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 three variations of 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 set 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 games 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 [7], an algorithm inspired by Hierarchical Open Loop Optimistic Planning (HOLOP, [11]). 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 [12]. Then, RHEA was firstly applied on general video game playing by Perez et al. in [9] and was proved 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), though we developed it from some Asteroids code we had, by removing the asteroids and adding another player-controlled ship. We then experimented with a number of variations to bring out some strengths and weaknesses of the various algorithms under test. A key finding is that the rankings of the algorithms depend very much on the details of the game; had we just tested on a single version of the game the conclusions would have been less robust and may have shown RHCA in a false light. 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 without weapons

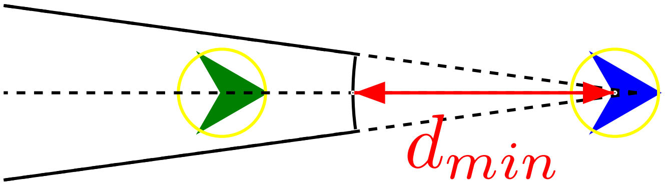
First, a simple version without missiles is designed, referred as G_1. Each spaceship, either owned by the first (green) or second (blue) player has the following properties:

- 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. *An upgraded battle game with* cooldown-needless *weapon*

We modify the previous battle game by providing a *cooldown-needless* weapon and an *infinity* weapon budget. Hence, a "fire a missile" action is added to the set of legal actions. A missile has

- an identical limited remaining time 100, which means it vanishes after 100 game ticks;
- a maximal speed equals to 4 units distance (4 pixels in the video game) per game tick.

When launching a missile, the spaceship is affected by a recoil; when being hit by any of the opponent's missiles, the spaceship loses one life and its direction and position are changed due to an explosion, where some random process are included. This new game is referred as G_2.

1) End condition: The game terminates immediately if one of the following game events occurs:

- if only one spaceship has $life = 0$ before taken the total ticks, it is a loss for the player with no lives left;
- if both spaceships have $life = 0$ before the total ticks have expired, it's a draw;
- if both spaceships survive when the total game ticks are used up, the one with more points wins;
- if both spaceships survive with the same number of points when the total game ticks are used up, it's a draw.

2) Game state evaluation: Every time a player hits its opponent, it obtains 10 points. Every time a player launches a missile, it is penalized by -1 points. The score function is modified, considering the current number of points. Given a game state s, a player has a $nbPoints$ calculated by:

$$nbPoints = 10 \times nb_k - nb_m \tag{1}$$

where nb_k is the number of lives subtracted from the opponent and nb_m indicates the number of launched missiles. The final score is a the sum of $nbPoints$ and $DistScore$ (as defined in Section III-A).

C. *An upgraded battle game with* cooldown-required *weapon*

Again, we modify the battle game by setting a $cooldown$ time to the weapon, i.e., after launching a missile, the weapon could not launch another during the next $cooldown$ game ticks. This version is referred as G_3. Both the end condition and the game state evaluation remain the same as in G_2.

D. *Perfect and incomplete information*

The battle game and its variants have *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; they have *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 Sections III-A1 and III-B1) 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. In the game without weapon (G_1), a draw only happens at the end of last tick. In the games with weapon (G_2 and G_3), a draw may happen earlier.

Algorithm 1 Heuristic.

```
function EvaluateState(State $s$, Player $p1$, Player $p2$)
    $fitness_1 \leftarrow 0$, $fitness_2 \leftarrow 0$
    $s \leftarrow Update(p1, p2)$
    if $Winner(s) == 1$ then
        $fitness_1 \leftarrow HIGH\_VALUE$
    else
        $fitness_1 \leftarrow LOW\_VALUE$
    end if
    if $Winner(s) == 2$ then
        $fitness_2 \leftarrow HIGH\_VALUE$
    else
        $fitness_2 \leftarrow LOW\_VALUE$
    end if
    if $Winner(s) == null$ then
        $fitness_1 \leftarrow Score_1(s) - Score_2(s)$
        $fitness_2 \leftarrow Score_2(s) - Score_1(s)$
    end if
    return $fitness_1$, $fitness_2$
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): In the experiments described later in Section V, this algorithm uses *truncation selection* with arbitrarily chosen threshold 20, i.e., the 20% best individuals will be selected as parents. The pseudo-code of this procedure is given in Algorithm 2.

2) Rolling Horizon Coevolutionary Algorithm (RHCA): RHCA (Algorithm 3) uses a tournament-based *truncation selection* with threshold 20 and two populations $\mathbf{x}$ and $\mathbf{y}$, where each individual in $\mathbf{x}$ represents some successive behaviours of current player and each individual in $\mathbf{y}$ represents some successive behaviours of its opponent. The objective of $\mathbf{x}$ is to evolve better actions to kill the opponent, whereas the objective of $\mathbf{y}$ is to evolve stronger opponents, thus provide a worse situation to the current player. At each generation, the best 2 individuals in $\mathbf{x}$ (respectively $\mathbf{y}$) are preserved

Algorithm 2 Rolling Horizon Genetic Algorithm (RHGA)

```
Require: $\lambda \in \mathbb{N}^*$: population size, $\lambda > 2$
Require: $ProbaMut \in (0, 1)$: mutation probability
 1: Randomly initialise population $\mathbf{x}$ with $\lambda$ individuals
 2: Randomly initialise opponent's individual $y$
 3: for $x \in \mathbf{x}$ do
 4:     Evaluate the fitness of $x$ and $y$
 5: end for
 6: Sort $\mathbf{x}$ by decreasing fitness value order, so that
        $x_1.fitness \geq x_2.fitness \geq \cdots$
 7: while time not elapsed do
 8:     Randomly generate $y$                 ▷ Update opponent's individual
 9:     $x'_1 \leftarrow x_1$, $x'_2 \leftarrow x_2$            ▷ Keep stronger individuals
10:     for $k \in \{3, \dots, \lambda\}$ do                     ▷ Offspring
11:         Generate $x'_k$ from $x'_1$ and $x'_2$ by uniform crossover
12:         Mutate $x'_k$ with probability $ProbMut$
13:     end for
14:     $\mathbf{x} \leftarrow \mathbf{x}'$                              ▷ Update population
15:     for $x \in \mathbf{x}$ do
16:         Evaluate the fitness of $x$ and $y$
17:     end for
18:     Sort $\mathbf{x}$ by decreasing fitness value order, so that
            $x_1.fitness \geq x_2.fitness \geq \cdots$
19: end while
20: return $x_1$, the best individual in $\mathbf{x}$
```

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()
 1: Randomly initialise population $\mathbf{x}$ with $\lambda$ individuals
 2: Randomly initialise opponent's population $\mathbf{y}$ with $\lambda$ individuals
 3: $(\mathbf{x}, \mathbf{y}) \leftarrow$ EVALUATEANDSORT$(\mathbf{x}, \mathbf{y}, SubPopSize)$
 4: while time not elapsed do
 5:     $y'_1 \leftarrow y_1$, $y'_2 \leftarrow y_2$               ▷ Keep stronger rivals
 6:     for $k \in \{3, \dots, \lambda\}$ do              ▷ Opponent's offspring
 7:         Generate $y'_k$ from $y'_1$ and $y'_2$ by uniform crossover
 8:         Mutate $y'_k$ with probability $ProbMut$
 9:     end for
10:     $\mathbf{y} \leftarrow \mathbf{y}'$                   ▷ Update opponent's population
11:     $x'_1 \leftarrow x_1$, $x'_2 \leftarrow x_2$            ▷ Keep stronger individuals
12:     for $k \in \{3, \dots, \lambda\}$ do                     ▷ Offspring
13:         Generate $x'_k$ from $x'_1$ and $x'_2$ by uniform crossover
14:         Mutate $x'_k$ with probability $ProbMut$
15:     end for
16:     $\mathbf{x} \leftarrow \mathbf{x}'$                              ▷ Update population
17:     $EvaluateAndSort(\mathbf{x}, \mathbf{y}, SubPopSize)$
18: end while
19: 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 $\mathbf{y}$, $SubPopSize \in \mathbb{N}^*$)
    for $x \in \mathbf{x}$ do
        for $i \in \{1, \dots, SubPopSize\}$ do
            Randomly select $y \in \mathbf{y}$
            Evaluate the fitness of $x$ and $y$ once, update their average fitness value
        end for
    end for
    Sort $\mathbf{x}$ by decreasing average fitness value order, so that
        $x_1.averageFitness \geq x_2.averageFitness \geq \cdots$
    Sort $\mathbf{y}$ by decreasing average fitness value order, so that
        $y_1.averageFitness \geq y_2.averageFitness \geq \cdots$
    return $(\mathbf{x}, \mathbf{y})$
end function
```

as parents (elites), afterwards the rest is 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 performance. The individuals in both algorithm have genomes with length equals to the number of future actions to be optimised, i.e., t_h.

In all the games, different numbers of actions per macro-action are considered in *RHGA* and *RHCA*, the primary results shows that, in all the games, the less actions compose a macro-action, the better the performance 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 (Algorithm 2 line 20; Algorithm 3 line 19). 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 [13], and was used for convenience. 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 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: $\pi$: recommendation policy
Generate $A(s)$ the set of legal actions for player 1 in state $s$
Generate $A'(s)$ the set of legal actions for player 2 in state $s$
for $i \in \{1, \dots, |A(s)|\}$ do
    for $j \in \{1, \dots, |A'(s)|\}$ do
        $a_i \leftarrow i^{th}$ action in $A$
        $a_j \leftarrow j^{th}$ action in $A'$
        $M_{i,j} \leftarrow Score_1(s, a_i, a'_j)$
        $M'_{i,j} \leftarrow Score_2(s, a_i, a'_j)$
    end for
end for
$\tilde{a} \leftarrow \pi$, using $M$ or $M'$ or both
return $\tilde{a}$: recommended action
```

Recommendation policy There are various choices of π, such as Wald [14] and Savage [15] criteria. Wald consists in optimizing the worst case scenario, which means that we choose the best solution for the worst scenarios. I.e., the recommended action for player 1 is

$$\tilde{a} = \underset{i \in \{1,\dots,|A(s)|\}}{\arg\max} \min_{j \in \{1,\dots,|A'(s)|\}} M_{i,j}. \tag{2}$$

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

$$\tilde{a} = \underset{i \in \{1,\dots,|A(s)|\}}{\arg\min} \max_{j \in \{1,\dots,|A'(s)|\}} M'_{i,j}. \tag{3}$$

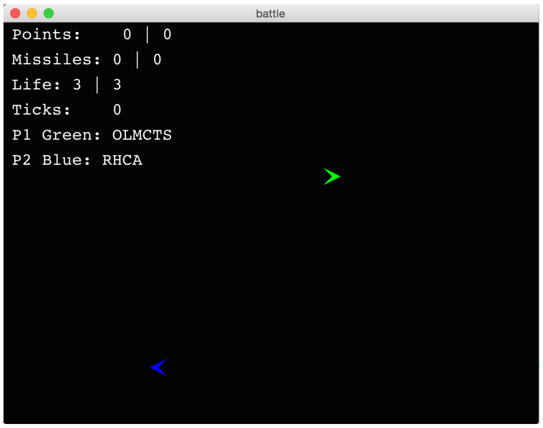
We also include a simple policy which chooses the action with maximal average score, i.e.,

$$\tilde{a} = \underset{i \in \{1,\dots,|A(s)|\}}{\arg\max} \sum_{j \in \{1,\dots,|A'(s)|\}} M_{i,j}. \tag{4}$$

Respectively,

$$\tilde{a} = \underset{i \in \{1,\dots,|A(s)|\}}{\arg\min} \sum_{j \in \{1,\dots,|A'(s)|\}} M'_{i,j}. \tag{5}$$

The *OneStep* controllers use separately (1) Wald (Equation 2) on its score; (2) maximal average score (Equation 4) policy; (3) Savage (Equation IV-D); (4) minimal the opponent's average score (Equation 5); (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.



```
battle
Points:    0 | 0
Missiles: 0 | 0
Life: 3 | 3
Ticks:    0
P1 Green: OLMCTS
P2 Blue: RHCA
```

Fig. 2: At the beginning of every game, each spaceship is randomly initialised with a back-to-back position.

V. Experiments on battle games

We compare a RHGA controller, a RHCA controller, an Open Loop MCTS controller and an One-Step Lookahead controller, a *move in circle* controller and a *rotate-and-shoot* controller, denoted as *RHGA*, *RHCA*, *OLMCTS*, *OneStep*, *ROT* and *RAS* respectively, by playing a two-player battle game of perfect and incomplete information, then the same controllers are tested on some upgraded battle games with weapons.

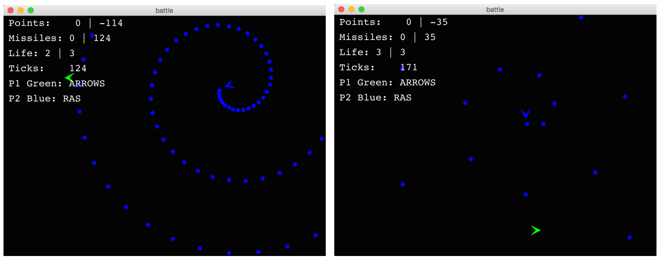
A. Parameter setting

All controllers should decide an action within $10ms$. In games with weapon, each controller is initialized with 3 lives. When a $cooldown$ time is required, it is set to 4 ticks. *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.

It's notable that the games with weapon (G_2 and G_3) are designed to kill the opponent as *soon* as possible with as *less missiles* as possible. If we only consider the number of wins, ignoring the cost of missiles, the best agent will be the one one



```
battle
Points:    0 | -114
Missiles: 0 | 124
Life: 2 | 3
Ticks:    124
P1 Green: ARROWS
P2 Blue: RAS
```

```
battle
Points:    0 | -35
Missiles: 0 | 35
Life: 3 | 3
Ticks:    171
P1 Green: ARROWS
P2 Blue: RAS
```

Fig. 3: *RAS* controller goes in circle and shoot. Left: *cooldown-needless*; right: *cooldown-required*.

that always goes round in circles and fires constantly (denoted as *RAS*, Figure 3). This is very important for understanding the experimental results.

When no weapon is included in the battle game (G_1), a rotated controller, denoted as *ROT*, is also compared. For comparison, a random controller, denoted as *RND*, is included in all the battle games.

Table I illustrates the experimental results[4]. Every entry in the table presents the number of wins of the column controller against the line 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 line controller accumulates 1 point; otherwise, it's a draw, both column and line controllers accumulates 0.5 point. If no averaged game ticks taken is given, it means that the column controller never wins; if no standard error is given, it means that the column controller only wins once.

1) ROT: The rotated controller *ROT*, which goes in circle, is deterministic and vulnerable in simple battle games without weapon (G_1).

2) RAS: The *RAS* controller dominates in games with *cooldown-needless* weapon (G_2) and outperforms most of the controllers in games with *cooldown-required* weapon (G_3). However, it's beaten by *RHCA* in G_3.

3) RND: The *RND* controller is not outstanding in the simple battle game without weapon (G_1). It's notable that its high winning rates in the games with weapon (G_2 and G_3) is foregone. The actions are uniform randomly chosen, without considering the cost of missiles. As a result, it shoots more frequently than the other controllers, except *RAS*. In most of the games terminated by a win of *RND*, *RND* has a much lower number of points than the other controllers (except *RAS*).

4) OneStep: OneStep is feeble in all games against all the other controllers except one case: against *ROT* in G_1. It's no surprise that *OneStep* beats *ROT*, a deterministic controller, in all the 100 trials of G_1. Among the 100 trials of G_1, *OneStep* is beaten separately by *OLMCTS* once and *RND* twice, the other trials finish by a draw. This explains the high standard error.

5) OLMCTS: OLMCTS outperforms *ROT* and *RND* in games without weapon (G_1), however, there is no clear advantage or disadvantage when against *OneStep*, *RHCA* or *RHGA*. The battle game is not hard enough to distinguish the strength of controllers. It is the main reason we upgrade the battle game by introducing weapon.

TABLE I: Analysis of the number of wins of the line controller against the column controller in the battle games described in Sections III-A (test case 1), III-B (test case 2) and III-C (test case 3). The bottom table presents the results for battle game with $cooldown$ required weapon, but the recoil is removed (test case 4). The more wins achieved, the better controller performs. $10ms$ is given at each game tick and $MaxTick = 2000$. Each game is repeated 100 times with random initialization. (i) $RHCA$ outperforms all the controllers in the test case 1. (ii) As no $cooldown$ is required, RAS dominates with a circular wall of bullets in the test case 2. (iii) When a $cooldown$ is required, $RHCA$ performs almost as well as RAS in the test case 3 (the difference in terms of winning rate is 1.2%). More discussion can be found in Sections V-B1-V-B7.

Test case 1: Simple battle game *without* weapon (G_1).

		RND	ROT	OneStep	OLMCTS	$RHCA$	$RHGA$
	Avg.	72.7	92.5	49.3	31.7	23.9	29.9
RND	27.3	-	62.5	51.0	9.0	6.5	7.5
RAS	7.5	37.5	-	0.0	0.0	0.0	0.0
OneStep	50.7	49.0	100.0	-	49.5	13.0	42.0
OLMCTS	68.3	91.0	100.0	50.5	-	50.0	50.0
$RHCA$	**76.1**	93.5	100.0	87.0	50.0	-	50.0
$RHGA$	70.1	92.5	100.0	58.0	50.0	50.0	-

Test case 2: Battle game with a $no - cooldown$ required weapon (G_2), considering $DistScore$ and $nbPoints$.

		RND	RAS	OneStep	OLMCTS	$RHCA$	$RHGA$
	Avg.	49.6	1.4	97.4	54.4	43.8	53.4
RND	50.4	-	0.0	89.0	61.0	44.0	58.0
RAS	**98.6**	100.0	-	100.0	99.0	96.0	98.0
OneStep	2.6	11.0	0.0	-	0.0	0.0	2.0
OLMCTS	45.6	39.0	1.0	100.0	-	38.0	50.0
$RHCA$	56.2	56.0	4.0	100.0	62.0	-	59.0
$RHGA$	46.6	42.0	2.0	98.0	50.0	41.0	-

Test case 3: Battle game with a $cooldown$ required weapon (G_3), considering $DistScore$ and $nbPoints$.

		RND	RAS	OneStep	OLMCTS	$RHCA$	$RHGA$
	Avg.	43.4	30.4	96.8	52.8	31.6	45.0
RND	56.6	-	17.0	86.0	73.0	43.0	64.0
RAS	**69.6**	83.0	-	99.0	70.0	26.0	70.0
OneStep	3.2	14.0	1.0	-	0.0	1.0	0.0
OLMCTS	47.2	27.0	30.0	100.0	-	40.0	39.0
$RHCA$	68.4	57.0	74.0	99.0	60.0	-	52.0
$RHGA$	55.0	36.0	30.0	100.0	61.0	48.0	-

6) RHCA: In all the games, the less actions set in a macro-action, the better *RHCA* performs. *RHCA* outperforms all the controllers in G_1, G_3, and is the second-best in G_2, where *RAS* without cooldown dominates.

7) RHGA: In all the games, the less actions set in a macro-action, the better *RHGA* performs. *RHGA* is the second-best in the battle game without weapon (G_1); it's beaten by *RHCA* when missiles are included; it performs sightly better than *OLMCTS* in G_2 and G_3.

C. What happens if the game is less stochastic?

In the games with weapon, the spaceship is affected by a recoil when launching a missile or an explosion when being hit by any of the opponent's missiles. We now remove the recoil and explosion in the battle game with *cooldown-required*

[4] A video of G3 can be found at https://youtu.be/X6SnZr10yB8.

TABLE II: Analysis of the number of wins of the line controller against the column controller in the battle games without recoil or explosion (test case 4, *deterministic*), as described in Section V-C. The more wins achieved, the better controller performs. $10ms$ is given at each game tick and $MaxTick = 2000$. Each game is repeated 100 times with random initialization. $RHCA$ performs almost as well as OLMCTS in the test case 4 (the difference in terms of winning rate is less than 1%).

Test case 4: Battle game with a $cooldown$ required weapon (G'_3), without *recoil*, considering $DistScore$ and $nbPoints$.

		RND	RAS	ONESTEP	OLMCTS	$RHCA$	$RHGA$
	Avg.	58.2	75.1	72.1	28.6	29.5	36.5
RND	41.8	-	27.0	52.0	57.0	31.0	42.0
RAS	24.9	73.0	-	32.5	1.0	3.0	15.0
ONESTEP	27.9	48.0	67.5	-	0.0	13.0	11.0
OLMCTS	**71.4**	43.0	99.0	100.0	-	64.5	50.5
$RHCA$	70.5	69.0	97.0	87.0	35.5	-	64.0
$RHGA$	63.5	58.0	85.0	89.0	49.5	36.0	-

weapon, thus make the game deterministic once the game begins, thus the initial positions of both ships are set. The new game is denoted as G'_3. The same experiments are performed on G'_3 and experimental results are analysed in Table II.

When no random process is included, *RAS* is weak against *OLMCTS*, *RHCA* and *RHGA*. *RHCA* is beaten by *OLMCTS*, however, it performs better than *OLMCTS* when playing against *RND* and *RHGA* controller.

D. What happens if the game state evaluation is modified?

Now we modify the way a given game state is evaluated and studied how the performances of previously used controllers vary in different games with weapon.

Instead of including the position, direction and number of points ($DistScore$ and $nbPoints$) when evaluating a given game state, we only consider $nbPoints$ in the following experiments on the games with weapon. Table III analyses the number of wins and shows how the performance of different controllers changes when the game is evaluated differently. However, the numerical results do not show the path of controllers, which are more or less meaningless and similar to a random controller[5].

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, and some more difficult variants to distinguish the strength of the compared algorithms. In all the games, 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). *RHCA* is found to perform the best or second-best in all the variants of games. It significantly outperforms the Open Loop MCTS in battle games with weapon, though immediate future work is to test a proper two-player version of Open Loop MCTS adapted for two-player adversarial simultaneous move games.

TABLE III: Analysis of the number of wins of the line controller against the column controllers in the battle games described in Sections III-B and III-C, taking into account only the number of points. The more wins achieved, the better controller performs. $10ms$ is given at each game tick and $MaxTick = 2000$. Each game is repeated 100 times with random initialization. (i) RAS dominates in both test cases. (ii) $RHEAs$ perform poorly due to the unsuitable heuristic, while OLMCTS achieves a winning rate around 50%.

Test case 5: Simple battle game with a $no-cooldown$ required weapon (G_2), considering $nbPoints$.

		RND	RAS	OneStep	OLMCTS	$RHCA$	$RHGA$
	Avg.	34.8	6.6	57.6	50.8	78.2	72.0
RND	65.2	-	0.0	85.0	85.0	74.0	82.0
RAS	**93.4**	100.0	-	67.0	100.0	100.0	100.0
OneStep	42.4	15.0	33.0	-	0.0	74.0	90.0
OLMCTS	49.2	15.0	0.0	100.0	-	73.5	57.5
$RHCA$	21.8	26.0	0.0	26.0	26.5	-	30.5
$RHGA$	28.0	18.0	0.0	10.0	42.5	69.5	-

Test case 6: Battle game with a $cooldown$ required weapon (G_3), considering $nbPoints$.

		RND	RAS	OneStep	OLMCTS	$RHCA$	$RHGA$
	Avg.	48.0	12.6	48.0	49.2	74.4	67.8
RND	52.0	-	8.0	67.0	67.0	57.0	61.0
RAS	**87.4**	92.0	-	47.0	100.0	99.0	99.0
OneStep	52.0	33.0	53.0	-	0.0	83.0	91.0
OLMCTS	50.8	33.0	0.0	100.0	-	65.5	55.5
$RHCA$	25.6	43.0	1.0	17.0	34.5	-	32.5
$RHGA$	32.2	39.0	1.0	9.0	44.5	67.5	-

Intransitivities were observed in the tournament (Section V-C). Intransitivities have been studied in [16]. Computing a Transitivity Index and KL-divergence were used to detect intransitivities. It will be interesting to use such techniques to analyse the extent to which transitivities are occurring in RHCA and take similar measure to cope with them.

In the battles games, 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, taken 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 roll-outs [19]. In fact, any of the techniques that can be used to improve MCTS rollouts can be used to improve RHCA.

[5] A video can be found at https://youtu.be/zLKO60YIKTY

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] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, and K. Kavukcuoglu, “Mastering the game of go with deep neural networks and tree search.”

[7] 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.

[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] 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.

[12] 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.

[13] 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.

[14] A. Wald, “Contributions to the theory of statistical estimation and testing hypotheses,” *Ann. Math. Statist.*, vol. 10, no. 4, pp. 299–326, 12 1939. [Online]. Available: http://dx.doi.org/10.1214/aoms/1177732144

[15] L. J. Savage, “The theory of statistical decision,” *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 55–67, 1951. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/01621459.1951.10500768

[16] S. Samothrakis, S. Lucas, T. Runarsson, and D. Robles, “Coevolving game-playing agents: Measuring performance and intransitivities,” *Evolutionary Computation, IEEE Transactions on*, vol. 17, no. 2, pp. 213–226, 2013.

[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.