Online Evolution for Multi-Action Adversarial Games

Niels Justesen[1], Tobias Mahlmann[2], and Julian Togelius[3]

[1] IT University of Copenhagen `njustesen@gmail.com`
[2] Lund University `tobias.mahlmann@lucs.lu.se`
[3] New York University `julian@togelius.com`

Abstract. We present *Online Evolution*, a novel method for playing turn-based multi-action adversarial games. Such games, which include most strategy games, have extremely high branching factors due to each turn having multiple actions. In Online Evolution, an evolutionary algorithm is used to evolve the combination of atomic actions that make up a single move, with a state evaluation function used for fitness. We implement Online Evolution for the turn-based multi-action game *Hero Academy* and compare it with a standard Monte Carlo Tree Search implementation as well as two types of greedy algorithms. Online Evolution is shown to outperform these methods by a large margin. This shows that evolutionary planning on the level of a single move can be very effective for this sort of problems.

1 Introduction

Game-playing can fruitfully be seen as search: the search in the space of game states for desirable states which are reachable from the present state. Thus, many successful game-playing programs rely on a search algorithm together with a heuristic function that scores the desirability (usually related to the probability of winning given that state). In particular many adversarial two-player games with low branching factors, such as Checkers and Chess, can be played very well by the Minimax algorithm [15] together with a state evaluation function. Other games have higher branching factors, which greatly reduces the efficacy of Minimax search, or make the development of informative heuristic functions very hard as many game states are deceptive. A classic example is *Go*, where computer players for a long time performed poorly. For such games, Monte Carlo Tree Search (MCTS) [4] tends to work much better; MCTS handles higher branching factors well by building an unbalanced tree, and performs state estimations by Monte Carlo simulations until the end of the game. The advent of the MCTS algorithm caused a qualitative improvement in the performance of Go-playing programs [2].

Many games, including all one-player games and many one-and-a-half-player games (where the player character faces non-player characters), are not adversarial [6]. These include many puzzles and video games. For such games, the game-playing problem is similar to a classic planning problem, and methods based on best-first search become applicable and in many cases effective. For example, a version of A* plays *Super Mario Bros* very well given reasonably linear levels [20]. But MCTS is also useful for many non-adversarial games, in particular with high branching factors, hidden information and/or non-deterministic outcomes.

First-Play Urgency (FPU) is one of many enhancements to MCTS for games with large branching factor [7]. FPU encourages early exploitation by assigning a fixed score to unvisited nodes. Rapid Action Value Estimation (RAVE) is another popular enhancement that has been shown to improve MCTS in Go [9]. Script-based approaches such as Portfolio Greedy Search [5] and Script-based UCT [11] deals with the large branching factor of real-time strategy games by exploring a search space of scripted behaviors instead of actions.

Recently, a method for playing non-adversarial games called *rolling horizon evolution* was introduced [17]. The basic idea is to use an evolutionary algorithm to evolve a sequence of actions to perform and during the execution of these actions a new action sequence is evolved. This process is continued until the game is over. This use of evolution differs sharply from how evolutionary algorithms are commonly used in game-playing and robotics, to evolve a controller that later selects actions [21, 3, 22]. The fitness function is the desirability of the final state in the sequence, as estimated by either a heuristic function or Monte Carlo playouts. This approach was shown to perform well on both the Physical Travelling Salesman Problem [16] and many games in the General Video Game Playing benchmark [13]. However, rolling horizon evolution cannot be straightforwardly applied to adversarial games, as it does not take the opponent's actions into account; in a sense, it only considers the best case.

In this paper, we consider a class of problems which has been relatively less studied, and for which none of the above described methods perform well. This is the problem of multi-action turn-based adversarial games, where each player each turn takes multiple separate actions, for example by moving multiple units or pieces. Games in this class include strategy games played either on tabletops or using computers, such as *Civilization*, *Warhammer 40k* or *Total War*; the class includes games more similar to classic board games, such as *Arimaa*, and arguably many real-world problems involving the coordinated action of multiple units. The problem with this class of games is the branching factor. Whereas the average branching factor hovers around 30 for Chess and 300 for Go, a game where you move six units every turn and each unit can do one out of ten actions has a branching factor of a million. Of course, neither MiniMax nor MCTS work very well with such a number; the trees become very shallow. The way such games are often played in practice is by making strongly simplifying. For example, if you assume independence between units your branching factor is only 60, but this assumption is typically wrong.

Rolling horizon evolution does not work on the class of games we consider either for the reason that they are adversarial. However, evolution can still be useful here, in the context of selecting *which actions to take during a single move*. The key observation here is that we are only looking to know which turn to take next, but finding the right combination of actions to compose that turn is a formidable search problem in itself. The method we propose here, which we call *online evolution*, evolves the actions in a single turn and uses an estimation of the state at the end of the turn (right before the opponent takes their turn) as a fitness function. It can be seen as a single iteration of rolling horizon evolution with a very short horizon (one turn).

In this paper, we apply online evolution to the game *Hero Academy*. It is contrasted with several other approaches, including MCTS, random search and greedy search, and shown to perform very well.

2 Methods

This section presents our testbed game, our methods for reducing the search space and evaluating game states, and search algorithms we test, including MCTS and Online Evolution.

2.1 Testbed Game: Hero Academy

Our testbed, a custom-made version[4] of *Hero Academy*[5], is a two-player turn-based tactics game inspired by chess and is very similar to the battles in the *Heroes of Might & Magic* series. Figure 1 shows a typical game state. Players have a pool of combat units and spells at their disposal to deploy and use on a grid-shaped battle field. Tactical variety is achieved by different unit classes that fulfil different combat roles (fighter, wizard, etc.) and the mechanic of "action points". Each turn, the active player starts with five action points, which can be freely distributed among units on the map, deploy new units, or cast spells. Especially noteworthy is that a player may chose to distribute more than one action point per unit, i.e. let a unit act twice or more times per turn. A turn is completed once all five action points are used. The game itself has no turn limit while our experiments did implement a limit of 100 turns per player. The first player to eliminate the enemy's units or base *crystals* wins the game. For more details on the implementation, rules, and tactics on the game, we kindly ask the reader to refer to the Master thesis referenced as [10].

The action point mechanic makes *Hero Academy* very challenging for decision making algorithms due to the number of possible future game states which is significantly higher than in other games. Many different action sequences may however, lead to the same end turn game state as units can be moved freely in any order. In the following, we present and discuss different methods in regard to this problem.

2.2 Action Pruning & Sorting

Our implemented methods used action pruning to reduce the enormous search space of a turn by removing (pruning) redundant swap actions and sub-optimal spell actions from the set of available actions in a state. Two swap actions are redundant if they swap the same kind of item and one can be removed as they produce the same outcome. A spell action is sub-optimal if another spell action covers the same or more enemy units. In this way spells that do not target any enemy units will also be pruned because it is always possible to target the opponent's crystals.

For some search methods, it makes sense to investigate the most promising moves first and thus a method for sorting actions is needed. A simple way would be to evaluate

[4] `https://github.com/njustesen/hero-aicademy`
[5] `http://www.robotentertainment.com/games/heroacademy/`

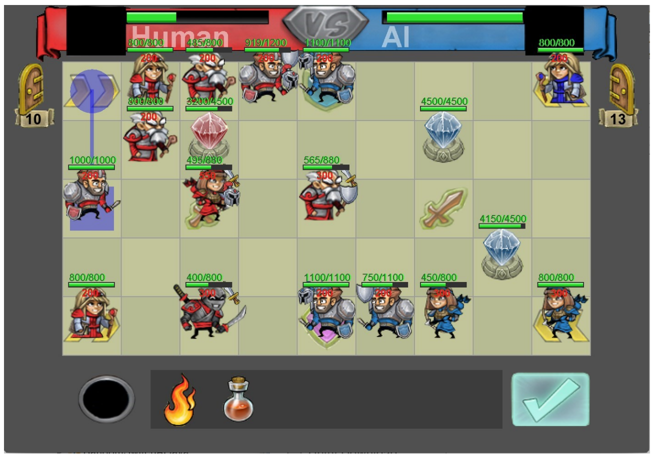

Fig. 1: A typical game state in Hero Academy. The screenshot is from our own implementation of the game.

the resulting game state of each action, but this is usually a slow method. The method we implemented rates an action by how much damage it deals or how much health it heals. If an enemy unit is removed from the game, it is given a large bonus. In the same way, healing actions are awarded a bonus if they are saving a knocked out unit. In this way, critical attack and healing actions are rated high and movement actions are rated low.

2.3 State Evaluation

Several of our algorithms require an evaluation of how "good" a certain state for a player is. For this case, we used a heuristic to evaluate the board in a given state. This heuristic is based on the difference between the values of both players' units, assuming it as the main indicator for which player is winning. This includes the units on the game board and those which are still at the players' disposal. Furthermore, the value of a unit u is calculated using a linear combination as follows:

$$\begin{aligned} v(u) = u_{hp} + \underbrace{u_{maxhp} \times up(u)}_{\text{standing bonus}} + \overbrace{eq(u) \times up(u)}^{\text{equipment bonus}} \\ + \underbrace{sq(u) \times (up(u) - 1)}_{\text{square bonus}} \end{aligned} \tag{1}$$

whereas u_{hp} is the number of health points u has, $sq(u)$ adds a bonus based on the type of square u stands on, and $eq(u)$ adds a bonus based on the unit's equipment. For brevity, we will not discuss these in detail, but instead list the exact modifiers in Table 1.

Lastly, the modifying term $up(u)$ is defined as:

$$up(u) = \begin{cases} 0, & \text{if } u_{hp} = 0 \\ 2, & \text{otherwise} \end{cases} \tag{2}$$

This will make standing units more valuable than knocked out units.

	Dragonscale	Runemetal	Helmet	Scroll
Archer	30	40	20	50
Cleric	30	20	20	30
Knight	30	-50	20	-40
Ninja	30	20	10	40
Wizard	20	40	20	50

(a) Bonus added to units with items.

	Assault	Deploy	Defence	Power
Archer	40	-75	80	120
Cleric	10	-75	20	40
Knight	120	-75	30	30
Ninja	50	-75	60	70
Wizard	40	-75	70	100

(b) Bonus added to units with items.

Table 1: For completeness, we list the modifiers used by our game state evaluation heuristic.

2.4 Tree Search

Game-tree based methods have gained much popularity and have been applied with success to a variety of games. In short, a game tree is a acyclic directed graph with one source node (the current game state is the root) and several leaf nodes. Its nodes depict hypothetical future game states and its edges define the players' actions that would lead to these states. A node has therefore as many edges leading from it, as the number of actions available for the *active player* in that game state. Additionally, each edge is assigned a value, and the edge leading from the actual gamestate (the root node of the tree) with the highest value is considered the best current move. In adversarial games, it is common that players take turns and hence the active player alternates between plies of the tree. The well-known Minimax algorithm makes use of this. However, in Hero Academy players take several actions before their turn ends. One possibility would be to encode multiple actions as one multi-action, e.g. as an array of actions, and assign it to one edge. Due to the number of possible permutations, this would raise the number of child nodes for a given game state immensely. Therefore, we decided to model each action as its own node, trading tree breadth for depth.

As the number of possible actions is variable, depending on the current game state, determining the exact branching factor is hardly possible. To get an estimate, we manually counted the number of possible actions in a recorded game to be 60 on average. We therefore estimate the average branching factor per turn to be $60^5 = 7.78 \times 10^8$ as each player has five actions. If we further assume through observation that the average game length is 40 turns and both players take a turn each round, we can calculate the average game-tree complexity to $((60^5)^2)^{40} = 1.82 \times 10^{711}$. As a comparison: Shannon calculated the game-tree complexity of Chess to be 10^{120} [19].

In the following, we will present three game-tree based methods, which were used as a baseline for our online evolution method.

Greedy search among actions The *Greedy Action* method is the most basic method developed. It makes a one-ply search among all possible actions, and selects the action that leads to the most promising game state based on our heuristic. It also uses action pruning described earlier. The Greedy Action search is invoked five times to complete a turn.

Greedy search among turns *Greedy Turn* performs a five-ply depth-first search corresponding to a full turn. Both action pruning and action sorting are applied at each node. The heuristic described earlier rates all states at leaf nodes and then chooses the action sequence that leads to the highest-rated state. A transposition table is used so that already visited game states will not be visited again. This method is very similar to a Minimax search that is depth-limited to only search in the first five ply. Except for some early and late game situations *Greedy Turn* is not able to make an exhaustive search of the space of actions, even with a time budget of a minute.

Monte Carlo Tree Search Monte Carlo Tree Search has successfully been implemented for games with large branching factors such as the strategy game Civilization II [1] and it thus seems to be an important algorithm to test in *Hero Academy*. Like the two greedy search variants, the Monte Carlo Tree Search algorithm was implemented with an action based approach, i.e. one ply in the tree represents an action, not a turn. Hence the search has to reach the depth of five to reach the beginning of the opponent's turn. In each exploration phase, one child is added to the node chosen in the selection phase, and a node will not be selected unless all of its siblings have been added in previous iterations. Additionally, we had to modify the standard backpropagation to handle two players with multiple actions. We solved this with an extension of the *BackupNegamax* [2] algorithm (see Algorithm 1). This backpropagation algorithm uses a list of edges corresponding to the traversal during the selection phase, a Δ value corresponding to the result of the simulation phase and a boolean $p1$ that is *true* if player one is the max player and *false* otherwise. The ϵ-greedy approach was used in the rollouts that combine random play with the highest rated action (rated by our actions sorting method). The MCTS agent was given a budget of b milliseconds. As agents in Hero Academy have to select not one but five actions, we experimented with two approaches: the first approach was to request one action from the agent five times each with a time budget of $\frac{b}{5}$. The second approach was to request five actions from the agent with a time budget of b. The second approach proved to be superior as it gives the search algorithms more flexibility.

2.5 Online Evolution

Evolutionary algorithms have been used in various ways to evolve controllers for many games. This is done by what is called *Offline Learning* where a controller first goes

Algorithm 1 Alteration of the BackupNegamax [2] algorithm for multi-action games.

```
1: procedure MultiNegamax(Edge[] $T$, Double $\Delta$, Boolean $p1$)
2:     for all Edge $e$ in $T$ do
3:         $e.visits$++
4:         if $e.to \neq null$ then
5:             $e.to.visits$ ++
6:         if $e.from = root$ then
7:             $e.from.visits$ ++
8:         if $e.p1 = p1$ then
9:             $e.value$ += $\Delta$
10:        else
11:            $e.value$ −= $\Delta$
```

through a training phase in which it learns to play the game. In this section we will present an evolutionary algorithm that, inspired by the rolling horizon evolution, evolves strategies while it plays the game. We call this algorithm *Online Evolution*. The online evolution was implemented to play *Hero Academy* and aims to evolve the best possible action sequence each turn. Each individual in a population thus represent a sequence of five actions. A brute force search, like the Greedy Turn search, is not able to explore the entire space of action sequences within a reasonable time frame and may miss many interesting choices. An evolutionary algorithm on the other hand can explore the search space in a very different way and we will show that it works very well for this game.

An overview of the online evolution algorithm will now be given and is also presented in pseudocode (see Algorithm 2). The online evolution first creates a population of random individuals. These are created by repeatedly selecting a random action in a forward model of the game until no more action points are left. In our case we were able to use the game implementation itself as a forward model.

In each generation all individuals are rated using a fitness function which is based on the hand-written heuristic described in the previous section, where after the worst individuals are removed from the population. The remaining individuals are then each paired with another random individual to breed an offspring through uniform crossover. An example of the crossover mechanism for two action sequences in Hero Academy can be seen on Figure 2. The offspring will the represent an action sequence that is a random combination of its two parents'. Crossover can however in its simplest form easily produce illegal action sequences for Hero Academy. E.g. moving a unit from a certain position obviously requires that there is a unit on that square, which might not be true due to an earlier action in the sequence. Illegal action sequences could be allowed but we believe the population would be swarmed with illegal sequences doing so. Instead actions are only selected from a parent if it is legal and otherwise the action will be selected from the other parent. If both actions are illegal it will try the same approach on the next action in the parents sequences and if they are illegal as well a completely random available action is finally selected.

Some offspring will also be mutated to introduce new actions in the gene pool. Mutation simply changes one random action to another legal action. Legal en respect to the previous actions only. In some cases this will still result in an illegal action sequence.

Algorithm 2 Online Evolution
(Procedures *Procreate* (Crossover and Mutation), *Clone* and *Eval* are omitted)

```
 1: procedure OnlineEvolution(State $s$)
 2:     Genome[] $pop$ = $\emptyset$                          ▷ Population
 3:     Init($pop$, $s$)
 4:     while time left do
 5:         for each Genome $g$ in $pop$ do
 6:             $clone$ = Clone($s$)
 7:             $clone$.update($g$.actions)
 8:             if $g$.visits = 0 then
 9:                 $g$.value = Eval($clone$)
10:             $g$.visits++
11:         $pop$.sort()                            ▷ Descending order after value
12:         $pop$ = first half of $pop$                         ▷ 50% Elitism
13:         $pop$ = Procreate($pop$)                     ▷ Mutation & Crossover
14:     return $pop$[0].actions                        ▷ Best action sequence
15:
16: procedure Init(Genome[] $pop$, State $s$)
17:     for $x$ = 1 to POP_SIZE do
18:         State $clone$ = clone($s$)
19:         Genome $g$ = new Genome()
20:         $g$.actions = RandomActions($clone$)
21:         $g$.visits = 0
22:         $pop$.add($g$)
23:
24: procedure RandomActions(State $s$)
25:     Action[] $actions$ = $\emptyset$
26:     Boolean $p1$ = $s$.p1                              ▷ Who's turn is it?
27:     while $s$ is not terminal AND $s$.p1 = $p1$ do
28:         Action $a$ = random available action in $s$
29:         $s$.update($a$)
30:         $actions$.push($a$)
31:     return $actions$
```

If this happens the following part of the sequence is changed to random but legal actions as well.

Attempts were made to use rollouts as the heuristic for the online evolution to incorporate information about possible counter moves. In this variation the fitness function is altered to perform one rollout with a depth limit of five actions i.e. one turn. The goal of introducing rollouts is to rate an action sequence by the outcome of the best possible counter-move. Individuals in the population that survive several generations will also be tested several times and in this case only the lowest found value is used. A good action sequence can thus survive many generations until a good counter-move is found. To avoid that such a solution re-enters the population the worst known value for each action sequence is stored in a table. Despite our efforts of using stochastic roll-

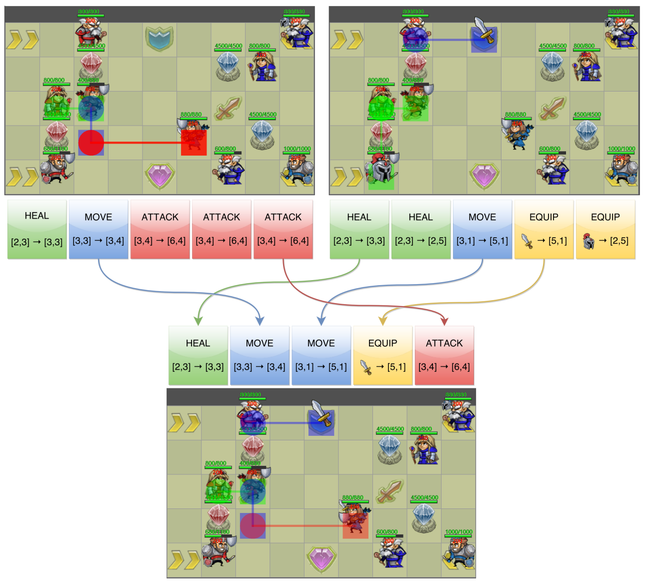

Fig. 2: An example of the uniform crossover used by the online evolution in *Hero Academy*. Two parent solutions are shown in the top and the resulting solution after crossover in the bottom. Each gene (action) are randomly picked from one of the parents. Colours on genes represent the type of action they represent. Healing actions are green, move actions are blue, attack actions are red and equip actions are yellow.

outs as a fitness function no significant improvement was observed compared to a static evaluation. The experiments of this variation are thus not included in this paper.

3 Experiments and Results

In this sections we will describe our experiments and present the results of playing each of the described methods against each other.

3.1 Experimental Setup

Experiments were made using the testbed described earlier. Each method was played against each other method 100 times, 50 times as the the starting player and 50 times as the second player. The map seen on Figure 1 was used and all methods played as the Council team. The testbed was configured to be without randomness and hidden information to focus further on the challenge of performing multiple actions. Each method was not allowed to use more than one processor and had a time budget of six seconds each turn. The winning percentages of each matchup will be presented where draws counts as half a win for each player. The rules of Hero Academy does not include

	Random	Greedy Action	Greedy Turn	MCTS	Online Evolution
Greedy Action	100%	-	36%	51.5%	10%
Greedy Turn	100%	64.0%	-	88.0%	19.5%
MCTS	100%	48.5%	22.0%	-	2%
Online Evolution	100%	90.0%	80.5%	98%	-

Table 2: Win percentages of the agents listed in the left-most column in 100 games against agents listed in the top row. Any win percentage of 62% or more is calculated to be significant with a significance level of 0.05 using the Wilcoxon Signed-Rank Test.

draws, but we enforced this when no winner was found in 100 rounds. The experiments were carried out on a Intel Core i7-3517U CPU with 4 $\times$ 1.90GHz cores and 8 GB of ram.

3.2 Configuration

The following configurations were used for our MCTS implementation. The traditional UCT tree policy $\overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$ was used with the exploration constant $C_p = \frac{1}{\sqrt{2}}$. The default policy is ϵ-greedy, where ϵ=0.5. Rollouts were depth-limited to one turn, using the heuristic state evaluator described above. Action pruning and sorting are used as described above. A transposition table was used with the descent-path only back-propagation strategy and thus values and visit counts are stored in edges. n_j in the tree policy is thus in fact extracted from the child edges instead of the nodes.

Our experiments clearly show that short rollouts are preferred over long rollouts and that rollouts of just one turn gives the best results. Also by adding some domain knowledge to the rollouts with the ϵ-greedy policy the performance is improved. ϵ-greedy picks a greedy action equivalent to the highest rated action by the action sorting method with a probability of ϵ and otherwise a random action is picked.

Online evolution used a population size of 100, survival rate 0.5, mutation probability 0.1 and uniform crossover. The heuristic state evaluator described earlier is also used by the online evolution.

3.3 Performance Comparison

Our results, shown in Table 2, show a clear performance ordering between the methods. Online evolution was the best performing method with a minimum winning percentage of 80.5% against the best of the other methods. *GreedyTurn* performs second best. In third place, MCTS plays on the same level as *GreedyAction*, which indicates that it is able to identify the action that gives the best immediate reward while it is unable to search sufficiently through the space of possible action sequences. All methods convincingly beat random search.

3.4 Search Characteristic Comparison

To further understand how the methods explores the search space, let us investigate some of the statistics gathered during the experiments, in particular the number of different action sequences each method is able to evaluate within the given time budget. Since many action sequences produce the same outcome, we have recorded the number of unique outcomes evaluated by each method. The *GreedyTurn* search was on average able to evaluate 579,912 unique outcomes during a turn. Online Evolution evaluated on average 9,344 unique outcomes, and MCTS only 201. Each node at the fifth ply of the MCTS tree corresponds to one unique outcome and the search only manages to expand the tree to a limited number of nodes at this depth. When looking into more statistics from MCTS, we can see that the average depth of leaf nodes in the final trees is 4.86 plies, while the deepest leaf node of each tree reached an average depth of 6.38 plies. This means that the search tree just barely enters the opponents' turn even though it manages to run an average of 258,488 iterations per turn. The Online Evolution ran an average of 3,693 generations each turn but seems to get stuck at a local optima very quickly as the number of unique outcomes evaluated is low. This suggests that it would play almost equally good with a much lower time budget, but also that the algorithm could be improved.

4 Discussion

The results strongly suggest that online evolution searches the space of plans more efficiently than any of the other methods. This should perhaps not be too surprising, since MCTS was never intended to deal with this type of problem, where the "turn-level branching factor" is so high that it all possible turns cannot even be enumerated during the time allocated. MCTS have also failed to work well in Arimaa which has only four actions each turn [12]. In other words, the superior performance of evolutionary computation on this problem might be due more to that very little research has been done on problems of this type. Given the similarities of Hero Academy to other strategy games, and to that these games model real-life strategic decision making, this is somewhat surprising. More research is clearly needed.

One immediately promising avenue for further research is to try using evolutionary algorithms with diversity maintenance methods (such as niching [14]), given that many strategies in the method used here seems to have been explored multiple times. Tabu-search could also be effective [8]. Exploration of a larger number of strategies is likely to lead to better performance.

Finally, it would be very interesting to try and take the opponents' move(s) into account as well. Obviously, a full Minimax search will not be possible, given that the first player's turn cannot even be explored exhaustively, but it might still be possible to explore this through competitive coevolution [18]. The idea here is that one population contains the first player's turn, and another population the second player's turn; the fitness of the second population's individuals is the inverse of that of the first population's individuals. There is a major unsolved problem here in that the outcome of the first turn decides the starting conditions for the second turn so that most individuals in the second population would be incompatible with most individuals in the first population, but it may be possible to define a repair function that addresses this.

5 Conclusion

This paper describes online evolution, a new method for playing adversarial games with very large branching factors. This is common in strategy games, and presumably in the real-world scenarios they model. The core idea is to use an evolutionary algorithm to search for the next turn, where the turn is composed of a sequence of actions. We compared this algorithm with several other algorithms on the game Hero Academy; the comparison set includes a standard version of Monte Carlo Tree Search. MCTS is the state of the art for many games with high branching factor. Our results show that online evolution convincingly outperforms all other methods on this problem. Further analysis shows that it does this despite considering fewer unique turns than the other algorithms. It should be noted that other variants of the MCTS algorithm are likely to perform better on problems of this type, just as other variants of Online Evolution might; we are not claiming that evolution outperforms all types of tree search. Future work will go into investigating how well this performance holds up in related games, and how to improve the evolutionary search. We will also compare our approach with more sophisticated versions of MCTS, as outlined in the introduction.

References

1. Branavan, S., Silver, D., Barzilay, R.: Non-linear monte-carlo search in civilization ii. AAAI Press/International Joint Conferences on Artificial Intelligence (2011)
2. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S., et al.: A survey of monte carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on 4(1), 1–43 (2012)
3. Cardamone, L., Loiacono, D., Lanzi, P.L.: Evolving competitive car controllers for racing games with neuroevolution. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation. pp. 1179–1186. ACM (2009)
4. Chaslot, G., Bakkes, S., Szita, I., Spronck, P.: Monte-carlo tree search: A new framework for game ai. In: AIIDE (2008)
5. Churchill, D., Buro, M.: Portfolio greedy search and simulation for large-scale combat in starcraft. In: Computational Intelligence in Games (CIG), 2013 IEEE Conference on. pp. 1–8. IEEE (2013)
6. Elias, G.S., Garfield, R., Gutschera, K.R.: Characteristics of games. MIT Press (2012)
7. Gelly, S., Wang, Y.: Exploration exploitation in go: Uct for monte-carlo go. In: NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop (2006)
8. Glover, F., Laguna, M.: Tabu Search*. Springer (2013)
9. Helmbold, D.P., Parker-Wood, A.: All-moves-as-first heuristics in monte-carlo go. In: IC-AI. pp. 605–610 (2009)
10. Justesen, N.: Artificial Intelligence for Hero Academy. Master's thesis, IT University of Copenhagen (2015)

11. Justesen, N., Tillman, B., Togelius, J., Risi, S.: Script-and cluster-based uct for starcraft. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 1–8. IEEE (2014)
12. Kozelek, T.: Methods of mcts and the game arimaa. Charles University, Prague, Faculty of Mathematics and Physics (2009)
13. Levine, J., Congdon, C.B., Ebner, M., Kendall, G., Lucas, S.M., Miikkulainen, R., Schaul, T., Thompson, T., Lucas, S.M., Mateas, M., et al.: General video game playing. Artificial and Computational Intelligence in Games 6, 77–83 (2013)
14. Mahfoud, S.W.: Niching methods for genetic algorithms. Urbana 51(95001), 62–94 (1995)
15. Neumann, J.v.: Zur Theorie der Gesellschaftsspiele. Mathematische Annalen 100(1), 295–320 (1928)
16. Perez, D., Rohlfshagen, P., Lucas, S.M.: Monte-carlo tree search for the physical travelling salesman problem. In: Applications of Evolutionary Computation, pp. 255–264. Springer (2012)
17. 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)
18. Rosin, C.D., Belew, R.K.: New methods for competitive coevolution. Evolutionary Computation 5(1), 1–29 (1997)
19. Shannon, C.E.: Xxii. programming a computer for playing chess. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 41(314), 256–275 (1950)
20. Togelius, J., Karakovskiy, S., Baumgarten, R.: The 2009 mario ai competition. In: Evolutionary Computation (CEC), 2010 IEEE Congress on. pp. 1–8. IEEE (2010)
21. Togelius, J., Karakovskiy, S., Koutník, J., Schmidhuber, J.: Super mario evolution. In: Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on. pp. 156–161. IEEE (2009)
22. Zhou, A., Qu, B.Y., Li, H., Zhao, S.Z., Suganthan, P.N., Zhang, Q.: Multiobjective evolutionary algorithms: A survey of the state of the art. Swarm and Evolutionary Computation 1(1), 32–49 (2011)