Evolving game-specific UCB alternatives for General Video Game Playing

Ivan Bravi[1], Ahmed Khalifa[2], Cllnchristoffer Holmgård[2], Julian Togelius[2]
ivan.bravi@gmail.com, ahmed.khalifa@nyu.edu, holmgard@nyu.edu, julian@togelius.com

[1] Dipartimento di Elettronica, Informatica e Bioingegneria, Politecnico di Milano
[2] New York University, Tandon School of Engineering

Abstract. At the core of the most popular version of the Monte Carlo Tree Search (MCTS) algorithm is the UCB1 (Upper Confidence Bound) equation. This equation decides which node to explore next, and therefore shapes the behavior of the search process. If the UCB1 equation is replaced with another equation, the behavior of the MCTS algorithm changes, which might increase its performance on certain problems (and decrease it on others). In this paper, we use genetic programming to evolve replacements to the UCB1 equation targeted at playing individual games in the General Video Game AI (GVGAI) Framework. Each equation is evolved to maximize playing strength in a single game, but is then also tested on all other games in our test set. For every game included in the experiments, we found a UCB replacement that performs significantly better than standard UCB1. Additionally, evolved UCB replacements also tend to improve performance in some GVGAI games for which they are not evolved, showing that improvements generalize across games to clusters of games with similar game mechanics or algorithmic performance. Such an evolved portfolio of UCB variations could be useful for a hyper-heuristic game-playing agent, allowing it to select the most appropriate heuristics for particular games or problems in general.

Keywords: General AI, Genetic Programming, Monte-Carlo Tree Search

1 Introduction

Monte Carlo Tree Search (MCTS) is a relatively new and very popular stochastic tree search algorithm, which has been used with great success to solve a large number of single-agent and adversarial planning problems [5]. Unlike most tree search algorithms, MCTS builds unbalanced trees; it spends more time exploring those branches which seem most promising. To do this, the algorithm must balance exploitation and exploration when deciding which node to expand next.

In the canonical formulation of MCTS, the UCB1 equation is used to select which node to expand [1]. It does this by trying to maximize expected reward while also making sure that nodes are not underexplored, so that promising paths are not missed.

While MCTS is a general-purpose algorithm, in practice there are modifications to the algorithm that make it perform better on various problems. In the decade since MCTS was invented (in the context of Computer Go [22]), numerous modifications have

been proposed to allow it to play better in games as different as Chess [3] and Super Mario Bros [10], and for tasks as different as real-value optimization and real-world planning. While some of these modifications concern relatively peripheral aspects of the algorithm, others change or replace the UCB1 equation at the heart of it.

The large number of different MCTS modifications that have been shown to improve performance on different problems poses the question whether we can automate the search for modifications suitable for particular problems. If we could do that, we could drastically simplify the effort of adapting MCTS to work in a new domain. It also poses the question whether we can find modifications that improve performance compared to the existing UCB1 not just on a single problem, but on a larger class of problems. If we can identify the class of problems on which a particular MCTS version works better, we can then use algorithm selection [21, 12] or hyper-heuristics [6] to select the best MCTS version for a particular problem. And regardless of practical improvements, searching the space of node selection equations helps us understand the MCTS algorithm by characterizing the space of viable modifications.

In this paper, we describe a number of experiments in generating replacements for the UCB1 equation using genetic programming. We use the General Video Game AI (GVGAI) framework as a testbed. We first evolve UCB replacements with the target being performance on individual games, and then we investigate the performance of the evolved equations on all games within the framework. We evolve equations under three different conditions: (1) only given access to the same information as UCB1 (UCB_{+}); (2) given access to additional game-independent information (UCB_{++}); and (3) given access to game-specific information ($UCB_{\#}$).

2 Background

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 [1, 5, 15].

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

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 encounter.

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 [19].

Of particular interest for the current investigation is Cazenave's work on evolving UCB1 alternatives for Go [7]. It was found that it was possible to evolve heuristics that significantly outperformed the standard UCB1 formulation; 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.

MCTS can be used for many of the same tasks as evolutionary algorithms, such as content generation [4, 5] and continuous optimization [14]. Evolutionary algorithms have also been used for real-time planning in single-player [16] and two-player games [11].

2.3 General Video Game Playing

The problem of General Video Game Playing (GVGP) [13] is 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 GVGP competition [17, 18]. In the 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 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 [9]. This points

to the need for finding the right modification for each individual game, manually or automatically.

Fig. 1. Each screen represent a different GVGAI game. Games in order from top left to bottom right: Zelda, Butterflies, Boulderdash, and Solarfox

We selected five different games from the framework as testbeds for our experiments:

- **Boulderdash:** is a VGDL (Video Game Description Language) port of Boulderdash. The player's goal is to collect at least ten diamonds then reach the goal while not getting killed by enemies or boulders.
- **Butterflies:** is an arcade game developed specifically for the framework. The player's goal is to collect all the butterflies before they destroy all the flowers.
- **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.
- **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 has to move continuously which makes the game harder.
- **Zelda:** is a VGDL port of The legend of Zelda dungeon system. The goal is to reach the exit without getting killed by enemies. The player can kill enemies using his sword.

Figure 1 shows some levels of these games. These games require very different strategies from agents for successful play providing varied testbeds for the approach. They also have in common that standard MCTS with the UCB1 equation does not play these games

perfectly (or even very well) and in the past it has been shown that other agents play them better than MCTS.

2.4 Genetic Programming

Genetic Programming (GP) [20] is a branch of evolutionary algorithms [2, 8] which evolves computer programs as a solution to the current problem. GP is essentially the application of genetic algorithms (GA) [23] 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). In GP, chromosomes are most commonly represented as syntax trees where inner nodes are functions (e.g. addition, subtraction, conditions) while leaf nodes are terminals (e.g. constants, variables). Fitness is calculated by running the current program and seeing 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, and mutation in GP alters a node value.

3 Methods

In the GP algorithm, chromosomes are represented as syntax trees where nodes are either *unary* or *binary functions* while leaves are either *constant values* or *variables*. The binary functions available are addition, subtraction, multiplication, division and power; instead the unary functions are square root, absolute value, multiplicative inverse and natural logarithm. The constant values can be 0.1, 0.25, 0.5, 1, 2, 5, 10, 30 and their opposites. The formula can be composed of variables belonging to three different sets:

- *Tree Variables*: represent the state of the tree built by MCTS, i.e. the variables that are used by the UCB1 formula;
- *Agent Variables*: related to the agent behavior;
- *Game Variables*: describe the state of the game;

The fitness function is a linear combination of two parameters, *win ratio* and *average score*, coming from the simulation of multiple playthroughs of one level of a single game. Equation 2 shows how these two parameters are combined. We used the same formula used in the GVGAI competition to rank agents, with *win ratio* having higher priority than *average score*.

$$Fitness = 1000 * win_ratio + avg_score \tag{2}$$

In the process of evolving a game-specific equation each chromosome is tested over 100 playthroughs. Rank-based selection creates the pool of chromosomes used to generate the next generation. Two chromosomes are selected and subtree crossover is applied to the couple, later a mutation operator is applied to each new chromosome. The mutation can be either a constant mutation, point mutation, or a subtree mutation. Constant mutation selects a constant, if present, from the tree and changes it with new

value between the ones available in the *constants values* set. Point mutation consists in selecting a node with a 0.05 probability and swapping it with a node of the same type (either unary node, binary node, variable or constant). Subtree mutation picks a subtree and substitutes it with another tree of depth randomly distributed between 1 and 3.

In the population of the first generation contains 1 $UCB1$ chromosome and 99 random ones. $UCB1$ is injected in the initial population in order to push the GP to possibly converge faster to a better equation. We run the GP for 30 generations. Between one generation and the next a 10% elitism is applied to guarantee that the best chromosomes are carried out to the next generation.

The performance of the best equations, between all the chromosomes evolved by the genetic algorithm, is validated through the simulation of 2000 playthroughs.

For Boulderdash, Butterflies, Missile Command, Solar Fox, and Zelda we evolved new formulae under three different conditions:

- UCB_{+}, using only Tree Variables;
- UCB_{++}, using both Tree and Agent Variables;
- $UCB_{\#}$, using Tree, Agent and Game Variables;

Game	Tree Policy	Mean	Median	Min	Max	SD	Win ratio
Boulderdash	$UCB1$	5.30	4.00	0	186	5.22	0
Boulderdash	UCB_{+}	5.05	4.00	0	18	2.85	0
Boulderdash	UCB_{++}	**28.48**	3.0	-1.0	36.0	23.92	**0.018**
Boulderdash	$UCB_{\#}$	**27.03**	3.0	-1.0	36.0	21.85	**0.014**
Butterflies	$UCB1$	37.39	32.00	8	86	18.92	0.902
Butterflies	UCB_{+}	36.34	30.00	8	88	18.68	0.89
Butterflies	UCB_{++}	35.84	30.00	8	80	18.43	**0.914**
Butterflies	$UCB_{\#}$	22.302	18.0	12.0	48.0	8.13	**0.993**
Missile Command	$UCB1$	2.88	2.00	2	8	1.37	0.641
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**
Missile Command	$UCB_{\#}$	**8.0**	8.0	8.0	8.0	0.0	**1.0**
Solarfox	$UCB1$	6.31	5.00	0	32	6.06	0.00565
Solarfox	UCB_{+}	**6.49**	5.00	0	32	5.81	**0.0075**
Solarfox	UCB_{++}	**7.765**	6.0	-7.0	32.0	9.152	**0.067**
Solarfox	$UCB_{\#}$	**18.57**	18.0	-5.0	32.0	12.318	**0.412**
Zelda	$UCB1$	3.58	4.00	0	8	1.85	0.088
Zelda	UCB_{+}	**6.32**	6.00	0	8	1.26	**0.155**
Zelda	UCB_{++}	**6.906**	8.0	-1.0	8.0	1.623	**0.633**
Zelda	$UCB_{\#}$	**6.661**	7.0	-1.0	8.0	1.731	**0.613**

Table 1. Descriptive statistics for all the tested games. Mean, Median, Min, Max, and SD all relate to the score attained using $UCB1$, UCB_{+}, UCB_{++}, and $UCB_{\#}$, respectively. A **bold** value is significantly better than $UCB1$ ($p < 0.05$).

4 Results

Table 1 shows the results of all the evolved equations compared to $UCB1$. We can see that $UCB_{\#}$ is almost always better than all the others equations, followed by UCB_{++}, then UCB_{+}, finally $UCB1$. This was expected as the later equations have more information about the current game than the previous. Interestingly, almost all evolved equations can be said to implement the core idea behind UCB1 equation 1, in that they consist of two parts: exploitation and exploration.

4.1 Evolved Equations

In this section we will discuss and try to interpret every equation evolved for each game. We describe the best formula found in each experimental condition (UCB_{+}, UCB_{++}, and $UCB_{\#}$) for each game. All variables and their meanings can be found in Table 2.

Boulderdash

$$UCB_{+} = max(X_j)(max(X_j)^9 d_j + 1) - 0.25 \tag{3}$$

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

$$UCB_{++} = max(X_j) + d_j + \frac{1}{E_{xy}} + 1.25 \tag{4}$$

Equation 4 consists of two exploitation terms and one exploration term. The exploitation term tries to focus on the deepest explored node with the highest value, while the exploration pushes MCTS to explored nodes that are least visited in the game space.

$$UCB_{\#} = max(X_j) + \frac{N_{port}}{min(D_{port}) \cdot \sqrt{E_{xy}}} \tag{5}$$

Equation 5 consists of two main parts. The first part makes the agent more courageous and it seeks actions that increase his maximum reward regardless of anything else. The second part pushes the agent toward exploring the map while staying close to the exit door. This equation reflects the goal of the game, we can say the first part makes the player collect gems, while the second part pushes the agent towards the exit.

Butterflies

$$UCB_{+} = \overline{X_j} + \frac{1}{n_j^2 \cdot \sqrt{d_j}(d_j \cdot \sqrt{0.2 \cdot d_j \cdot max(X_j)} + 1)} \tag{6}$$

Equation 6 is similar to MCTS with exploitation and exploration terms. The exploitation term is similar to UCB1 while the 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 term is very small compared to the exploitation one so it will make a difference only between similar valued nodes.

$$UCB_{++} = \sqrt{max(X_j)} + 2\overline{X_j} - \frac{X_j}{R_j} - \left(\frac{\ln X_j}{\sqrt{max(E_{xy}) + X_j^{-0.25}}} + \sqrt{U_j}\right)^{max(E_{xy})} \quad (7)$$

Equation 7 is similar to MCTS with the mixmax modification [9]. The first two terms resemble 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 highest number of reverse moves. The useless moves force the agent to go deeper in branches that have more moves, while the number of reverse moves in *butterflies* forces the agent to move similarly to the butterflies in the game which leads to capture more of them.

$$UCB_{\#} = \overline{X_j} + \frac{1}{\Sigma D_{npc}} \quad (8)$$

Equation 8 consists of two main parts. The first part is similar to the exploitation term in MCTS. The second part makes the agent get closer to the butterflies. This equation reflects the goal of the game, we can say the first term makes the agent collect the butterflies, while the second term makes the agent get closer to them.

MissileCommand

$$UCB_{+} = \overline{X_j} + \left(10 + \frac{X_j^{X_j}}{n}\right)^{-1/\ln n} \quad (9)$$

Equation 9 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 it's relevant only when two nodes have nearly identical values.

$$\begin{aligned} UCB_{++} = \overline{X_j} + \frac{max(X_j)}{n \cdot E_{xy} \cdot (2X_j)^{0.2n_j}} \cdot \\ \left(d_j - \frac{1}{2/max(X_j) + 2U_j/X_j + 2\ln X_j + 1/n}\right)^{-1} \end{aligned} \quad (10)$$

Equation 10 has the same exploitation term from $UCB1$. Even though 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.

$$UCB_{\#} = \overline{X_j} + \frac{1}{min(D_{npc})} \quad (11)$$

Equation 11 is similar to Butterflies equation 8 with a very small difference. The difference is that the second term instead of summing all the distances, it just goes

toward the nearest missile. One of the reasons is that the goals of both games are similar: the agent wants to get closer to the moving entity (either its a butterfly or a missile). In Butterflies, the agent wins only if it collects (destroys) all butterflies while in MissileCommand, the agent wins if it destroys at least one missile. This is the reason for having a more loose equation for MissileCommand than Butterflies.

Solarfox

$$UCB_{+} = \overline{X_j} + \frac{\sqrt{d_j}}{n_j} \quad (12)$$

Equation 12 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.

$$UCB_{++} = \overline{X_j} + 2 \cdot \frac{E_{xy}}{X_j} \quad (13)$$

Equation 13 consists of two parts. The first part is similar to the exploitation term in the $UCB1$ equation 1. The second part pushes the agent towards the most visited position with the least total reward.

$$UCB_{\#} = 2 \cdot \overline{X_j} + \frac{1}{min(D_{mov})} \quad (14)$$

Equation 14 is similar to the MissileCommand equation 11 but with a different target. It focuses more on the exploitation term (increase the average score) than on getting closer to movable objects. The reason for this is that in Solarfox, the movable objects can be gems, enemies, and missiles. The agent is not able to differentiate between these three movable objects.

Zelda

$$UCB_{+} = (n + max(X_j))^{(n+max(X_j))} \quad (15)$$

Equation 15 is pure exploitation. This equation selects the node with maximum value. This new equation leads the player to be more courageous which then leads to a higher win rate and a higher score than $UCB1$.

$$UCB_{++} = 0.1 \cdot max(X_j) \cdot (\frac{1}{\sqrt{n_j}} - R_j - \sqrt{0.5 \cdot U_j} + \frac{n_j^{0.25} \cdot (2 \cdot d_j + 1)}{30 + E_{xy} - max(E_{xy})}) \quad (16)$$

Equation 16 is a very complex equation but it consists of two parts. The first one is exploitation using max value which forces the agent to be more courageous. The second part consists of multiple parts which are all related to exploration. These parts push the agent to explore the least visited node with the least repetition of the same action, the least useless actions, and the least visited position on the map. The main reason of having a very complex equation is Zelda being a hard game with multiple objectives: navigating the level, avoiding or killing enemies, getting the key then reach goal.

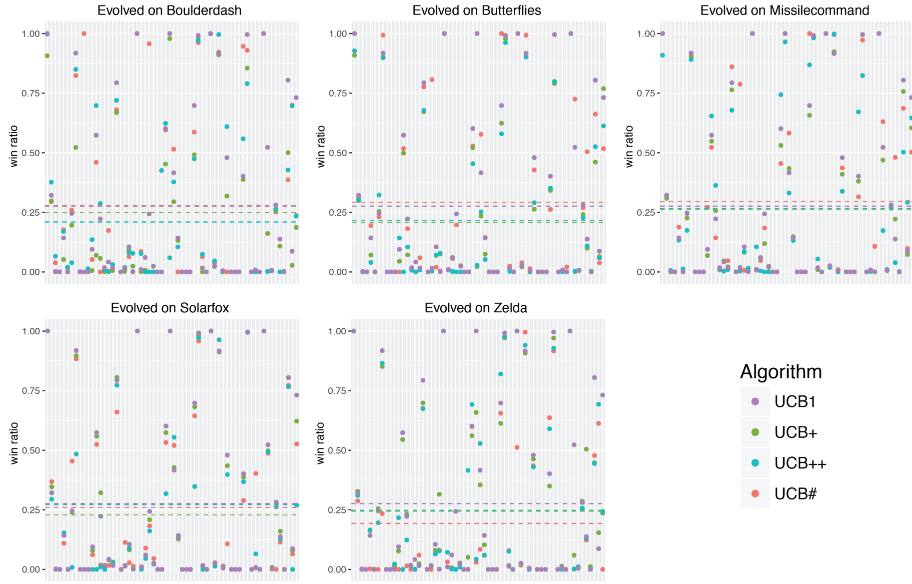

Fig. 2. Each graph shows the win ratio in all the games of the equations evolved and UCB1. Each column in a graph is a game and each dot in the column represents a win ratio. The dotted lines show the average win ratio across all the games.

$$UCB_{\#} = \sqrt{max(X_j - 0.25)} + (\Sigma D_{res})^{0.5 \cdot max(D_{mov})} + (max(X_j) \cdot \Sigma D_{immov})^{-0.5 \cdot E_{xy}} + 0.1 \cdot n \tag{17}$$

Equation 17 consists of three main parts as $0.1 \cdot n$ is equal for all children of the same node. The first part is the exploitation term, while the others are very complex terms for exploration and evading the enemies. The reason of having a very complex equation is the same for equation 16.

4.2 Testing evolved equations on all other games

We ran all evolved equations on all 62 public games in GVGAI framework to see if any of these equations can be generalized. All the results are shown in Figure 2. The average win ratio of the evolved equations is generally rather similar to $UCB1$ *on average*. In particular the two $UCB_{\#}$ equations evolved for Butterflies and Missile Command are better than $UCB1$ over all games, in the latter case with a gain of 0.02 (2%) for the win ratio. The only $UCB_{\#}$ that performs poorly compared to the others is the one evolved for Zelda. The probable reason is that the evolved formula for Zelda, a complex game, overfit to this game (as evidenced by the complexity and length of the formula).

For over 20 games, excluded the ones used in the evolutionary process, one of the evolved equation could score an improvement in the win ratio of 0.05 (5%). In particular

Variables		
d_j	Tree	child depth
n	Tree	parent visits
n_j	Tree	child visits
X_j	Tree	child value
U_j	Agent	useless moves for this child
E_{xy}	Agent	number of visits of the current tile
R_j	Agent	repeated actions as the current
RV_j	Agent	opposite action count to the current
D_{mov}	Game	distance from movable object
D_{immov}	Game	distance from immovable object
D_{npc}	Game	distance from NPC
D_{port}	Game	distance from portal
N_{port}	Game	number of portals

Table 2. Variables used in the formula and their meanings, as well as what type of variable it is (tree variable, agent variable or game variable).

in some cases the gain is more than 0.6 (60%), resulting in the algorithm mastering the game. For example, $UCB_{\#}$ evolved for Boulderdash in the games Frogs, Camel Race and Zelda is remarkable, as it takes MCTS to win rates of 96%, 100% and 70% respectively. This happens because the evolved formula embeds important information about the game: the benefit of decreasing the distance from portals.

Another great example is $UCB_{\#}$ evolved for Butterflies, which encapsulates the importance of staying close to NPCs (either to collect resources or to kill enemies) in the games of DigDug (80%), Missile Command (100%), Survive Zombies (72%) and Wait For Breakfast (50%).

5 Discussion and Conclusion

In this paper we have presented a system to evolve heuristics, used in the node selection phase of MCTS, for specific games. The evolutionary process implements a Genetic Programming technique that promotes chromosomes with the highest win rate. The goal is to examine the possibility of systematically finding UCB alternatives.

Our data supports the hypothesis that it is possible to find significantly enhance heuristics. Moreover, we can argue that embedding knowledge about the game ($UCB_{\#}$) and about the agent's behavior (both UCB_{++} and $UCB_{\#}$) allows to get exceptional improvements. With either UCB_{++} or $UCB_{\#}$ we were able to beat $UCB1$ in all the five games used in our experiments; UCB_{+} was able to beat $UCB1$ for three games, while using the same information. This supports the idea of developing a portfolio of equations that can conveniently be selected by an hyper-heuristic agent or through algorithm selection to achieve higher performance.

Many of the UCB alternatives evolved still resemble the exploitation/exploration structure of the original UCB. While the exploitation term is still the same, although sporadically swapped or enhanced by the mixmax modification, the equations instead

push the concept of exploration toward different meanings, such as spatial exploration and game-element hunt, embodying in the equation some general knowledge of the domain of games. One might even use the evolved formula to better understand the games.

Subsequently we tested the evolved equations across all games currently available in the GVG-AI framework. We could notice an overall slight improvement for $UCB_{\#}$ evolved for Missile Command and Butterflies. We also noted how a single clean equation can behave very well on games that share some design aspect. This encourages us to evolve heuristics not just for a single game but for clusters of games.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine learning 47(2-3), 235–256 (2002)
2. Bäck, T., Schwefel, H.P.: An overview of evolutionary algorithms for parameter optimization. Evolutionary computation 1(1), 1–23 (1993)
3. Baier, H., Winands, M.H.: Monte-carlo tree search and minimax hybrids. In: Computational Intelligence in Games (CIG), 2013 IEEE Conference on. pp. 1–8. IEEE (2013)
4. Browne, C.: Towards mcts for creative domains,". In: Proc. Int. Conf. Comput. Creat., Mexico City, Mexico. pp. 96–101 (2011)
5. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on 4(1), 1–43 (2012)
6. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Qu, R.: Hyper-heuristics: A survey of the state of the art. Journal of the Operational Research Society 64(12), 1695–1724 (2013)
7. Cazenave, T.: Evolving monte carlo tree search algorithms. Dept. Inf., Univ. Paris 8 (2007)
8. Eiben, A.E., Smith, J.E.: Introduction to evolutionary computing, vol. 53. Springer (2003)
9. Frydenberg, F., Andersen, K.R., Risi, S., Togelius, J.: Investigating mcts modifications in general video game playing. In: Computational Intelligence and Games (CIG), 2015 IEEE Conference on. pp. 107–113. IEEE (2015)
10. Jacobsen, E.J., Greve, R., Togelius, J.: Monte mario: platforming with mcts. In: Proceedings of the 2014 conference on Genetic and evolutionary computation. pp. 293–300. ACM (2014)
11. Justesen, N., Mahlmann, T., Togelius, J.: Online evolution for multi-action adversarial games. In: Applications of Evolutionary Computation, pp. 590–603. Springer (2016)
12. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. AI Magazine 35(3), 48–60 (2014)
13. Levine, J., Congdon, C.B., Ebner, M., Kendall, G., Lucas, S.M., Miikkulainen, R., Schaul, T., Thompson, T.: General video game playing. Dagstuhl Follow-Ups 6 (2013)
14. McGuinness, C.: Monte Carlo Tree Search: Analysis and Applications. Ph.D. thesis (2016)
15. Park, H., Kim, K.J.: Mcts with influence map for general video game playing. In: Computational Intelligence and Games (CIG), 2015 IEEE Conference on. pp. 534–535. IEEE (2015)
16. Perez, D., Samothrakis, S., Lucas, S., Rohlfshagen, P.: 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. pp. 351–358. ACM (2013)
17. Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S., Couëtoux, A., Lee, J., Lim, C.U., Thompson, T.: The 2014 General Video Game Playing Competition (2015)

18. Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S.M.: General Video Game AI: Competition, Challenges and Opportunities (2016)
19. Pettit, J., Helmbold, D.: Evolutionary Learning of Policies for MCTS Simulations. In: Proceedings of the International Conference on the Foundations of Digital Games. pp. 212–219. ACM (2012)
20. Poli, R., Langdon, W.B., McPhee, N.F., Koza, J.R.: A field guide to genetic programming. Lulu. com (2008)
21. Rice, J.R.: The algorithm selection problem. Advances in computers 15, 65–118 (1976)
22. Rimmel, A., Teytaud, O., Lee, C.S., Yen, S.J., Wang, M.H., Tsai, S.R.: Current frontiers in computer go. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 229–238 (2010)
23. Whitley, D.: A genetic algorithm tutorial. Statistics and computing 4(2), 65–85 (1994)