continuous optimization [McGuinness, 2016]. Evolutionary algorithms have also been used for real-time planning in single-player [Perez *et al.*, 2013] and two-player games [Justesen *et al.*, 2016].

This points to the ability of both MCTS and evolutionary search to focus limited computational resources on the most promising parts of a large search space, given simple metrics of outcomes and, in the case of MCTS, indications of how to balance exploration and exploitation of promising areas found during the search. More fundamentally, it raises the question whether these two algorithms are similar on some deeper level– perhaps there could even be a framework defining a space with MCTS at one end and a genetic algorithm or evolution strategy on the other.

2.3 General Video Game Playing

The field of General Video Game Playing (GVGP) is an extension of General Game Playing (GGP) [Levine *et al.*, 2013] which focuses on asking computational agents to play unseen games. Agents are evaluated on their performance on a number of games which the designer of the agent did not know about before submitting the agent. GVGP focuses on real time games compared to board games (turn based) in General Game Playing.

In this paper, we use the General Video Game AI framework (GVGAI), which is the software framework associated with the GVGAI competition [Perez *et al.*, 2015; Perez-Liebana *et al.*, 2016]. In the learning track of the GVGAI competition, competitors submit agents which are scored on playing ten unseen games which resemble (and in some cases are modeled on) classic arcade games from the seventies and eighties.

It has been shown in the past that for most of these games, simple modifications to the basic MCTS formulation can provide significant performance improvements. However, these modifications are non-transitive; a modification that increases the performance of MCTS on one game is just as likely to decrease its performance on another [Frydenberg *et al.*, 2015]. This points to the need for finding the right modification for each individual game, either manually or automatically.

We selected five different games from the framework as testbeds for the tree selection policy evolution:

- **Boulderdash:** is a VGDL port of Boulderdash. The player's goal is to collect at least ten diamonds then reach the goal while avoiding getting killed either by enemies or boulders.
- **Zelda:** is a VGDL port of The legend of Zelda dungeon system. The player's goal is to reach the exit without getting killed by enemies. The player can kill enemies using its sword.
- **Missile Command:** is a VGDL port of Missile Command. The player's goal is to protect at least one city building from being destroyed by the incoming missiles. The player can move around and destroy missiles by attacking them.
- **Solar Fox:** is a VGDL port of Solar Fox. The player's goal is to collect all the diamonds and avoid hitting the side walls or the enemy bullets. The player is always moving like a missile which makes it harder to control.

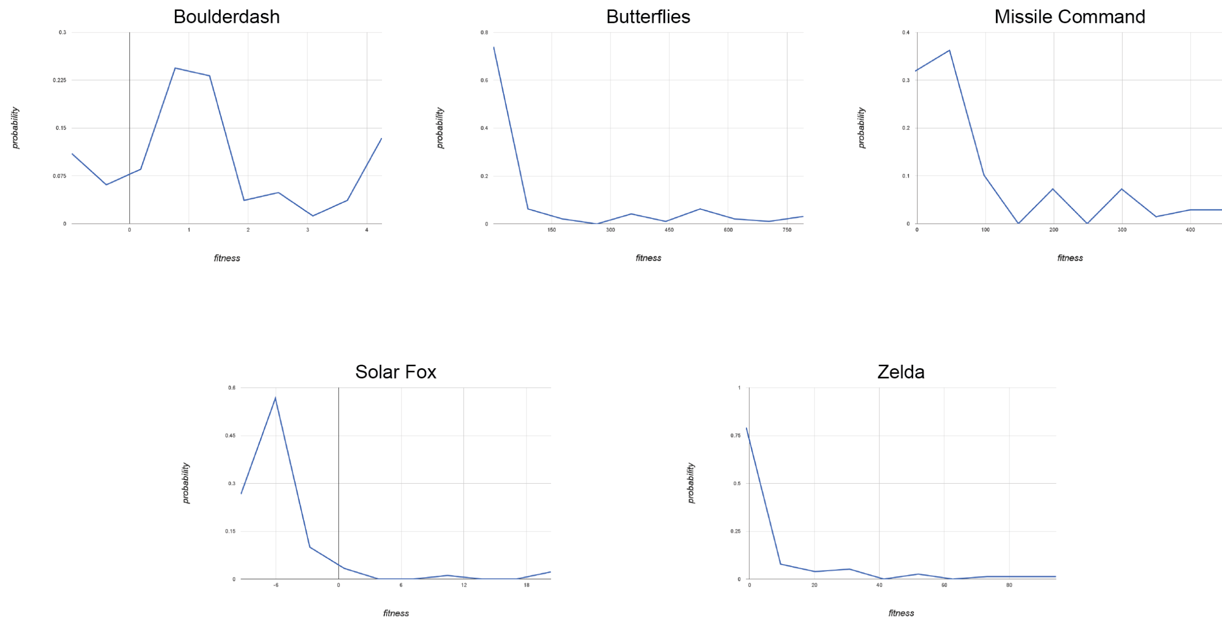

Figure 1: Distribution of fitness of random UCT_+ equations over five games

- **Butterflies:** is an arcade game developed for the framework. The player's goal is to collect all the butterflies before they destroy all the flowers.

These games require very different strategies from agents for successful play and together provide varied testbeds for the approach. They also have in common that standard MCTS with the UCB1 equation does not play the game perfectly (or even very well) and that other agents have been shown to play the game better in the past.

2.4 Genetic Programming

Genetic Programming (GP)[Poli *et al.*, 2008] is a branch of evolutionary algorithms[Bäck and Schwefel, 1993] which evolves computer programs as a solution to the current problem. GP is essentially the application of genetic algorithms (GA)[Whitley, 1994] to computer programs. Like GAs, GP evolves solutions based on Darwinian theory of evolution. A GP run starts with a population of possible solutions called chromosomes. Each chromosome is evaluated for its fitness (how well it solves the problem). New chromosomes are generated using genetic operators, such as crossover and mutation, from the current chromosomes, creating a new population. This process is repeated until a given termination condition is met.

In GP, chromosomes are most commonly represented as syntax trees where inner nodes are functions (e.g. addition, subtraction, if-condition, ...etc) while leaf nodes are terminals (e.g. constants, variables, ...etc). Fitness is calculated by running the current program and see how well it solves the problem. GP uses Crossover and Mutation to evolve the new chromosomes. Crossover in GP combines two different programs at a selected node by swapping the subtrees at these nodes. Mutation in GP alters the selected node value to a new suitable value.

3 Methods

The chromosome representation in our GP algorithm is a syntax tree where the nodes represent either *unary* or *binary functions* while the leaves are either *constants* values or *variables*. The binary functions are addition, subtraction, multiplication, division and power. The unary functions are square root, absolute value, multiplicative inverse and natural logarithm. The constant values come from a set of possible values ranging from -30 to 30. Namely: 0.1, 0.25, 0.5, 1, 2, 5, 10, 30, -0.1, -0.25, -0.5, -1, -2, -5, -10, -30. The formula can contain variables belonging to two sets: $TreeVariable$ set and $AgentVariable$ set. The variables regarding the state of tree built by MCTS belong to the Tree Variables set, namely: child depth, child value, child visits, parent visits and child max value. Instead the Agent Variables set contains the variables related to the agent's behavior like: history reverse value, it represents the number of opposite actions taken w.r.t. the current; history repeating value, times the current action has been repeated; useless value, number of actions that don't produce any effect; exploration value, number of times the current position has been visited before; exploration max value, it represents the number of times the most visited position has been visited.

The fitness function is based on two parameters derived from the simulation of 100 playthroughs of one level of the game: *win ratio* and *average score*. Equation 2 shows how these two parameters are combined. We followed the same competition rules as *win ratio* have higher priority than *aver-*

age score.

$$Fitness = 1000 * win_ratio + avg_score \quad (2)$$

We use a rank based selection to choose the chromosomes to generate the new chromosomes. The offsprings are created as follows. Two chromosomes are selected from the current generation, then a subtree crossover is performed and finally a mutation operator is applied between: point mutation, subtree mutation and constant mutation. Point mutation selects a node with 5% probability and swap it with a node of the same type, the types are unary node, binary node, variable and constants. The subtree mutation, instead, selects a subtree and substitutes it with a random tree of randomly distributed depth between 1 and 3. Finally, the constants mutation selects a *constant* node from the tree and selects a new value from the set of available constant values. Both trees derived from the two trees selected are put in the new generation.

The population of the first generation is composed of 1 $UCB1$ chromosome and 99 random chromosomes. We added the $UCB1$ equation in the initial population to push the GP to converge faster and find something better. We run the GP for 30 generation. In each generation, we use a 10% elitism to guarantee that the best chromosome is carried out to the next generation.

The number of repetitions per fitness evaluation is high enough to give a decent evaluation within a reasonable amount of time.

Once gathered the results from the genetic algorithm we pick the best tree evolved and we verify its validity by running a simulation over 2000 playthroughs.

For Solar Fox and Zelda we evolved a UCT_{+} formula using only the Tree Variable set. While for Boulderdash, Butterflies and Missile Command we evolved two new formulae: UCT_{+}, using only the Tree Variable set; and UCT_{++}, using both the Tree Variable and Agent Variable sets.

4 Results

In this section, we explore the implications of replacing the UCB1 equation with alternative equations in five games from the publicly available training set in the GVGAI framework.

For each game, we first attempted replacing the UCB1 equation with random replacement equations. This was done to investigate the impact the UCB1 equation itself has, and the range of performance exhibited by the basic MCTS algorithm when its core equation is changed. Our experiments showed that the replacement resulted in agents with very low performance, as shown in Table 1. Figure 1 shows the distribution of the fitness of these random equations over all five games.

In the following we describe the results of evolving new UCB1 replacements following the process outlined above. For each game we describe the best equation found, compare its performance to that of UCB1, and discuss what this says about the role of UCB1 and about the AI problem posed by the specific game.

Table 2 compares the $UCB1$ equation with the generated alternatives (UCB_{+} and UCB_{++}) and only exploitation term ($\overline{X_j}$) for each different game. The data is collected from 2000 runs using $UCB1$, $\overline{X_j}$, UCB_{+}, and UCB_{++} for all

Missile Command

Metric	min	max	$25\%ile$	$50\%ile$	$75\%ile$
Win ratio	0	0.5	0.02	0.07	0.13
Score	-3	0.49	-2.31	-1.9	-1.46

Butterflies

Metric	min	max	$25\%ile$	$50\%ile$	$75\%ile$
Win ratio	0	0.84	0.02	0.04	0.08
Score	2.18	40.5	25.46	27.58	30.56

Solar Fox

Metric	min	max	$25\%ile$	$50\%ile$	$75\%ile$
Win ratio	0	0.2	0	0	0
Score	-9.36	3.64	-7.15	-5.19	-4.52

Boulderdash

Metric	min	max	$25\%ile$	$50\%ile$	$75\%ile$
Win ratio	0	0	0	0	0
Score	-0.97	4.83	0.77	1.34	2.22

Zelda

Metric	min	max	$25\%ile$	$50\%ile$	$75\%ile$
Win ratio	0	0.1	0	0	0
Score	-1	5.84	-0.31	0.14	1.77

Table 1: Results from using random chromosome to define tree policies across the testbed games used in this paper. As is evident, these randomly controlled agents achieve very low performances.

the five games. The performance of $\overline{X_j}$ is slightly worse or equal in all games except win ration in *Solarfox*. The performance of UCB_{+} is nearly similar to the $UCB1$ over score and wins. UCB_{+} achieves higher win rate over *Butterflies* and *Zelda* while worse in *Missile Command* and *Solarfox*. UCB_{+} achieves higher mean score over *Butterflies* and *Solarfox* while worse in the three remaining games. The performance of UCB_{++} is always better than $UCB1$ in both wins and score except for score in *Butterflies*. In this table, we compare the mean scores and win ratio for $UCB1$, $\overline{X_j}$, UCB_{+}, and UCB_{++} for each game. The scores are compared using the Mann-Whitney U test for scores and the Chi Squared test applied to the absolute number of wins out of 2000. For ease of reading, double asterisk means $p - value < 0.01$ while single asterisk means $p - value < 0.05$. Figure 2 shows the increase of average fitness over generations during evolving UCB_{+} for all five games. Figure 3, instead, shows the increase of average fitness over generations during evolving UCB_{++} for Boulderdash, Butterflies and Missile Command.

4.1 Boulderdash

$$UCB_{+} = maxValue(maxValue^{9}d_j + 1) - 0.25 \quad (3)$$

The evolved Equation 3 pushes MCTS to exploit more without having any exploration. The reason is Boulderdash map is huge compared to other games with a small amount of diamonds scattered throughout the map. GP finds exploiting the best path is far more better than wasting time steps in exploring the rest of the tree.

$$UCB_{+} = maxValue + d_j + \frac{1}{E_j} + 1.25 \quad (4)$$

The evolved Equation 4 consists of two exploitation terms and one exploration term. The exploitation term tries to focus on the deepest

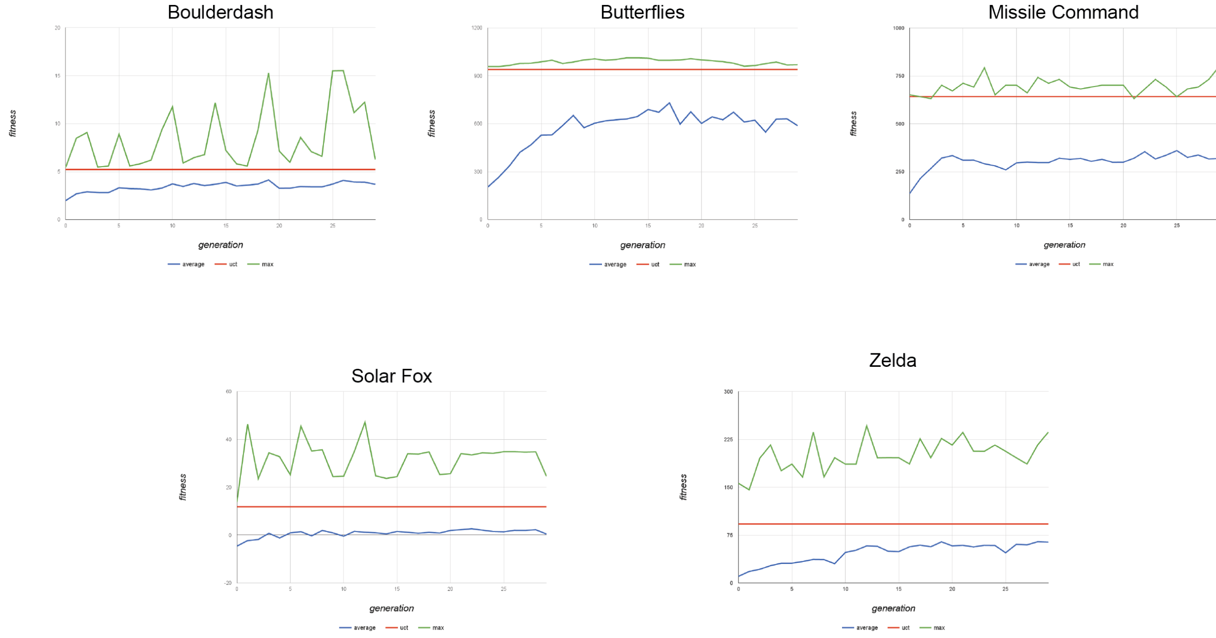

Figure 2: Average fitness over generations over all five games

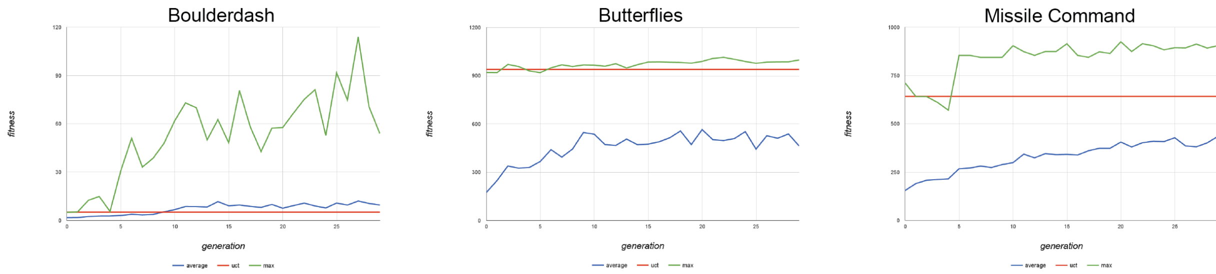

Figure 3: Average fitness over generations for Boulderdash, Butterflies and Missile Command

explored node with the highest value, while the exploration pushes MCTS to explored nodes that are least visited in the game space.

4.2 Butterflies

$$UCB_+ = \overline{X_j} + \frac{1}{n_j^2 \cdot \sqrt{d_j}(d_j \cdot \sqrt{0.2 \cdot d_j \cdot maxValue} + 1)} \tag{5}$$

The evolved Equation 5 is similar to MCTS with exploitation and exploration terms. The exploitation term is similar to MCTS while exploration term is more complex. The exploration term tries to explore the shallowest least visited nodes in the tree with the least maximum value. The huge map of the game with butterflies spread all over it leads MCTS to explore the worst shallowest least visited node. The value of the exploration is very small compared to the exploitation term so it will only differentiate between similar valued nodes.

$$UCB_{++} = \sqrt{maxValue} + 2\overline{X_j} - \frac{X_j}{R_j} - \left(\frac{\ln X_j}{\sqrt{maxExp + X_j^{-0.25}}} + \sqrt{U_j}\right)^{maxExp} \tag{6}$$

The evolved Equation 6 is similar to MCTS with mixmax modification [Frydenberg *et al.*, 2015]. The first two terms resemble the mixmax with different balancing between average child value and maximum child value. The other two terms force the MCTS to search for nodes with the least useless moves and with the most number of reverse moves. The useless forces the agent to go deeper in branches that have more moves, while the number of reverse moves in butterfly force the agent to move similar to the butterflies in the game which leads to capture more of them.

4.3 Missile Command

$$UCB_+ = \overline{X_j} + \left(10 + \frac{X_j^{X_j}}{n}\right)^{-1/\ln n} \tag{7}$$

Game	Tree Policy	Mean	Median	Min	Max	SD	Win ratio
Boulderdash	$UCB1$	5.30	4.00	0	186	5.22	0
Boulderdash	$\overline{X_j}$	4.83**	3.00	0	18	3.00	0
Boulderdash	UCB_+	5.05**	4.00	0	18	2.85	0
Boulderdash	UCB_{++}	19.51	3.00	0	1580	117.72	0.0182**
Butterflies	$UCB1$	37.39	32.00	8	86	18.92	0.902
Butterflies	$\overline{X_j}$	37.04	32.00	8	88	18.78	0.852**
Butterflies	UCB_+	36.34	30.00	8	88	18.68	0.89
Butterflies	UCB_{++}	35.84**	30.00	8	80	18.43	0.914
Missile Command	$UCB1$	2.88	2.00	2	8	1.37	0.641
Missile Command	$\overline{X_j}$	2.57**	2.00	2	5	1.18	0.409**
Missile Command	UCB_+	3.03*	2.00	2	8	1.44	0.653
Missile Command	UCB_{++}	4.95**	5.00	2	8	2.13	0.785**
Solarfox	$UCB1$	6.31	5.00	0	32	6.06	0.00565
Solarfox	$\overline{X_j}$	6.30	5.00	0	32	5.84	0.00633
Solarfox	UCB_+	6.49	5.00	0	32	5.81	0.0075
Zelda	$UCB1$	3.58	4.00	0	8	1.85	0.088
Zelda	$\overline{X_j}$	3.58	4.00	0	8	1.85	0.064**
Zelda	UCB_+	6.32**	6.00	0	8	1.26	0.155**

Table 2: Descriptive statistics for all the tested games. Mean, Median, Min, Max, and SD all relate to the score attained using $UCB1$, $\overline{X_j}$, UCB_+, and UCB_{++}, respectively. Wins simply indicates the number of wins out of 2000 possible obtained with either policy.

The evolved Equation 7 has the same exploitation term as $UCB1$. Although the second term is very complex, it forces MCTS to pick nodes with less value. This second term is very small compared to the first term so its only affecting when two nodes have nearly similar values.

$$UCB_{++} = \overline{X_j} + \frac{maxValue}{n \cdot E_j \cdot (2X_j)^{0.2n_j}} \cdot \left(d_j - \frac{1}{2/maxValue + 2U_j/X_j + 2\ln X_j + 1/n}\right)^{-1} \quad (8)$$

The evolved Equation 8 have the same exploitation term from $UCB1$. Although the second term is very complex, it forces MCTS to explore the least spatially visited node with the least depth. This solution is most likely evolved due to the simplicity of *Missile Command* which allows GP to generate an overfitted equation that suits this particular game.

4.4 Solarfox

$$UCB_+ = \overline{X_j} + \frac{\sqrt{d_j}}{n_j} \quad (9)$$

The evolved Equation 9 is a variant of the original UCB1 equation. The exploitation term is the same while the exploration term is simplified to select the deepest least selected node regardless of anything else.

4.5 Zelda

$$UCB_+ = (n + maxValue)^{(n+maxValue)} \quad (10)$$

The evolved Equation 10 is pure exploitation. This equation selects the node with maximum value. This new equation leads the player to be more courageous which leads to higher win rate and higher score than standard $UCB1$.

5 Discussion and Conclusion

We have described an experiment in evolving node selection heuristics to replace UCB1 in individual GVGAI games. The goal has not been to find a better alternative to UCB1 in general, rather to find alternatives that can exploit the properties of individual games. Such alternatives could then be used in a hyper-heuristic approach to create a general agent, and also studied to understand the characteristics of the individual problem.

Our results show that changing the node selection heuristic has very large effects on the performance of MCTS-based agents, with most heuristics performing very poorly. Evolved heuristics perform on par with UCB1 for all game save one. Analyzing the evolved heuristics shows a large variety, though in almost all cases both an exploration and an exploitation term can be discerned. Almost all the evolved UCB equations kept the exploitation term in some way or another while the second term varied from being total exploration to endorse more exploitation. This makes sense, for if an agent would know the real score for each node, the best playing algorithm is a greedy algorithm which exploits the best path. For some games, the state evaluation heuristic is simply more accurate, suggesting that exploration can be downplayed. These variations reflect basic properties of the games, as expected; e.g. games with little need for exploration downplay this element. Using more variables in evolving the equation grants us better evolved equations than the original $UCB1$

In the future, it would be interesting to explore the inclusion of additional variables in the evolved heuristics and apply on more games. For example, using parameters that is tailored for VGDL. It is likely that the advantages of evolving node selection heuristics will come out more clearly when provided with more primitives to build such heuristics from.

References

[Auer *et al.*, 2002] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Ma-*

chine learning, 47(2-3):235–256, 2002.

[Bäck and Schwefel, 1993] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993.

[Browne *et al.*, 2012] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[Browne, 2011] Cameron Browne. Towards mcts for creative domains,". In *Proc. Int. Conf. Comput. Creat., Mexico City, Mexico*, pages 96–101, 2011.

[Burke *et al.*, 2013] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[Cazenave, 2007] Tristan Cazenave. Evolving monte carlo tree search algorithms. *Dept. Inf., Univ. Paris*, 8, 2007.

[Frydenberg *et al.*, 2015] Frederik Frydenberg, Kasper R Andersen, Sebastian Risi, and Julian Togelius. Investigating mcts modifications in general video game playing. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 107–113. IEEE, 2015.

[Jacobsen *et al.*, 2014] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. Monte mario: platforming with mcts. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 293–300. ACM, 2014.

[Justesen *et al.*, 2016] Niels Justesen, Tobias Mahlmann, and Julian Togelius. Online evolution for multi-action adversarial games. In *Applications of Evolutionary Computation*, pages 590–603. Springer, 2016.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[Levine *et al.*, 2013] John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. *Dagstuhl Follow-Ups*, 6, 2013.

[McGuinness, 2016] Cameron McGuinness. *Monte Carlo Tree Search: Analysis and Applications*. PhD thesis, 2016.

[Park and Kim, 2015] Hyunsoo Park and Kyung-Joong Kim. Mcts with influence map for general video game playing. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 534–535. IEEE, 2015.

[Perez *et al.*, 2013] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 351–358. ACM, 2013.

[Perez *et al.*, 2015] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon Lucas, Adrien Couëtoux, Jeyull Lee, Chong-U Lim, and Tommy Thompson. The 2014 General Video Game Playing Competition. 2015.

[Perez-Liebana *et al.*, 2016] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. General Video Game AI: Competition, Challenges and Opportunities. 2016.

[Pettit and Helmbold, 2012] James Pettit and David Helmbold. Evolutionary Learning of Policies for MCTS Simulations. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 212–219. ACM, 2012.

[Poli *et al.*, 2008] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[Whitley, 1994] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.