Playing Multiaction Adversarial Games: Online Evolutionary Planning Versus Tree Search

Niels Justesen, Tobias Mahlmann, Sebastian Risi, and Julian Togelius

***Abstract*—We address the problem of playing turn-based multiaction adversarial games, which include many strategy games with extremely high branching factors as players take multiple actions each turn. This leads to the breakdown of standard tree search methods, including Monte Carlo tree search (MCTS), as they become unable to reach a sufficient depth in the game tree. In this paper, we introduce online evolutionary planning (OEP) to address this challenge, which searches for combinations of actions to perform during a single turn guided by a fitness function that evaluates the quality of a particular state. We compare OEP to different MCTS variations that constrain the exploration to deal with the high branching factor in the turn-based multiaction game *Hero Academy*. While the constrained MCTS variations outperform the vanilla MCTS implementation by a large margin, OEP is able to search the space of plans more efficiently than any of the tested tree search methods as it has a relative advantage when the number of actions per turn increases.**

***Index Terms*—Computational complexity, evolutionary computation, Monte Carlo tree search, tree search.**

I. Introduction

ADVERSARIAL games, in which one player's loss is the other's gain, have a long tradition as testbeds in artificial intelligence. In this context, playing the game well can be viewed as a search from the current state of the game to desirable future states. In fact, many well-performing game-playing programs rely on search algorithms that are guided by some heuristic function that evaluates the desirability of a given state. For adversarial two-player games with relatively low branching factors, such as *Checkers* and *Chess*, search algorithms such as Minimax together with well-designed heuristic functions perform remarkably well [1].

However, as the branching factor increases, the efficacy of Minimax search is greatly reduced. In cases where it is hard to develop or learn informative heuristic functions, this problem is further compounded. A classic example is *Go*, where it took several decades of research to come up with algorithms that play at world-class human level [2], [3].

While Go has a branching factor of "only" 300 (a magnitude higher than the 30 for Chess), it is still far lower than most of the turn-based multiaction adversarial games, where each player takes multiple separate actions each turn, for example, by moving multiple units or pieces. In this case, none of the mentioned methods currently perform well. Examples of multiaction turn-based adversarial games include strategy games such as *Civilization*, *Warhammer 40k*, *XCOM* or *Heroes of Might and Magic* but also card and board games such as the board game *Arimaa*. This class of problems arguably also includes many real-world problems involving coordination of multiple agents.

By including multiple actions and multiple units, the branching factor quickly reaches intractable dimensions. For example, a strategy game that allows the movement of six units every turn and each unit can perform one out of ten actions has a branching factor of a million (10^6). Standard tree search methods tend to fail with such high branching factors because the trees become very shallow. To allow tree search methods to be applied in such circumstances, some authors resort to making strong assumptions to guide how to explore actions [4]. One naive assumption, which is wrong in the general case, is independence between units, which decreases the branching factor to only 60 rather than a million.

Instead of relying on a tree-based search method, Perez *et al.* introduced an evolutionary algorithm-based method for playing nonadversarial games called rolling horizon evolution (RHE) [5]. RHE evolves a sequence of game actions to perform in the near future. The agent then performs the first action in the found sequence and evolves a new sequence from scratch. This process is continued a number of times until the game is over. The RHE algorithms runs online during the game, differing sharply with the way evolution is typically applied in game playing, in which a controller is evolved that determines the actions of the agent [6]–[9]. The fitness function of an action sequence is the desirability of the final state reached by performing these actions, which is estimated by some heuristic. RHE has been successful in a number of real-time environments including the physical traveling salesman problem [10] and in the general video game playing benchmark [11], [12]. It is specifically designed for real-time games and thus cannot be directly applied to adversarial games. Evolution can, however, still be useful for these games by evolving a sequence of actions to take

Manuscript received March 24, 2017; revised June 21, 2017; accepted July 25, 2017. Date of publication August 11, 2017; date of current version September 13, 2018. *(Corresponding author: Niels Justesen.)*

N. Justesen and S. Risi are with the Center for Computer Games Research, IT University of Copenhagen, 2300 Copenhagen, Denmark (e-mail: njustesen@gmail.com; sebastian.risi@gmail.com).

T. Mahlmann is with the Department of Philosophy, Lund University, 223 62 Lund, Sweden (e-mail: tobias.mahlmann@lucs.lu.se).

J. Togelius is with the Tandon School of Engineering, New York University, New York, NY 10003 USA (e-mail: julian@togelius.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2017.2738156

during a single turn. Finding the optimal combination of actions to perform during just one turn is a formidable search problem in itself due to combinatorial complexities.

The novel online evolutionary planning (OEP) algorithm introduced in this paper searches for the actions to perform in a single turn and uses an estimation of the state at the end of the turn (i.e., right before the opponent takes their turn) as a fitness function. We compare the OEP approach to Monte Carlo tree search (MCTS) [13], which has shown to work well for games with higher branching factors. 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 [14], and MCTS has been part of almost every high-level Go-playing program since including the world champion AlphaGo [2].

The domain investigated in this paper is the multiaction adversarial turn-based game called *Hero Academy*, a competitive strategy game playable on PC and iPad. Because of the extremely high branching factor of *Hero Academy*, we also developed two variations of MCTS that attempt to limit exploration, resulting in a more focused search. The first MCTS variation has a greedy tree policy, that always selects the most valuable node, and a deterministic default policy during rollouts. The other variation prunes branches aggressively to push the search in the most promising direction.

This paper builds and expands on results previously published in conference proceedings [15]. In more detail, we added the following:

1) two new variations of MCTS that aggressively constrain the exploration to deal with the enormous action space of multiaction games;
2) an investigation of how the methods perform with varying numbers of actions per turn, which demonstrates the scalability of OEP;
3) a user study of 111 games with users of various skill levels, showing the usefulness of applying OEP to a real game product;
4) a deeper complexity analysis of the game *Hero Academy* as well as a more in-depth discussion of the results.

The new MCTS variations, bridge burning MCTS (BB-MCTS) and nonexploring MCTS, are shown to outperform vanilla MCTS in this domain. While OEP performs slightly worse than nonexploring MCTS in case of low numbers of actions per turn, it outperforms all other methods with increasing numbers of actions. Additionally, the user study suggests that OEP also performs well against human players. Only OEP was tested against humans.

This paper begins with a brief review of relevant related work. Section III describes the testbed used in our experiments, which is a game called *Hero Academy*. Section IV describes OEP, which is followed by a number of different tree search approaches including MCTS. Next, Section VII presents the experimental setup and results, and, finally, this paper concludes with a discussion (see Section VIII) and conclusion (see Section IX).

II. Related Work and Background

This section reviews relevant work on MCTS and evolutionary algorithms that runs while the agent is playing a game. There does not exist much work on multiaction adversarial games in the literature.

A. Monte Carlo Tree Search

MCTS is a best-first search that uses stochastic sampling as a heuristic [13], [16] and has been successfully applied to games with large branching factors such as *Civilization II* [17], *Magic the Gathering* [18], and *Settlers of Catan* [19]. The algorithm starts with a root node representing the current game state. Four phases are sequentially executed in iterations until a given time budget is used or a satisfying goal state is reached. In the selection phase, the tree is traversed from the root node using a tree policy until a node with unexpanded children is reached. In the expansion phase, a child node is expanded from the selected node. In the simulation phase (also called rollout), the remaining part of the game, from the expanded node's game state, is played out using a default policy. In the backpropagation phase, the outcome of the game is backpropagated up the tree until the root node is reached.

The tree policy determines how the search balances exploration and exploitation during the selection phase. Usually, the upper confidence bounds (UCB) algorithm is used [20], which selects the node that maximizes

$$\text{UCB1} = \overline{X}_j + C\sqrt{\frac{2\ln n}{n_j}}$$

where n is the visit count of the current node, n_j is the visit count of the child j, C is a constant determining the amount of exploration versus exploitation, and $\overline{X}_j$ is the normalized value of child j. The default policy is used during rollouts to select actions, which can be a complex scripted policy or one that selects random actions. An ϵ-greedy strategy can also be used to select a random action at probability ϵ and at probability $1-\epsilon$ follows some predefined policy.

MCTS has shown promise for many nonadversarial games as well, in particular with high branching factors, hidden information, and/or nondeterministic outcomes. To allow MCTS to be applied to games with increasingly higher branching factors, a variety of different MCTS variations have been developed. Numerous enhancements exist for MCTS to handle large branching factors, such as first-play urgency [21], which encourages exploitation in the early stages by assigning a fixed score to unvisited nodes. Another enhancement that has been shown to improve MCTS in Go is rapid action value estimation [22], which updates statistics in nodes, with a decreasing effect, when their corresponding action is selected during rollouts. Portfolio greedy search [23] and hierarchical portfolio search [24] introduced a Script-based approach, which have also been applied to MCTS [25], which deals with large branching factors in

real-time strategy games by exploring a search space of scripted behaviors instead of actions. NaïveMCTS builds a tree where each node corresponds to a combination of actions, and the exploration policy is based on a naive assumption that unit's actions are independent of other units' actions [4]. Portfolio greedy search and NaïveMCTS require that actions must be tied to units, as is common in real-time strategy games. Progressive strategies have been used to limit the search space with success in Go [26] by focusing the search using domain knowledge and then slowly unpruning nodes. Several progressive pruning methods have shown to improve MCTS for Go [27], where the idea is to prune nodes that are statistically inferior to their siblings. Sequential halving splits the time budget into a number of phases wherein exploration happens in a uniform manner, and after each phase, the worst half of the nodes are eliminated [28]. MCTS can use macroactions (repeated actions) to reduce the depth of the search tree, which can be beneficial in domains that require continuous control [29].

Fig. 1. The user interface of *Hero AIcademy* showing a battlefield of 9×5 squares with two deploy zones (yellow arrows) on each side, two crystals, and a number of units on each team. The symbols in the bottom represent the player's hand, and the numbers below the doors show the deck sizes.

B. Rolling Horizon and Online Evolutionary Algorithms

Evolutionary algorithms have been used to evolve controllers for numerous games [8]. Usually, learning happens offline as a fixed behavior is evolved in a training phase, while evolution is not applied during the game. Genetic programing has been used to evolve programs that can perform planning, where each candidate planner is evaluated by simulating the outcome of its generated plan [30]. Perez *et al.* introduced an evolutionary algorithm called RHE that runs online while the agent is playing to evolve action sequences. RHE has been applied, with good results, to a number of real-time environments including the Physical Traveling Salesman Problem [10] and many games in the General Video Game Playing benchmark [11], [12]. RHE evolves a sequence of actions for a fixed number of steps into the future. After the time budget is used, the first action in the most fit action sequence is performed, where after new actions sequences are evolved from scratch one step further into the future, i.e., the horizon is "rolling." The evolved action sequences are evaluated by simulating these in a forward model and evaluating the outcome. This paper introduces our algorithm OEP, which can be applied to multiaction adversarial games, and was first introduced in a conference paper [15]. OEP is more general than RHE, as it does not include the "rolling" approach. Instead, the entire time budget is used to evolve an action sequence for a complete turn, which will be performed to end. This paper will show that this algorithm works well in the turn-based adversarial game *Hero Academy*. Building on our previous conference paper [15], a portfolio-based version of OEP has already been tested for small-scale battles in *StarCraft* [31] and a continual variation of OEP for build order planning [32].

Another evolutionary algorithm that runs online while the game is being played is real-time neuroevolution of augmenting topologies [33]. This approach improves the behavior of multiple agents as the game is being played by replacing one individual every few game ticks with an offspring of the two most fit individuals. This method is, however, not directly applicable to the problem of searching for action sequences and is thus not a planning algorithm.

III. Testbed Game: *Hero Academy*

Before we describe our evolutionary algorithm and the two new MCTS variations in more detail, the testbed used for the experiments in this paper is described, which is a game called *Hero Academy*.[1] In order to run experiments efficiently, we use a simple clone of *Hero Academy* called *Hero AIcademy*.[2]

The two-player turn-based tactics game *Hero Academy* is inspired by chess, with battles similar to the ones in the *Heroes of Might & Magic* series. An example game state in the Hero AIcademy implementation is shown in Fig. 1. Each player has a pool of combat units and spells at their disposal, which they can deploy and use on a grid-shaped battle field of 9 × 5 squares. Special squares on the battlefield can boost a unit's attributes, while others allow the deployment of more units. Different classes of units have different combat roles, which allows players to employ a variety of different tactics. For example, the Council team has fighters, which are robust close-combat units that knock opponents back and wizards that can cast a powerful chain lightning spell, striking multiple units at once. Other units include archers, which are long-ranged units, clerics whose spells can heal friendly units, and a single ninja, a powerful close-combat unit with the ability to swap position with a friendly unit through teleportation. Each player has a hand of up to six cards and a deck from which they draw new cards each turn. Each card symbolizes either a unit, an item, or a spell.

The most central mechanic in the game is the usage of action points (APs). Each turn, the active player starts with five AP, which can be freely distributed among a number of different types of actions. These types are the following.

[1]http://www.robotentertainment.com/games/heroacademy/
[2]https://github.com/njustesen/hero-aicademy

1) *Deployment*: A unit can be deployed from the hand of cards onto an unoccupied deploy zone.
2) *Movement*: One unit can be moved a number of squares equal to or lower than its speed attribute.
3) *Attacking*: One unit can attack an opponent unit within the number of squares equal to its attack range attribute.
4) *Spell casting*: Each team has one unique spell that can be cast from the hand onto a square on the board, where after the spell card is discarded.
5) *Swapping a card*: A card on the hand can be shuffled into the deck in hopes of drawing other cards in the following round.
6) *Special*: Some units have special actions such as healing and teleportation. We will refer to healing as a unique action type in this paper.

Especially noteworthy is that a player may choose to distribute more than one AP per unit, i.e., let a unit act twice or more times per turn. Because players make multiple actions per turn, we call it a multiaction game. The first player to eliminate all enemy units or *crystals* wins the game. When a unit loses all of its health, it is not immediately removed from the game, but it instead becomes knocked down. Knocked down units have to be healed within one turn; otherwise, they are removed from the game. Units from the other team can, however, spend an AP during their turn to move a unit onto the square of a knocked down unit to remove it immediately, which is called stomping.

Because there are two win conditions, players can either go for one of them or try balance their strategy throughout the game. However, the key challenges in the game are the puzzles of finding the optimal action combination each turn. A clever action combination using several units and different types of actions can result in critical turnarounds during the game, and it is usually hard to plan several turns ahead.

A. Complexity Analysis

Due to the AP mechanic in *Hero Academy*, which makes the number of future game states significantly higher than in other games, the game is challenging for decision-making algorithms. Different combinations of actions can, however, result in the same game state. It is hardly feasible to determine the exact branching factor, as the number of allowed actions for a unit highly depends on the configuration of units on the board. Instead, it is trivial to estimate the branching factor by counting the number of possible actions in a recorded game. By doing this, we have estimated the branching factor to be 60 on average. The average branching factor per turn can, thus, be estimated to be $60^5 = 7.78 \times 10^8$ as players have five actions per turn. Also, based on observation, we estimated the game length to be around 40 rounds on average. The game-tree complexity can, thus, be estimated to $((60^5)^2)^{40} = 1.82 \times 10^{711}$ (the branching factor of a turn is squared since both players take turn during a round). The game-tree complexity of Chess is "just" 10^{120} [34]. Another interesting complexity measure is the size of the state space, i.e., the number of possible board configurations. To simplify this estimation, items and cards are ignored, and only situations with all four crystals on the board with full health are considered. There are $45 - 4 - 2 = 39$ possible squares to place a unit on a 9×5 board with two crystals and deploy zones for each player. There can be between 0 and 26 units on the board, each with a health value between 0 and around 800, which gives us the following formula for estimating the state-space complexity:

$$\prod_{n=0}^{26} \prod_{i=1}^{n} ((39 - i + 1) \times 800) = 1.57 \times 10^{199}. \quad (1)$$

Since the board configuration is only one part of the game state and items and cards are not considered, the state-space complexity of *Hero Academy* is much larger than our estimated lower bound. As a comparison, Chess has a state-space complexity of 10^{43} [34].

Hero Academy also introduces hidden information, as the opponent's cards are unknown, as well as the order of cards in the deck. Stochastic games with hidden information can be approached with various forms of determinization methods [35], [36]. In this paper, we ignore the aspect of hidden information and randomness, since we are only interested in ways to deal with the complexities of multiaction games.

IV. Online Evolutionary Planning

In this section, we present an evolutionary algorithm that, inspired by the RHE, evolves strategies while it plays the game. We call this algorithm OEP, and we have implemented it to play *Hero Academy*, where it aims to evolve optimal action sequences every turn. Each genome in a population, thus, represents a sequence of five actions. An exhaustive 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.

An overview of our OEP algorithm is presented next, which is also shown in pseudocode (see Algorithm 1). Evolutionary algorithms iteratively optimize an initially randomized population of candidate solutions (genomes) through recombination and selection based on elitism. When applied to planning in *Hero Academy* games, the genotype of each genome in the population is a vector of five actions, where one action represents a type and one or more locations if needed. An example of a genotype is: [**Move**$((0,4) \rightarrow (2,4))$, **Heal**$((2,4) \rightarrow (4,4))$, **Heal**$((2,4) \rightarrow (4,4))$, **Attack**$((4,0) \rightarrow (6,1))$, **Deploy**$(0 \rightarrow (0,4))$], which is also visualized in Fig. 2. Note that identical attack or heal actions can be repeated to deal more damage or gain more health. Locations given in two dimensions are on the board (from the top left square) and in one dimension are cards on the hand (from left to right). The phenotype is the resulting game state after taking these actions in the current game state.

The initial population is composed of random genomes, which are created by repeatedly selecting random actions based on the given forward model. This process is repeated until no more APs are left. After the creation of the initial population, the population is improved over a large number of generations until a given time budget is exhausted.

(a) (b) (c)

Fig. 2. Example of an evolved action sequence that demonstrates that OEP can find solid action sequences in *Hero Academy*. In this example a critical combination is found that first heals the knocked down knight at (4, 4). Because the knight afterwards stands on an assault square, any damage towards crystals are doubled which the archer utilizes by a lethal strike. (a) shows the initial state in the beginning of the red player's turn and (b) shows the actions evolved by OEP. The exact action sequence, shown in (b), is: [**Move**$((0,4) \rightarrow (2,4))$, **Heal**$((2,4) \rightarrow (4,4))$, **Heal**$((2,4) \rightarrow (4,4))$, **Attack**$((4,0) \rightarrow (6,1))$, **Deploy**$(0 \rightarrow (0,4))$]. The resulting state after performing the evolved action sequence and ending the turn is shown in (c). (a) Initial state. (b) Evolved actions. (c) Resulting state.

Algorithm 1: EOP for multiaction adversarial games (procedures PROCREATE, CROSSOVER, MUTATION, CLONE, and EVAL are omitted for brevity).

```
1:  procedure ONLINEEVOLUTIONARYPLANNING(State s)
2:      Genome[] pop = ∅                          ▷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   ▷ EVAL is deterministic
9:                  g.value = EVAL(clone)
10:             g.visits++
11:         sort pop in descending order by 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 = ∅
26:     Boolean p1 = s.p1                  ▷Whose 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
```

In this paper, *Hero AIcademy* itself serves as the forward model, and the fitness of an action sequence is calculated as the difference between the values of both players' units. Both the units on the game board as well as those still at the players' disposal are taken into account. The assumption behind this particular fitness function is that the difference in units serves as a good indicator, for which player is more likely to win. In more detail, the value $v(u)$ of unit u is calculated as follows:

TABLE I
BONUSES ADDED WHEN UNITS ARE EQUIPPED [USED BY $eq(u)$], AND BONUSES ADDED WHEN UNITS STAND ON SPECIAL SQUARES [USED BY $sq(u)$]

	Archer	Cleric	Knight	Ninja	Wizard
Equipment:					
Dragonscale	30	30	30	30	20
Runemetal	40	20	−50	20	40
Helmet	20	20	20	10	20
Scroll	50	30	−40	40	50
Special squares:					
Assault	40	10	120	50	40
Deploy	−75	−75	−75	−75	−75
Defense	80	20	30	60	70
Power	120	40	30	70	100

$$v(u) = u_{hp} + \underbrace{u_{\text{max}hp} \times up(u)}_{\text{standing bonus}} + \overbrace{eq(u) \times up(u)}^{\text{equipment bonus}} + \underbrace{sq(u) \times (up(u) - 1)}_{\text{square bonus}} \tag{2}$$

where u_{hp} is the unit's number of health points, $sq(u)$ is a bonus based on the type of the square the unit stands on, and $eq(u)$ is a bonus that depends on the unit's equipment. The exact modifiers are shown in Table I. The modifying term $up(u)$ gives a negative reward for knocked down units:

$$up(u) = \begin{cases} 0, & \text{if } u_{hp} = 0 \\ 2, & \text{otherwise.} \end{cases} \tag{3}$$

After the evaluation, the genomes with the lowest scores are removed from the population. Each one of the remaining genomes is paired with another randomly selected genome, creating a new offspring through uniform crossover. Fig. 3 shows the crossover between two example action sequences in *Hero Academy*. Because a naive crossover can lead to illegal action sequences, the crossover checks for the legality of a move when

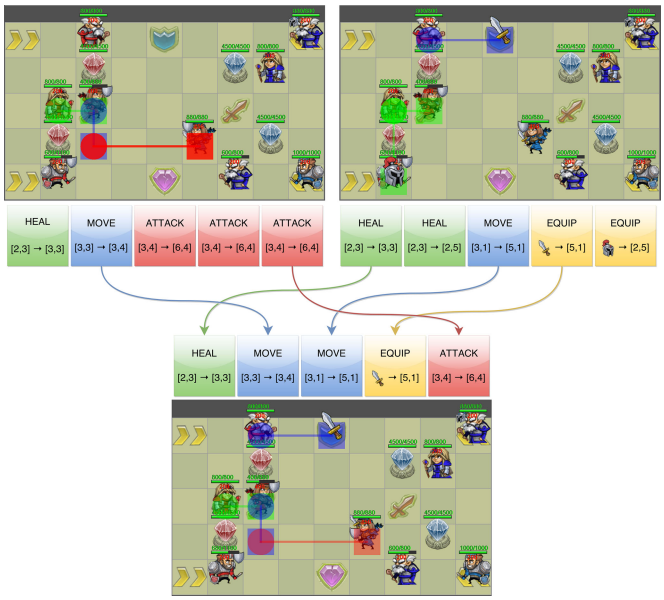

Fig. 3. Uniform crossover example in *Hero Academy*. Genes (actions) are randomly picked from the two parents.

two sequences are combined. For example, to be able to move a unit from a certain position on the board, it is required that a unit, in fact, stands on that particular square. However, this precondition might not be fulfilled due to an earlier action in the sequence. To avoid such situations, actions are only selected from a randomly chosen parent if it can be performed legally; otherwise, the action will be taken from the other parent. If neither one of the two actions results in a legal move, the next action in the parent's sequence is chosen. In case this fails as well, a completely random available action is selected. Additionally, a certain proportion of the created offspring is mutated to introduce new actions to the population. Mutation changes one randomly chosen action to another legal action. If this results in an illegal action sequence, the following part of the sequence is changed to random but legal actions. Legality is checked by requesting the framework for available actions at each state by traversing the offspring's actions sequence.

To incorporate information about possible counter moves, attempts were made to base the heuristic on rollouts. Here, fitness is determined by performing one rollout with a depth limit of five actions, corresponding to one turn. When a genome is tested more than once (because it has survived several generations in the population), the lowest value found in the genome's evaluations is used. The rating of an action sequence, thus, depends on the best of the known countermoves. Because our experiments did not show any significant difference between a stochastic rollout as a fitness measure and a static evaluation, the later was chosen for the experiments in this paper. The large branching factor in this game is possibly the reason why evaluations are unreliable when based on a low number of rollouts.

V. Tree Search

A game tree can be described as an acyclic directed graph with the root node being the current game state. Edges in the graph represent available actions in the game that lead from one state to other hypothetical future game states. Therefore, the number of edges from a node corresponds to the number of actions available to the active player in that game state. Nodes also have certain values assigned to them, where higher values indicate more desirable game situations. For many adversarial games, in which the utility of one agent is the opposite of the other, agents take turns, and thus, the active player alternates between plies of the tree. In these situations, the well-known Minimax algorithm can be applied. However, in *Hero Academy*, players can take several actions before the end of their turn. A potential tree search setup for *Hero Academy* could be to encode multiple actions as a joint action (i.e., an array of actions) that is assigned to each edge. However, because of the high number of possible permutations and therefore increased branching factor, we decided to model each action as its own node, essentially trading tree breath for depth.

In the following, we will present five game-playing tree-search methods for *Hero Academy*. Two of these are simple game-tree-based methods, which were used as baselines, followed by MCTS including two novel variations.

Greedy Action: The greedy search among actions method is the simplest of the developed methods. Greedy Action performs a one-ply search among all possible actions, and based on the employed heuristic, (2) selects the action that leads to the most promising game state. The search is invoked five times to complete a turn.

Greedy Turn: The greedy search among turns performs a five-ply depth-first search, which corresponds to a full turn. The same heuristic as for Greedy Action rates all leaf nodes states and selects the action sequence that leads to the highest rated state. A transposition table keeps track of already visited game states so that they are not visited again. Because this search is usually not exhaustive, choosing which actions to try first (i.e., action sorting) is critical.

Vanilla MCTS: Following the two greedy search variants, the vanilla MCTS method was implemented with an action-based approach. In other words, one ply in the tree represents an action, not a turn. Therefore, a search with a depth of five has to be performed to reach the beginning of the opponent's turn. Our vanilla MCTS agent implements the four phases described in Section II-A, which follows the traditional MCTS algorithm using UCB [14]. The standard MCTS backpropagation was modified to handle two players with multiple actions. Our approach [15] is an extension of the *BackupNegamax* [14] algorithm (see Algorithm 2). It uses a list of edges reflecting 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.

An ϵ-greedy approach is employed in the rollouts that combines random play with the highest rated action based on the amount of damage dealt/healed. Rollouts that stop prior to a terminal state are evaluated by our heuristic [see (2)].

As players in *Hero Academy* have to select five actions, two different approaches were tried: in the first approach, the agent is invoked five times sequentially, with each iteration using a fifth of the total time budget, whereafter the actions are executed in

Algorithm 2: A multiaction game modification of the BackupNegamax algorithm [15].

```
1:  procedure MultiNegamax(Edge[] T, Double Δ, Boolean p1)
2:      for all Edge e in T do
3:          e.visits++
4:          if e.to ≠ 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 += Δ
10:         else
11:             e.value −= Δ
```

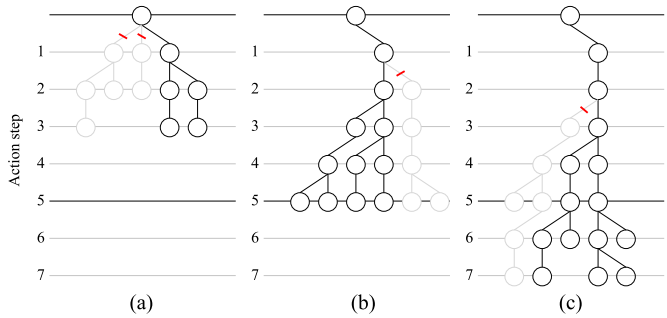

Fig. 4. Example of how nodes are progressively pruned using the bridge burning strategy in three phases to force the search to reach deeper plies (action steps). (a) After the first phase, all nodes except the most promising in ply one are pruned. (b) and (c) Search continues where after nodes are pruned one ply deeper.

the order found. However, another approach was found to be superior that uses the entire time budget, whereafter the tree is traversed from the root by selecting the best five nodes (actions) until the opponents turn is reached. This will allow the tree to expand as much as possible before actions are selected, and thus, more options that are complete will be considered within the time budget.

The following describes two novel exploration-constrained variations of MCTS. We will refer to MCTS without these extensions as Vanilla MCTS.

Nonexploring MCTS: The search trees created by MCTS will barely be able to reach into the opponent's turn due to the complexity of the game. To overcome this, we have developed a variation of MCTS that uses a nonexploring tree policy, i.e., $C = 0$, in combination with deterministic rollouts. All children of a node are still visited at least once before any of them are expanded further. This ensures that a very limited form of exploration still occurs, and since rollouts are deterministic, controlled by a greedy policy, it is acceptable to value children based only on one visit. If stochastic rollouts were used instead, some branches would never be revisited when rollouts happen to return unlucky outcomes.

BB-MCTS: Another novel approach to MCTS in multiaction games is what we call the BB-MCTS. This approach splits the time budget into a number of sequential phases equal to the number of actions in a turn. During each phase, the search functions as an ordinary MCTS search. However, in the end of each phase, all but the most promising node from the root are pruned and will never be added again. Another way to implement the same behavior is to treat the most promising child node as the root of the tree in the following phase. This approach is, thus, an aggressive progressive pruning strategy that will enable the search to reach deeper plies with the drawback of ignoring parts of the search space. The name bridge burning emphasizes that the nodes are aggressively pruned and can never be visited again. Fig. 4 shows how nodes are pruned in three phases in a multiaction game with three actions in a turn.

For the tree-search methods, it makes sense to investigate the most promising moves first, and thus, a method for sorting actions is useful. A simple way would be to evaluate the resulting game state of each action, but this is usually a slow method. Instead, attack and spell actions are rated by how much damage they will deal, heal actions (including healing potions) by how many health points they will heal, equip actions by $p_u(h_u/m_u)$, where p_u, h_u, and m_u are the power (equivalent to damage output) of the equipped unit u, health points, and maximum health points. Movement actions are given a 30-point rating if movement is to a special square type and 0 otherwise. If an enemy unit is removed from the game, it is given a $2m_u$ point rating. If a knocked down unit u is healed, the rating of the action is $m_u + |e_u| \times 200$, where $|e_u|$ is the number of items carried by the healed unit u.

BB-MCTS is to some extend similar to an extreme implementation of sequential halving where only one node survives, and the number of phases is fixed to the number of actions.

VI. Experimental Setup

Here, the experimental setup is described, which is the basis for the results presented in the next section. Each method played 100 games (as the Council team) against all other methods, 50 games as the starting player and 50 games as the second player. Fig. 1 shows the map used. In contrast to the original game, Hero AIcademy was configured to be deterministic and without hidden information to focus our experiments on the challenge of performing multiple actions. Each method was limited to one processor and had a time budget of 6 s each turn. The winning percentages of each matchup counted draws as half a win for each player. While the rules of the original *Hero Academy* do not include draws, in these experiments, a draw was called if no winner was found in 100 rounds. The experiments were carried out on an Intel Core i7-3517U CPU with 4×1.90 GHz cores and 8 GB of ram. The specific configurations for the implemented game-playing methods were as follows.

Vanilla MCTS: The traditional UCT tree policy $\overline{X}_j + 2C\sqrt{(2\ln n/nj)}$ was employed with an exploration constant of $C = (1/\sqrt{2})$. The default policy was ϵ-greedy, where $\epsilon = 0.5$. A transposition table was implemented with the *descent-path only* backpropagation strategy. Thus, values and visit counts are stored in edges [37]. In fact, n_j in the tree policy is extracted from the child edges instead of the nodes. Rollouts

TABLE II
WIN PERCENTAGES OF THE AGENTS LISTED IN THE LEFT-MOST COLUMN IN 100 GAMES AGAINST AGENTS LISTED IN THE TOP ROW

	Random	Greedy Action	Greedy Turn	Vanilla MCTS	Non-expl. MCTS	BB-MCTS	OEP
Greedy Action	100%	–	36%	51.5%	2%	7.0%	2.0%
Greedy Turn	100%	64.0%	–	88.0%	23.0%	26.5%	19.5%
Vanilla MCTS	100%	48.5%	22.0%	–	0.0%	4.5%	2%
Non-exploring MCTS	100%	98.0%	77.0%	100%	–	80.0%	58.0%
BB-MCTS	100%	93.0%	73.5%	95.5%	20.0%	–	48.0%
OEP	100%	90.0%	80.5%	98%	42.0%	52.0%	–

A win percentage of 62% or more is significant with a significance level of 0.05 using the Wilcoxon Signed-Rank Test.

were depth-limited to one turn, following the heuristic state evaluator described in Section IV. Preliminary experiments showed that short rollouts are preferred over long rollouts for MCTS in *Hero Academy*, and that rollouts of just one turn show the best performance. Additionally, by adding some domain knowledge to the default policy through a specific ϵ-greedy strategy, the performance improves; ϵ-greedy selects a greedy action equivalent to the highest rated action by the action sorting method with a probability of ϵ, and a random action otherwise.

BB-MCTS: Same as for Vanilla MCTS, but with the Bridge Burning strategy as well.

Nonexploring MCTS: Same as for Vanilla MCTS, but with the exploration constant $C = 0$ and the default policy ϵ-greedy, where $\epsilon = 1$, such that rollouts are deterministic following the action sorting heuristic (see Section V).

OEP: The population size was 100 with a survival rate of 0.5, a mutation probability of 0.1, and a uniform crossover operator. These parameters were found through prior experimentation.

The MCTS-based methods and OEP employ action pruning, which reduces the large search space of a turn by removing (pruning) redundant swap actions and suboptimal spell actions from the set of available actions in a state. Swap actions are redundant if they swap the same kind of item (i.e., they will produce the same outcome) and can thus be removed. Additionally, spell actions are suboptimal if other spell actions cover the same or more enemy units.

VII. RESULTS

Our results, shown in Table II, show the performance of each method. The nonexploring MCTS is the best performing method but with no significant difference compared to OEP and BB-MCTS, which have a similar performance. Vanilla MCTS plays on the same level as the greedy action baseline, 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. A video of OEP playing against the greedy action baseline has been uploaded to YouTube.[3]

A. Search Characteristic Comparison

To get further insights into how the different methods explore the search space, the number of different action sequences each method is able to evaluate within the given time budget was tracked. Since many action sequences produce the same outcome, only the number of unique outcomes evaluated by each method were counted and only those after taking five actions. The "greedy turn" search evaluated 579 912 unique outcomes on average during a turn. OEP evaluated on average 9344 unique outcomes and MCTS only 201. Each node at the fifth ply of the MCTS tree corresponds to one outcome, and the search only manages to expand the tree to a limited number of nodes at this depth. Further analysis reveals that the average depth of leaf nodes in the final MCTS 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. OEP ran an average of 3693 generations each turn but appeared to get stuck at local optima quickly due to the low number of unique outcomes evaluated. These results suggest that OEP would play almost equally good with a much lower time budget, but also that there could be room to improve the algorithm itself.

B. Changing the Number of Actions

Multiaction games can have many forms and *Hero Academy* is just one example. An additional experiment was performed, in which our methods were tested in variations of *Hero Academy*. The rules were altered by changing the number of APs per turn to 1, 3, 5, 10, 15, 20, and 25. This also increases the complexity of one turn exponentially. The time budget in this experiment was set to 2000 ms, and the turn limit to $5/AP \times 100$. Reaching the turn limit results in a draw, which is counted as half a win for both players. Only MCTS with our two variations as well as OEP are included in this experiment, as it makes the most interesting comparison. The results, which are plotted in Fig. 5, show the win percentage of OEP against each MCTS variation in 100 games. The results show that OEP handles the increased number of APs best with a win rate of 55% or more with 10 or more AP against any of the other methods. This indicates that OEP has the best scalability in terms of complexity in multiaction games. Increasing the number of AP to 20 and more makes it possible to win the game in a few turns. This make the outcome of the game highly depend on who gets to start. Vanilla MCTS does, however, not show that it is able to identify these fast win strategies.

[3]https://www.youtube.com/watch?v=WYGwBbccYCQ

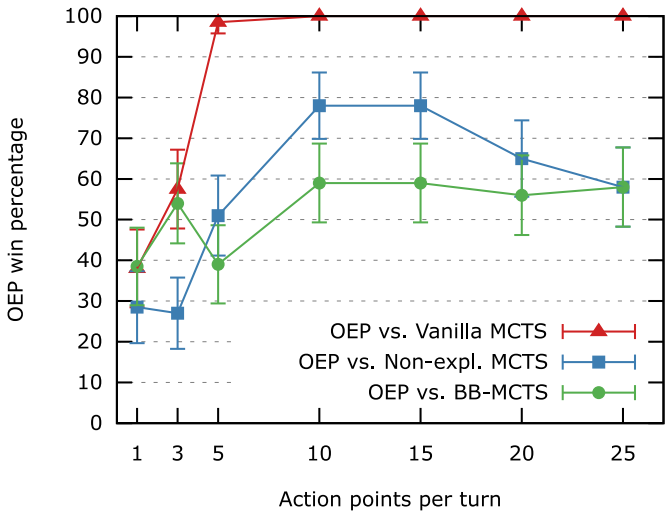

Fig. 5. Average win percentages of OEP versus BB-MCTS, nonexploring MCTS, and Vanilla MCTS. 100 games were played in each matchup for each game configuration, which each had a different number of AP per turn. This shows that MCTS, including the two variations, performs best with few AP per turn, while OEP performs best with many AP per turn. Error bars show 95% confidence intervals.

C. Versus Human Players

One important and often overseen experiment is to test game-playing algorithms against human players. Such comparisons can provide important insights about the current state of a method, and whether there is promise implementing it in a real game product. Since OEP, BB-MCTS, and nonexploring MCTS show similar performance in games with five APs per turn, we suspect this is also the case when compared against humans. Thus, in this paper, we focus on only testing OEP against human players but believe a more comprehensive comparison that includes MCTS remains important future work. One hundred and eleven games were recorded, which were played with hidden information and randomness as in the actual game. It makes no difference for OEP whether or not the game contains randomness as it only considers its own turn. The Hero AIcademy client was distributed on social networks and community web sites for *Hero Academy* players to reach a sufficient number of players with various skill level. The client was extended to push game events to a web server every turn to be stored in a database. Unfortunately, we do not know the number of participants, only the number of games played in total. The client asked players about their skill level prior to each game with the options beginner, intermediate, and expert. If beginner was selected, a very short introduction to the game was presented before the game started. One hundred and eleven game records were collected, and the results are shown in Table III. In 56 of the 111 games, the game was quit before a terminal state was reached, possibly as a way of surrendering. Of the games that reached the end, the human players won 41 out of 55 games. Games that were quit before a winner was found were evaluated using the heuristic described in Section IV to determine a winner. Games were, however, excluded if the heuristic estimated an advantage lower than 10% of the maximum to either of the players. By including these games, the human players won only 63.5% of the games showing that many players left the game while they were behind. These results show that OEP is competitive against human players while being inferior to expert players and some intermediate players.

TABLE III
NUMBER OF WINS BY HUMAN PLAYERS OF VARIOUS SKILL LEVELS IN 85 GAMES AGAINST OEP

Skill level	Human wins (a)	Human wins (b)
Beginner	13/26 (50%)	19/45 (42.2%)
Intermediate	12/13 (92%)	15/19 (78.9%)
Expert	16/16 (100%)	20/21 (95.2%)
All	41/55 (80%)	**54/85 (63.5%)**

(a) shows results from games that ended with a winner, and (b) also includes games where the player quit prematurely.

VIII. DISCUSSION

The results show that OEP performs better than all tree search methods for most of the settings of our benchmark, and as the number of actions per turn (and thus the branching factor) increases, the relative advantage of OEP over tree search seems to increase. The crucial question is why this is so. Answering this question will require much further research and is likely to contribute to our understanding of planning in general with broad domain-general implications. Our current understanding is that tree search algorithms, which by definition start from the root node of the tree and explore outward, concentrate their search on the part of plan space closest to the root in problems with a high branching factor. However, all parts of the plan are important. By seeing the plan as a string and searching the space of strings, the evolutionary planning approach ensures that the search considers full plans instead of focusing on the region around the origin.

Vanilla MCTS is unable to deal with the complexity of *Hero Academy* as the search space is simply too large. A similar conclusion has been made for the game Arimaa, in which the player makes four actions each turn [38]. We show that by constraining the exploration of MCTS, performance can be significantly improved. In the future, it will be interesting to compare our approaches to MCTS with optimized parameters as well as existing MCTS enhancements that have shown to work well in Go, such as progressive strategies [26], [27]. They would most likely need to be configured to constrain the exploration aggressively as our methods do. MCTS was not designed to deal with multiple actions each turn, and little research has been done on problems of this type. Clearly, more research is needed since many strategy games are multiaction games, which may also model real-life decision making.

One problem with OEP is that it does not take the opponent's turn into account. This could perhaps be achieved with competitive coevolution [39] by having another population for the seconds player's turn and let the genomes in each population compete when determining their fitness. One problem with this idea is that evolved action sequences in the other population

highly depend on the outcome of the action sequences in the first population (e.g., attacking a certain square is only effective if the opponent actually moves a unit there). Coevolution has already been tested for the RHE for games with small branching factors [40], and applying this to OEP for multiaction games is an interesting direction for future research. OEP reached an optimum with only a small amount of generations, which means that it either finds the global optimum or is stuck locally. Diversity maintenance methods (such as niching [41]) or Tabu search could perhaps improve the performance [42]. Finally, it would be very interesting to see how OEP performs in more complex multiaction games.

IX. Conclusion

This paper described and compared three methods for playing adversarial games with very large branching factors. Such branching factors, while extreme in comparison to classical board games, are common in strategy games and presumably also in the real-world scenarios they model. To tackle this challenge, we propose OEP. 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 MCTS, which is the state of the art for many games with high branching factor, as well as two new variations of MCTS designed to better handle the high branching factor. Nonexploring MCTS does not explore other paths than the currently most promising one and uses deterministic rollouts, while BB-MCTS deepens the search by periodically removing upper parts of the tree so that only lower parts are explored. Our results show that both OEP and the two new MCTS variations convincingly outperform standard MCTS as well as a depth first search on this problem. As for the relative performance of the new methods, nonexploring MCTS slightly outperforms OEP on small numbers of actions, but as the number of actions per turn (and therefore the branching factor) is increased, OEP's superiority over other approaches increases. Wang *et al.* also conclude that OEP outperforms MCTS when the complexity of the problem increases [31]. We further tested OEP's ability to play against human players of various skill levels. OEP was able to win 31 games out of 85 (36.5%) and can play competitively against human players with a low-to-medium skill level while being easily outplayed by medium-to-high-level players. Future work will investigate how well this performance holds up in other games, and how to improve the evolutionary search.

References

[1] J. v. Neumann, "Zur theorie der gesellschaftsspiele," *Math. Ann.*, vol. 100, no. 1, pp. 295–320, 1928.

[2] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] B. Bouzy and T. Cazenave, "Computer go: An AI oriented survey," *Artif. Intell.*, vol. 132, no. 1, pp. 39–103, 2001.

[4] S. Ontanón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *Proc. 9th Artif. Intell. Interactive Digit. Entertainment Conf.*, 2013, pp. 58–64.

[5] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling horizon evolution versus tree search for navigation in single-player real-time games," in *Proc. 15th Annu. Conf. Genetic Evol. Comput.*, 2013, pp. 351–358.

[6] J. Togelius, S. Karakovskiy, J. Koutník, and J. Schmidhuber, "Super Mario evolution," in *Proc. IEEE Symp. Comput. Intell. Games*, 2009, pp. 156–161.

[7] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Evolving competitive car controllers for racing games with neuroevolution," in *Proc. 11th Annu. Conf. Genetic Evol. Comput.*, 2009, pp. 1179–1186.

[8] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *IEEE Trans. Comput. Intell. AI Games*, vol. 9, no. pp. 25–41, Mar. 2017.

[9] S. M. Lucas and G. Kendall, "Evolutionary computation and games," *IEEE Comput. Intell. Mag.*, vol. 1, no. 1, pp. 10–18, Feb. 2006.

[10] D. Perez, P. Rohlfshagen, and S. M. Lucas, "Monte-Carlo tree search for the physical travelling salesman problem," in *Applications of Evolutionary Computation*. New York, NY, USA: Springer, 2012, pp. 255–264.

[11] R. D. Gaina, J. Liu, S. M. Lucas, and D. Pérez-Liébana, "Analysis of vanilla rolling horizon evolution parameters in general video game playing," in *Proc. Eur. Conf. Appl. Evol. Comput.*, 2017, pp. 418–434.

[12] J. Levine *et al.*, "General video game playing," *Artif. Comput. Intell. Games*, vol. 6, pp. 77–83, 2013.

[13] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game AI," in *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertainment*, 2008, pp. 216–217.

[14] C. B. Browne *et al.*, "A survey of monte carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[15] N. Justesen, T. Mahlmann, and J. Togelius, "Online evolution for multi-action adversarial games," in *Proc. Eur. Conf. Appl. Evol. Comput.*, Springer, 2016, pp. 590–603.

[16] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Proc. 5th Int. Conf. Comput. Games*, 2006, pp. 72–83.

[17] S. Branavan, D. Silver, and R. Barzilay, "Non-linear monte-carlo search in Civilization II," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, 2011, pp. 2404–2410.

[18] P. I. Cowling, C. D. Ward, and E. J. Powley, "Ensemble determinization in monte carlo tree search for the imperfect information card game Magic: The gathering," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 4, pp. 241–257, Dec. 2012.

[19] I. Szita, G. Chaslot, and P. Spronck, "Monte-Carlo tree search in Settlers of Catan," in *Proc. 12th Int. Conf. Adv. Comput. Games*, 2009, pp. 21–32.

[20] P. Audouard, G. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud, "Grid coevolution for adaptive simulations: Application to the building of opening books in the game of go," in *Proc. Workshops Appl. Evol. Comput.*, 2009, pp. 323–332.

[21] S. Gelly and Y. Wang, "Exploration exploitation in go: UCT for Monte-Carlo go," in *Proc. Neural Inf. Process. Syst. Conf. On-line Trading Exploration Exploitation Workshop*, 2006.

[22] D. P. Helmbold and A. Parker-Wood, "All-moves-as-first heuristics in Monte-Carlo go," in *Proc. Int. Conf. Artif. Intell.*, 2009, pp. 605–610.

[23] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in StarCraft," in *Proc. IEEE Conf. Comput. Intell. Games*, 2013, pp. 1–8.

[24] D. Churchill and M. Buro, "Hierarchical portfolio search: Prismatas robust AI architecture for games with large search spaces," in *Proc. Artif. Intell. Interact. Digit. Entertainment Conf.*, 2015, pp. 16–22.

[25] N. Justesen, B. Tillman, J. Togelius, and S. Risi, "Script-and cluster-based UCT for StarCraft," in *Proc. IEEE Conf. Comput. Intell. Games*, 2014, pp. 1–8.

[26] G. M. J. Chaslot, M. H. Winands, H. J. van den Herik, J. W. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte-Carlo tree search," *New Math. Natural Comput.*, vol. 4, no. 3, pp. 343–357, 2008.

[27] B. Bouzy, "Move-pruning techniques for Monte-Carlo go," in *Advances in Computer Games*. New York, NY, USA: Springer, 2005, pp. 104–119.

[28] Z. Karnin, T. Koren, and O. Somekh, "Almost optimal exploration in multi-armed bandits," in *Proc. 30th Int. Conf. Mach. Learn.*, 2013, pp. 1238–1246.

[29] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Monte Carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem," in *Proc. IEEE Conf. Comput. Intell. Games.*, 2012, pp. 234–241.

[30] J. R. Koza, "Genetically breeding populations of computer programs to solve problems in artificial intelligence," in *Proc. 2nd Int. IEEE Conf. Tools Artif. Intell.*, 1990, pp. 819–827.

[31] C. Wang, P. Chen, Y. Li, C. Holmgård, and J. Togelius, "Portfolio online evolution in StarCraft," in *Proc. 12th Artif. Intell. Interactive Digital Entertainment Conf.*, 2016, pp. 114–120.
[32] N. Justesen and S. Risi, "Continual online evolution for in-game build order adaptation in StarCraft," in *Proc. Genetic Evol. Comput. Conf.*, 2017, pp. 187–194.
[33] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the NERO video game," *IEEE Trans. Evol. Comput.*, vol. 9, no. 6, pp. 653–668, Dec. 2005.
[34] C. E. Shannon, "XXII. programming a computer for playing chess," *London, Edinburgh, Dublin Philosoph. Mag. J. Sci.*, vol. 41, no. 314, pp. 256–275, 1950.
[35] T. Cazenave, "A phantom-go program," in *Advances in Computer Games*. New York, NY, USA: Springer, 2005, pp. 120–125.
[36] P. Nijssen and M. H. Winands, "Monte carlo tree search for the hide-and-seek game Scotland Yard," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 4, pp. 282–294, Dec. 2012.
[37] A. Saffidine, T. Cazenave, and J. Méhat, "UCD: Upper confidence bound for rooted directed acyclic graphs," *Knowl.-Based Syst.*, vol. 34, pp. 26–33, 2012.
[38] T. Kozelek, "Methods of MCTS and the game Arimaa," master's thesis, Faculty Math. Phys., Charles Univ., Prague, Czech Republic, 2009.
[39] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution," *Evol. Comput.*, vol. 5, no. 1, pp. 1–29, 1997.
[40] J. Liu, D. Pérez-Liébana, and S. M. Lucas, "Rolling horizon coevolutionary planning for two-player video games," in *Proc. 8th Comput. Sci. Electron. Eng.*, 2016, pp. 174–179.
[41] S. W. Mahfoud, "Niching methods for genetic algorithms," Ph.D. dissertation, Illinois Genetic Algorithm Lab., Univ. Illinois Urbana-Champaign, Urbana, IL, USA, 1995.
[42] F. Glover and M. Laguna, *Tabu Search**. New York, NY, USA: Springer, 2013.

Niels Justesen received the B.Sc. degree in software development and the M.Sc. degree in games technology from the IT University of Copenhagen, Copenhagen, Denmark, in 2012 and 2015, respectively.

He is a Ph.D. fellow with the IT University of Copenhagen, where he is part of the Center for Computer Games Research and the Robotics, Evolution and Art Lab. He has previously worked at IT Minds. His research interests include game-playing algorithms for strategy games, including tree search, evolutionary algorithms, and deep learning.

Tobias Mahlmann received the M.Sc. degree from the Technical University of Braunschweig, Braunschweig, Germany, in 2006, and the Ph.D. degree from the IT University of Copenhagen, Copenhagen, Denmark, in 2012.

He also has previously worked at Lund University. He works on different aspects of procedural content generation for games, mostly focusing on game mechanics. His other directions of research include game metrics mining for improving gameplay experiences and agents' decision-making functions.

Sebastian Risi received the Ph.D. degree in computer science from the University of Central Florida, Orlando, FL, USA, in 2012.

He is an Associate Professor with the IT University of Copenhagen, Copenhagen, Denmark, where he is part of the Center for Computer Games Research and the Robotics, Evolution and Art Lab. He is a consultant for the recently formed Uber AI labs and was a co-Founder of FinchBeak, a company that focused on casual and educational social games enabled by AI technology. His research interests include neuroevolution, evolutionary robotics, and human computation.

Dr. Risi has received several best paper awards at the Genetic and Evolutionary Computation Conference, the International Conference on Computational Intelligence in Music, Sound, Art and Design, the International Joint Conference on Neural Networks, and the Continual Learning Workshop at NIPS for his work on adaptive systems, the HyperNEAT algorithm for evolving complex artificial neural networks, and music generation.

Julian Togelius received the B.A. degree from Lund University, Lund, Sweden, in 2002, the M.Sc. degree from the University of Sussex, Brighton, U.K., in 2003, and the Ph.D. degree from the University of Essex, Colchester, U.K., in 2007.

He is an Associate Professor with the Department of Computer Science and Engineering, New York University, New York, NY, USA. He has previously worked at IDSIA in Lugano and at the IT University of Copenhagen. He works on all aspects of computational intelligence and games and on selected topics in evolutionary computation and evolutionary reinforcement learning. His current main research directions involve search-based procedural content generation in games, general video game playing, player modeling, and fair and relevant benchmarking of AI through game-based competitions.

Dr. Togelius is a past chair of the IEEE Computational Intelligence Society Technical Committee on Games and an Associate Editor of the IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES.