Rolling Horizon Coevolutionary Planning for Two-Player Video Games

Jialin Liu
University of Essex
Colchester CO4 3SQ
United Kingdom
jialin.liu@essex.ac.uk

Diego Pérez-Liébana
University of Essex
Colchester CO4 3SQ
United Kingdom
dperez@essex.ac.uk

Simon M. Lucas
University of Essex
Colchester CO4 3SQ
United Kingdom
sml@essex.ac.uk

Abstract—This paper describes a new algorithm for decision making in two-player real-time video games. As with Monte Carlo Tree Search, the algorithm can be used without heuristics and has been developed for use in general video game AI.

The approach is to extend recent work on rolling horizon evolutionary planning, which has been shown to work well for single-player games, to two (or in principle many) player games. To select an action the algorithm co-evolves two (or in the general case N) populations, one for each player, where each individual is a sequence of actions for the respective player. The fitness of each individual is evaluated by playing it against a selection of action-sequences from the opposing population. When choosing an action to take in the game, the first action is chosen from the fittest member of the population for that player.

The new algorithm is compared with a number of general video game AI algorithms on a two-player space battle game, with promising results.

I. INTRODUCTION

Since the dawn of computing, games have provided an excellent test bed for AI algorithms, and increasingly games have also been a major AI application area. Over the last decade progress in game AI has been significant, with Monte Carlo Tree Search dominating many areas since 2006 [1]–[5], and more recently deep reinforcement learning has provided amazing results, both in stand-alone mode in video games, and in combination with MCTS in the shape of AlphaGo [6], which achieved one of the greatest steps forward in a mature and competitive area of AI ever seen. AlphaGo not only beat Lee Sedol, previous world Go champion, but also soundly defeated all leading Computer Go bots, many of which had been in development for more than a decade. Now AlphaGo is listed at the second position among current strongest human players[1].

With all this recent progress a distant observer could be forgiven for thinking that Game AI was starting to become a solved problem, but with smart AI as a baseline the challenges become more interesting, with ever greater opportunities to develop games that depend on AI either at the design stage or to provide compelling gameplay.

The problem addressed in this paper is the design of general algorithms for two-player video games. As a possible solution, we introduce a new algorithm: Rolling Horizon Coevolution

Algorithm (RHCA). This only works for cases when the forward model of the game is known and can be used to conduct what-if simulations (roll-outs) much faster than real time. In terms of the General Video Game AI (GVGAI) competition[2] series, this is known as the two-player planning track [7]. However, this track was not available at the time of writing this paper, so we were faced with the choice of using an existing two-player video game, or developing our own. We chose to develop our own simple battle game, bearing some similarity to the original 1962 Spacewar game[3] though lacking the fuel limit and the central death star with the gravity field. This approach provides direct control over all aspects of the game and enables us to optimise it for fast simulations, and also to experiment with parameters to test the generality of the results.

The two-player planning track is interesting because it offers the possibility of AI which can be instantly smart with no prior training on a game, unlike Deep Q Learning (DQN) approaches [8] which require extensive training before good performance can be achieved. Hence the bots developed using our planning methods can be used to provide instant feedback to game designers on aspects of gameplay such as variability, challenge and skill depth.

The most obvious choice for this type of AI is Monte Carlo Tree Search (MCTS), but recent results have shown that for single-player video games, Rolling Horizon Evolutionary Algorithms (RHEA) are competitive. Rolling horizon evolution works by evolving a population of action sequences, where the length of each sequence equals the depth of simulation. This is in contrast to MCTS where by default (and for reasons of efficiency, and of having enough visits to a node to make the statistics informative) the depth of tree is usually shallower than the total depth of the rollout (from root to the final evaluated state).

The use of action-sequences rather than trees as the core unit of evaluation has strengths and weaknesses. Strengths include simplicity and efficiency, but the main weakness is that the individual sequence-based approach would by default ignore the savings possible by recognising shared prefixes, though

---

[1]http://www.goratings.org

[2]http://gvgai.net
[3]https://en.wikipedia.org/wiki/Spacewar_(video_game)

prefix-trees can be constructed for evaluation purposes if it is more efficient to do so [9]. A further disadvantage is that the system is not directly utilising tree statistics to make more informed decisions, though given the limited simulation budget typically available in a real-time video game, the value of these statistics may be outweighed by the benefits of the rolling horizon approach.

Given the success of RHEA on single-player games, the question naturally arises of how to extend the algorithm to two-player games (or in the general case N-player games), with the follow-up question of how well such an extension works. The main contributions of this paper are the algorithm, RHCA, and the results of initial tests on our two-player battle game. The results are promising, and suggest that RHCA could be a natural addition to a game AI practitioner's toolbox.

The rest of this paper is structured as follows: Section II provides more background to the research, Section III describes the battle game used for the experiments, Section IV describes the controllers used in the experiments, Section V presents the results and Section VI concludes.

II. BACKGROUND

A. Open Loop control

Open loop control refers to those techniques which action decision mechanism is based on executing sequences of actions determined independently from the states visited during those sequences. Weber discusses the difference between closed and open loop in [10]. An open loop technique, Open Loop Expectimax Tree Search (OLETS), developed by Couëtoux, won the first GVGAI competition in 2014 [11], an algorithm inspired by Hierarchical Open Loop Optimistic Planning (HOLOP, [12]). Also in GVGAI, Perez et al. [9] discussed three different open loop techniques, then trained them on 28 games of the framework, and tested on 10 games.

The classic closed loop tree search is efficient when the game is deterministic, i.e., given a state s, $\forall a \in A(s)$ ($A(s)$ is the set of legal actions in s), the next state $s' \leftarrow S(a,s)$ is unique. In the stochastic case, given a state s, $\forall a \in A(s)$ ($A(s)$ is the set of legal actions in s), the state $s' \leftarrow S(a,s)$ is drawn with a probability distribution from the set of possible future states.

B. Rolling Horizon planning

Rolling Horizon planning, sometimes called Receding Horizon Control or Model Predictive Control (MPC), is commonly used in industries for making decisions in a dynamic stochastic environment. In a state s, an optimal input trajectory is made based on a forecast of the next t_h states, where t_h is the tactical horizon. Only the first input of the trajectory is applied to the problem. This procedure repeats periodically with updated observations and forecasts.

A rolling horizon version of an Evolutionary Algorithm that handles macro-actions was applied to the Physical Traveling Salesman Problem (PTSP) as introduced in [13]. Then, RHEA was firstly applied on general video game playing by Perez et

al. in [9] and was shown to be the most efficient evolutionary technique on the GVGAI Competition framework.

III. BATTLE GAMES

Our two-player space battle game could be viewed as derived from the original Spacewar (as mentioned above). The agents are given full information about the game state and make their actions simultaneously: the games are symmetric with perfect and incomplete information. Each game commences with the agents in random symmetric positions to provide varied conditions while maintaining fairness.

A. A simple battle game

First, a simple version without missiles is designed, referred as G_1. Each spaceship, either owned by the first (green) or second (blue) player has the following properties:

- has a maximal speed equals to 3 units distance per game tick;
- slows down over time;
- can make a clockwise or anticlockwise rotation, or to thrust at each game tick.

Thus, the agents are in a fair situation.

1) End condition: A player wins the game if it faces to the back of its opponent in a certain range before the total game ticks are used up. Figure 1 illustrates how the winning range is defined. If no one wins the game when the time is elapsed, it's a draw.

Fig. 1: The area in the bold black curves defines the "winning" range with $d_{min} = 100$ and $cos(\alpha/2) = 0.95$. This is a win for the green player.
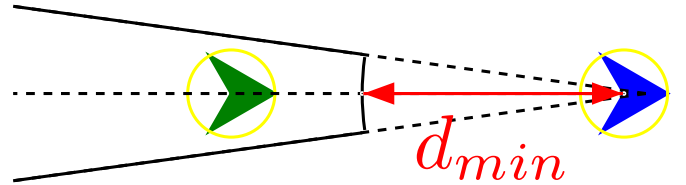
2) Game state evaluation: Given a game state s, if a ship i is located in the predefined winning range (Fig. 1), the player gets a score $DistScore_i = HIGH\_VALUE$ and the winner is set to i; otherwise, $DistScore_i = \frac{100}{dot+100}$ ($\in (0,1]$), where dot_i is the scalar product of the vector from the ship i's position to its opponent and the vector from its direction to its opponent's direction. The position and direction of both players are taken into account to direct the trajectory.

B. Perfect and incomplete information

The battle game has *perfect* information, because each agent knows all the events (e.g. position, direction, life, speed, etc. of all the objects on the map) that have previously occurred when making any decision; it has *incomplete* information because neither of agents knows the type or strategies of its opponent and moves are made simultaneously.

IV. CONTROLLERS

In this section, we summarize the different controllers used in this work. All controllers use the same heuristic to evaluate states (Section IV-A).

A. Search Heuristic

All controllers presented in this work follow the same heuristic in the experiments where a heuristic is used (Algorithm 1), aiming at guiding the search and evaluating game states found during the simulations. The end condition of the game is checked (detailed in Section III-A1) at every tick. When a game ends, a player may have won or lost the game or there is a draw. In the former two situations, a very high positive value or low negative value is assigned to the fitness respectively. A draw only happens at the end of last tick. Give a game state $s \in \S$,

$$Winner(s) = \begin{cases} 0, & \text{if the game finishes with a draw} \\ 1, & \text{if player 1 wins} \\ 2, & \text{if player 2 wins} \\ null, & \text{otherwise.} \end{cases}$$

Algorithm 1 Heuristic.

1: **function** EVALUATESTATE(State s, Player $p1$, Player $p2$)
2:     $s \leftarrow Update(p1, p2)$
3:     $winner = Winner(s)$
4:     **if** $winner == 1$ **then**   ▷ Player 1 wins
5:         $fitness_1 \leftarrow HIGH\_VALUE$
6:         $fitness_2 \leftarrow LOW\_VALUE$
7:     **else if** $winner == 2$ **then**   ▷ Player 2 wins
8:         $fitness_1 \leftarrow LOW\_VALUE$
9:         $fitness_2 \leftarrow HIGH\_VALUE$
10:     **else if** $winner = 0$ **then**   ▷ A draw
11:         $fitness_1 \leftarrow Score_1(s) - Score_2(s)$
12:         $fitness_2 \leftarrow Score_2(s) - Score_1(s)$
13:     **else**   ▷ Game not finished
14:         $fitness_1 \leftarrow Score_1(s) - Score_2(s)$
15:         $fitness_2 \leftarrow Score_2(s) - Score_1(s)$
16:     **end if**
17:     **return** $fitness_1$, $fitness_2$
18: **end function**

B. RHEA Controllers

In the experiments described later in Section V, two controllers implement a distinct version of rolling horizon planning: the Rolling Horizon Genetic Algorithm (RHGA) and Rolling Horizon Coevolutionary Algorithm (RHCA) controllers. These two controllers are defined next.

1) Rolling Horizon Genetic Algorithm (RHGA): The pseudo-code of RHGA is given in Algorithm 2. RHGA uses a tournament-based truncation selection with threshold with arbitrarily chosen threshold 20, i.e., the 20% best individuals will be selected as parents, and a population $\mathbf{x}$, where each individual in $\mathbf{x}$ represents some successive behaviours of

current player. The objective of $\mathbf{x}$ is to evolve better actions to kill the opponent. At each generation, the best 20% individuals in $\mathbf{x}$ are preserved as parents (elites), afterwards the rest are generated using the parents by uniform crossover and mutation.

Algorithm 2 Rolling Horizon Genetic Algorithm (RHGA)

Require: $\lambda \in \mathbb{N}^*$: population size, $\lambda > 2$
Require: $ProbaMut \in (0, 1)$: mutation probability
Require: $\alpha \in (0, 1)$: truncation threshold
1: $\mu = \lceil \alpha\lambda \rceil$ ▷ the size of elites
2: Randomly initialise population $\mathbf{x}$ with λ individuals
3: Randomly initialise opponent's individual $\mathbf{y}$
4: **for** $x \in \mathbf{x}$ **do**
5:     Evaluate the fitness of $\mathbf{x}$ and $\mathbf{y}$
6: **end for**
7: Sort $\mathbf{x}$ by decreasing fitness value order, so that

$$x_1.fitness \geq x_2.fitness \geq \cdots \geq x_\lambda.fitness$$

8: **while** time not elapsed **do**
9:     Randomly generate $\mathbf{y}$ ▷ Update opponent's individual
10:     $x'_1 \leftarrow x_1, \cdots, x'_\mu \leftarrow x_\mu$ ▷ Keep stronger individuals
11:     **for** $k \in \{\mu + 1, \ldots, \lambda\}$ **do**   ▷ Offspring
12:         Generate x'_k from $\{x'_1, \cdots, x'_\mu\}$ by uniform crossover
13:         Mutate x'_k with probability $ProbMut$
14:     **end for**
15:     $\mathbf{x} \leftarrow \mathbf{x}'$ ▷ Update population
16:     **for** $x \in \mathbf{x}$ **do**
17:         Evaluate the fitness of $\mathbf{x}$ and $\mathbf{y}$
18:     **end for**
19:     Sort $\mathbf{x}$ by decreasing fitness value order, so that

$$x_1.fitness \geq x_2.fitness \geq \cdots$$

20: **end while**
21: **return** x_1, the best individual in $\mathbf{x}$

---

2) Rolling Horizon Coevolutionary Algorithm (RHCA): RHCA (Algorithm 3) also uses a tournament-based *truncation selection* with threshold 20 and a population $\mathbf{x}$ to represent some successive behaviours of current player. Additionally, a population $\mathbf{y}$ is included, where each individual in $\mathbf{y}$ represents some successive behaviours of its opponent. The objective of $\mathbf{y}$ is to evolve stronger opponents, thus provide a worse situation to the current player. Similarly, at each generation, the best 20% individuals in $\mathbf{x}$ (respectively in $\mathbf{y}$) are preserved as parents (elites), afterwards the rest are generated using the parents by uniform crossover and mutation. Algorithm 4 is used to evaluate game state, given two populations, then sort both populations by average fitness value. Only a subset of the second population is involved.

3) Macro-actions and single actions: A macro-action is the repetition of the same action for t_o successive time steps. Different values of t_o are used during the experiments in this work in order to show how this parameter affects the

Algorithm 3 Rolling Horizon Coevolutionary Algorithm (RHCA)

Require: $\lambda \in \mathbb{N}^*$: population size, $\lambda > 2$
Require: $ProbaMut \in (0,1)$: mutation probability
Require: $SubPopSize$: the number of selected individuals from opponent's population
Require: EVALUATEANDSORT()
Require: $\alpha \in (0,1)$: truncation threshold
1: $\mu = \lceil \alpha\lambda \rceil$ ▷ the size of elites
2: Randomly initialise population $\mathbf{x}$ with λ individuals
3: Randomly initialise opponent's population $\mathbf{y}$ with λ individuals
4: $(\mathbf{x}, \mathbf{y}) \leftarrow$ EVALUATEANDSORT($\mathbf{x}, \mathbf{y}, SubPopSize$)
5: **while** time not elapsed **do**
6: $\quad y'_1 \leftarrow y_1, \cdots, y'_\mu \leftarrow y_\mu$ ▷ Keep stronger rivals
7: $\quad$ **for** $k \in \{\mu+1, \ldots, \lambda\}$ **do** ▷ Opponent's offspring
8: $\quad\quad$ Generate y'_k from $\{y'_1, \cdots, y'_\mu\}$ by uniform crossover
9: $\quad\quad$ Mutate y'_k with probability $ProbMut$
10: $\quad$ **end for**
11: $\quad \mathbf{y} \leftarrow \mathbf{y}'$ ▷ Update opponent's population
12: $\quad x'_1 \leftarrow x_1, \cdots, x'_\mu \leftarrow x_\mu$ ▷ Keep stronger individuals
13: $\quad$ **for** $k \in \{\mu+1, \ldots, \lambda\}$ **do** ▷ Offspring
14: $\quad\quad$ Generate x'_k from $\{x'_1, \cdots, x'_\mu\}$ by uniform crossover
15: $\quad\quad$ Mutate x'_k with probability $ProbMut$
16: $\quad$ **end for**
17: $\quad \mathbf{x} \leftarrow \mathbf{x}'$ ▷ Update population
18: $\quad EvaluateAndSort(\mathbf{x}, \mathbf{y}, SubPopSize)$
19: **end while**
20: **return** x_1, the best individual in $\mathbf{x}$

Algorithm 4 Evaluate fitness of population and subset of another population then sort the populations.

---
function EVALUATEANDSORT(population $\mathbf{x}$, population $\mathbf{y}$, $SubPopSize \in \mathbb{N}^*$)
$\quad$ **for** $x \in \mathbf{x}$ **do**
$\quad\quad$ **for** $i \in \{1, \ldots, SubPopSize\}$ **do**
$\quad\quad\quad$ Randomly select $y \in \mathbf{y}$
$\quad\quad\quad$ Evaluate the fitness of x and y once, update their average fitness value
$\quad\quad$ **end for**
$\quad$ **end for**
$\quad$ Sort $\mathbf{x}$ by decreasing average fitness value order, so that

$$x_1.averageFitness \geq x_2.averageFitness \cdots \geq x_\lambda.averageFitness$$

$\quad$ Sort $\mathbf{y}$ by decreasing average fitness value order, so that

$$y_1.averageFitness \geq y_2.averageFitness \cdots \geq y_\lambda.averageFitness$$

$\quad$ **return** $(\mathbf{x}, \mathbf{y})$
end function

OLMCTS version using a minimax (more max-N) tree policy and immediate future work is to evaluate such an agent.

Recommendation policy The recommended action in the next single time step is the one present in the most visited root child, i.e., robust child [1]. If more than one child ties as the most visited, the one with the highest average fitness value is recommended.

D. One-Step Lookahead

One-Step Lookahead algorithm (Algorithm 5) is deterministic. Given a game state s at timestep t and the sets of legal actions of both players $A(s)$ and $A'(s)$, One-Step Lookahead algorithm evaluates the game then outputs an action a_{t+1} for the next single timestep using some recommendation policy.

Algorithm 5 One-Step Lookahead algorithm.

Require: s: current game state
Require: $Score$: score function
Require: π: recommendation policy
1: Generate $A(s)$ the set of legal actions for player 1 in state s
2: Generate $A'(s)$ the set of legal actions for player 2 in state s
3: **for** $i \in \{1, \ldots, |A(s)|\}$ **do**
4: $\quad$ **for** $j \in \{1, \ldots, |A'(s)|\}$ **do**
5: $\quad\quad a_i \leftarrow i^{th}$ action in A
6: $\quad\quad a_j \leftarrow j^{th}$ action in A'
7: $\quad\quad M_{i,j} \leftarrow Score_1(s, a_i, a'_j)$
8: $\quad\quad M'_{i,j} \leftarrow Score_2(s, a_i, a'_j)$
9: $\quad$ **end for**
10: **end for**
11: $\tilde{a} \leftarrow \pi$, using M or M' or both
12: **return** $\tilde{a}$: recommended action

---

performance. The individuals in both algorithms have genomes with length equals to the number of future actions to be optimised, i.e., t_h.

In all games, different numbers of actions per macro-action are considered in *RHGA* and *RHCA*. The first results show that there is an improvement in performance, for all games, the shorter the macro-action is. The result using one action per macro-action, i.e., $t_o = 1$, will be presented.

Recommendation policy In both algorithms, the recommended trajectory is the individual with highest average fitness value in the population. The first action in the recommended trajectory, presented by the gene at position 1, is the recommended action in the next single time step.

C. Open Loop MCTS

A Monte Carlo Tree Search (MCTS) using Open Loop control (OLMCTS) is included in the experimental study. This was adapted from the OLMCTS sample controller included in the GVGAI distribution [14]. The OLMCTS controller was developed for single player games, and we adapted it for two player games by assuming a randomly acting opponent. Better performance should be expected of a proper two-player
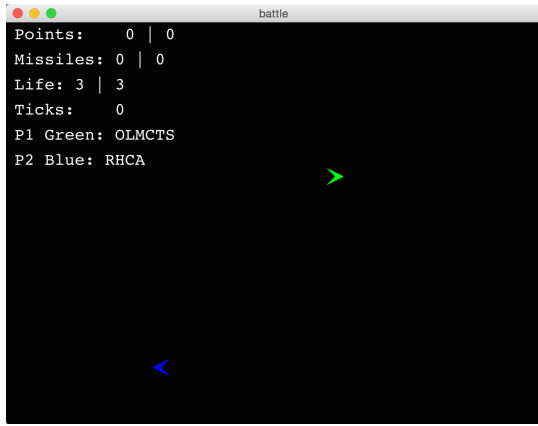
Fig. 2: At the beginning of every game, each spaceship is randomly initialised with a back-to-back position.

Recommendation policy There are various choices of π, such as Wald [15] and Savage [16] criteria. Wald consists in optimizing the worst case scenario, which means that we choose the best solution for the worst scenarios.

$$\tilde{a} = \underset{i \in \{1,...,|A(s)|\}}{\arg \max} \; \underset{j \in \{1,...,|A'(s)|\}}{\min} M_{i,j}. \quad (1)$$

Savage is an application of the Wald maximin model to the regret:

$$\tilde{a} = \underset{i \in \{1,...,|A(s)|\}}{\arg \min} \; \underset{j \in \{1,...,|A'(s)|\}}{\max} M'_{i,j}. \quad (2)$$

We also include a simple policy which chooses the action with maximal average score, i.e.,

$$\tilde{a} = \underset{i \in \{1,...,|A(s)|\}}{\arg \max} \sum_{j \in \{1,...,|A'(s)|\}} M_{i,j}. \quad (3)$$

Respectively,

$$\tilde{a} = \underset{i \in \{1,...,|A(s)|\}}{\arg \min} \sum_{j \in \{1,...,|A'(s)|\}} M'_{i,j}. \quad (4)$$

The *OneStep* controllers use separately (1) Wald (Equation 1) on its score; (2) maximal average score (Equation 3) policy; (3) Savage (Equation IV-D); (4) minimal the opponent's average score (Equation 4); (5) Wald on (its score - the opponent's score) and (6) maximal average (its score - the opponent's score). In the experiments described in Section V we only present the results obtained by using (6), which performs the best among the 6 policies.

V. EXPERIMENTS ON BATTLE GAMES

We compare a RHGA controller, a RHCA controller, an Open Loop MCTS controller and an One-Step Lookahead controller, denoted as *RHGA*, *RHCA*, *OLMCTS*, *OneStep* and a *move in circle* controller, denoted as *ROT*, by playing a two-player battle game of perfect and incomplete information.

TABLE I: Analysis of the number of wins of the row controller against the column controller in the battle games described in Section III-A. The last column shows the average #wins of the row controller, the higher the better; the last row show the average #loss of the column controller, the lower the better. $10ms$ is given at each game tick and $MaxTick = 2000$. Each game is repeated 100 times with random initialization. $RHCA$ outperforms all the controllers.

Test case 1: Simple battle game *without* weapon.						
	RND	ROT	OneStep	OLMCTS	$RHCA$	$RHGA$	Avg. wins
RND	-	62.5	51.0	9.0	6.5	7.5	27.3
ROT	37.5	-	0.0	0.0	0.0	0.0	7.5
OneStep	49.0	100.0	-	49.5	13.0	42.0	50.7
OLMCTS	91.0	100.0	50.5	-	50.0	50.0	68.3
$RHCA$	93.5	100.0	87.0	50.0	-	50.0	**76.1**
$RHGA$	92.5	100.0	58.0	50.0	50.0	-	70.1
Avg. loss	72.7	92.5	49.3	31.7	**23.9**	29.9	

A. Parameter setting

All controllers should decide an action within $10ms$. *OLMCTS* uses a maximum depth= 10. Both *RHCA* and *RHGA* have a population size $\lambda = 10$, $ProbaMut = 0.3$ and $t_h = 10$. The size of sub-population used in tournament ($SubPopSize$ in Algorithm 3) is set to 3. These parameters are arbitrarily chosen. Games are initialised with random positions of spaceships and opposite directions (cf. Figure 2).

B. Analysis of numerical results

We measure a controller by the number of wins and how quickly it achieves a win. Controllers are compared by playing games with each other using the full round-robin league. As the game is stochastic (due to the random initial positions of ships, the random process included in recoil or hitting), several games with random initialisations are played between any pair of the controllers.

For comparison, a random controller, denoted as *RND*, is also included in the experiments.

Table I illustrates the experimental results. Every entry in the table presents the number of wins of the column controller against the row controller among 100 trials of battle games. The number of wins is calculated as follows: for each trial of game, if it's a win of the column controller, the column controller accumulates 1 point, if it's a loss, the row controller accumulates 1 point; otherwise, it's a draw, both column and row controllers accumulates 0.5 point.

1) ROT: The rotated controller *ROT*, which goes in circle, is deterministic and vulnerable in simple battle game.

2) RND: Though the *RND* controller is not outstanding in the simple battle game, it still outperforms the deterministic *ROT*.

3) OneStep: *OneStep* performs poorly against all the other controllers except one case: against *ROT*. It's no surprise that *OneStep* beats *ROT*, a deterministic controller, in all the 100 trials. Among the 100 trials, *OneStep* is beaten separately by *OLMCTS* once and *RND* twice, the other trials finish by a draw.

4) OLMCTS: *OLMCTS* outperforms *ROT* and *RND*, however, there is no clear advantage or disadvantage when against *OneStep*, *RHCA* or *RHGA*.

5) RHCA: The fewer actions set in a macro-action, the better *RHCA* performs. *RHCA* outperforms all the controllers.

6) RHGA: The fewer actions set in a macro-action, the better *RHGA* performs. *RHGA* is the second-best in the battle game.

VI. Conclusions and further work

In this work, we design a new Rolling Horizon Coevolutionary Algorithm (RHCA) for decision making in two-player real-time video games. This algorithm is compared to a number of general algorithms on the simple battle game designed. In the game, more actions per macro-action lead to a worse performance of both Rolling Horizon Evolutionary Algorithms (controllers denoted as *RHGA* and *RHCA* in the experimental study). Rolling Horizon Coevolution Algorithm (*RHCA*) is found to perform the best or second-best.

More work on battle games with weapon is in progress. In the battle game, the sum of two players' fitness value remains 0. An interesting further work is to use a mixed strategy by computing Nash Equilibrium [17]. Furthermore, a Several-Step Lookahead controller is used to recommend the action at next tick, taking into account actions in the next n ticks. A One-Step Lookahead controller is a special case of Several-Step Lookahead controller with $n = 1$. As the computational time increases exponentially as a function of n, some adversarial bandit algorithms may be included to compute an approximate Nash [18] and it would be better to include some infinite armed bandit technique.

Finally, it is worth emphasizing that rolling horizon evolutionary algorithms provide an interesting alternative to MCTS that has been very much under-explored. In this paper we have taken some steps to redress this with initial developments of a rolling horizon coevolution algorithm. The algorithm described here is a first effort and while it shows significant promise, there are many obvious ways in which it can be improved, such as biasing the rollouts [19]. In fact, any of the techniques that can be used to improve MCTS rollouts can be used to improve RHCA. In the future, we are interested in improving our RHEAs and comparing them using some stronger opponents, such as an improved MCTS.

References

[1] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Computers and games*. Springer, 2006, pp. 72–83.

[2] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved Monte-Carlo Search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[3] S. Gelly, Y. Wang, O. Teytaud, M. U. Patterns, and P. Tao, "Modification of UCT with Patterns in Monte-Carlo Go," 2006.

[4] G. Chaslot, S. De Jong, J.-T. Saito, and J. Uiterwijk, "Monte-Carlo Tree Search in Production Management Problems," in *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*. Citeseer, 2006, pp. 91–98.

[5] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo Tree Search: A New Framework for Game AI," in *AIIDE*, 2008.

[6] "Mastering the game of Go with deep neural networks and tree search, author=Silver, David and Huang, Aja and Maddison, Chris J and Guez, Arthur and Sifre, Laurent and Van Den Driessche, George and Schrittwieser, Julian and Antonoglou, Ioannis and Panneershelvam, Veda and Lanctot, Marc and others, journal=Nature, volume=529, number=7587, pages=484–489, year=2016, publisher=Nature Publishing Group."

[7] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couetoux, J. Lee, C. U. Lim, and T. Thompson, "The 2014 General Video Game Playing Competition," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. PP, no. 99, pp. 1–1, 2015.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-Level Control Through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[9] D. Perez, J. Dieskau, M. Hünermund, S. Mostaghim, and S. Lucas, "Open Loop Search for General Video Game Playing," in *Proc. of the Conference on Genetic and Evolutionary Computation (GECCO)*, 2015.

[10] R. Weber, "Optimization and control," *University of Cambridge*, 2010.

[11] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 General Video Game Playing Competition," 2015.

[12] A. Weinstein and M. L. Littman, "Bandit-Based Planning and Learning in Continuous-Action Markov Decision Processes," in *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS, Brazil*, 2012.

[13] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling Horizon Evolution versus Tree Search for Navigation in Single-player Real-time Games," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 351–358.

[14] D. Perez, J. Dieskau, M. Hünermund, S. Mostaghim, and S. M. Lucas, "Open loop search for general video game playing," *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 337–344, 2015.

[15] A. Wald, "Contributions to the Theory of Statistical Estimation and Testing Hypotheses," *Ann. Math. Statist.*, vol. 10, no. 4, pp. 299–326, 12 1939.

[16] L. J. Savage, "The Theory of Statistical Decision," *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 55–67, 1951.

[17] M. J. Osborne and A. Rubinstein, *A course in Game Theory*. MIT press, 1994.

[18] J. Liu, "Portfolio Methods in Uncertain Contexts," Ph.D. dissertation, INRIA, 2015.

[19] S. M. Lucas, S. Samothrakis, and D. Perez, "Fast Evolutionary Adaptation for Monte Carlo Tree Search," in *Applications of Evolutionary Computation*. Springer, 2014, pp. 349–360.