The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement

Kamolwan Kunanusont, Raluca D. Gaina, Jialin Liu, Diego Perez-Liebana and Simon M. Lucas

University of Essex, Colchester, UK

Email: {kkunan, rdgain, jialin.liu, dperez, sml}@essex.ac.uk

Abstract—This paper describes a new evolutionary algorithm that is especially well suited to AI-Assisted Game Design. The approach adopted in this paper is to use observations of AI agents playing the game to estimate the game's quality. Some of best agents for this purpose are General Video Game AI agents, since they can be deployed directly on a new game without game-specific tuning; these agents tend to be based on stochastic algorithms which give robust but noisy results and tend to be expensive to run. This motivates the main contribution of the paper: the development of the novel N-Tuple Bandit Evolutionary Algorithm, where a model is used to estimate the fitness of unsampled points and a bandit approach is used to balance exploration and exploitation of the search space. Initial results on optimising a Space Battle game variant suggest that the algorithm offers far more robust results than the Random Mutation Hill Climber and a Biased Mutation variant, which are themselves known to offer competitive performance across a range of problems. Subjective observations are also given by human players on the nature of the evolved games, which indicate a preference towards games generated by the N-Tuple algorithm.

I. INTRODUCTION

Automatic game design algorithms are systems capable of designing proper and playable games with close to none human intervention. Designing a "playable" game usually involves tuning an appropriate set of game parameters. Manually doing this might be time-consuming due to large search space of game parameters. Evolutionary Algorithms (EAs) are therefore employed to evolve game parameters, one of the first attempts being that of Togelius and Schmidhuber [1]. Their results inspired the usage of EAs for game parameter tuning in later works. This paper presents the results of a project that was solely focused on AI informed game design using three different Evolutionary Algorithms. The main aim is to explore the possibility of using AI controllers with different skill levels as human-player representations to evolve suitable game parameter sets using the same fitness criteria. In this context, a suitable game parameter set should be reflected in a game that better distinguishes player skill levels.

To achieve this goal, a simple game *Space Battle* was chosen and redesigned to *Space Battle Evolved* by adding three new mechanics. A set of parameters of this game variant was selected (an interesting problem in itself) and evolved using a Random Mutation Hill Climbing algorithm [2], an improved version of RMHC called Biased-Mutation RMHC (B-RMHC) and a proposed noisy optimization N-Tuple Bandit Mutation. In order to reduce the dimensions of the game space, the rules

were kept fixed and only several parameters of the game were tuned, having all 3 Evolutionary Algorithms generate unique variants of *Space Battle Evolved*. Several runs of automatic play testing were also carried out to ensure fine-tuned results, as well as human play testing to assess subjective game quality.

This paper is organised as follows: Section II briefly reviews the related work on automatic game design. Section III describes the game and AI controllers used in this paper. Section IV introduces our approach of evolving game instances using three Evolutionary Algorithms. The experimental results are presented in Section V and Section VI concludes the paper.

II. LITERATURE REVIEW

This section contains a brief review of several materials consulted as part of this research work.

Automatic game design is a sub-field of Game Artificial Intelligence that explores the idea of developing a system capable of generating dynamic and playable games. One of the first attempts at game design using such a system was developed by Togelius and Schmidhuber [1]. The benefits of the work they pioneered include the possibility of creating multiple new and unique games automatically by making use of advanced computation methods and speed of execution. Their results suggested that evolutionary algorithms can indeed be used to automatically search a space of possible games.

Nelson and Mateas [3] used a generative process in their paper, which refers to factoring a game design process into four interacting domains: abstract game mechanics, game representation, thematic content and control mapping. The game design space, which is the space of all possible games that the resultant system can reason about, is defined by the specific knowledge given for each of these domains.

The basic methodology of creating a generative system was employed by Isaksen et al. [4], [5], [6]. They explored the possibilities of discovering useful variants of games by tuning aspects of the game space and analysing the resulting player experience. Their paper is focused on the possibilities of achieving this effect by varying game parameters without changing the game rules and how this process could yield games of varying difficulties.

Another application is the Physical Traveling Salesman Problem map evolution. Perez et al. [7] use three AI players of different skill levels to evaluate the maps produced by their algorithm, the hypothesis described in the paper being based on the fact that the players rankings would be kept consistent

in a good map; the higher the skill depth, the better the map. A similar approach was used in the experiments carried out for this present paper, with a fitness calculation aiming to distinguish the between the skill levels of three AI players.

Additionally, research has looked at automated maze generation for Ms Pac-Man. Safak et al. [8] use genetic algorithms to this end, using as a measure of fitness the ability of a player to finish the game in the newly created map. Their results show that Evolutionary Algorithms can be used to generate interesting mazes quite different from the original design, offering the players new challenging experiences.

A detailed account of the all game design aspects through search-based Procedural Content Generation is given by Togelius et al. in [9]. One thing highlighted in this survey is the importance of efficiently encoding the search space, not only for the EA to be able to process it correctly, but also for the evaluation function to analyse it effectively. To this extent, an emphasis is put on the necessity of search space constraints and a clear encoding in [10], which in turn would result in the same genotype being easily adapted to different phenotypes in other applications by simply varying the mapping functions. Therefore the work in the present paper focuses on describing the game space in terms of a set of interesting parameters with limited value ranges, adapted for fitness evaluations by inserting the generated values into games to be analysed.

Nielsen et al. [11] expand the automated design process to general video games and game rules, using the Video Game Description Language. A similar fitness measure as the one employed in our study is used, therefore comparing bad players against good players and using the difference in their performance to quantify the quality of the game. The authors analyse the differences in human-designed games, mutated variants of these games and newly generated games, reporting mixed results, with interesting but hardly playable games resulting from their study. They finally recommend evolutionary algorithms to be used only for idea generation and improvement, instead of actually creating complete games.

Menezes et al. [12] present their initial conceptual model incorporating a novel approach to generating engaging and surprising game worlds, based on the theory that complex things may emerge from simple interactions. They focus on co-evolution as the core of a Complex Adaptive System and encourage less human involvement in the evolutionary process, instead making use of auto-organizative systems.

III. BACKGROUND

This sections offers a brief description of the games and AI controllers used in the experiments.

A. Space Battle game

The original Space Battle game is a 2-player competitive game wherein players pilot ships and aim to shoot their opponent. The ships are in convex-quadrilateral shapes, as shown in Figure 1, with the front being indicated by a single acute angle point. The first player controls the blue ship while
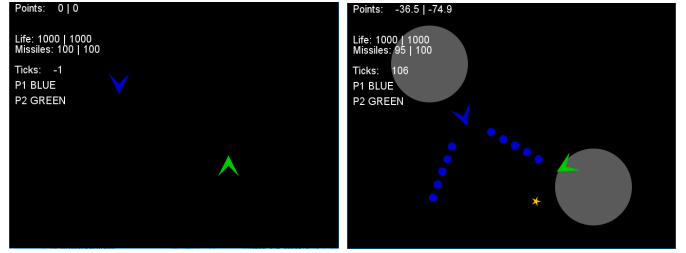

Fig. 1: Space Battle (left) and Space Battle Evolved (right)

the second players ship is green. The starting positions of the player ships are as depicted in Figure 1 (left).

Both players act simultaneously and have the same range of actions available, these being: turn clockwise, turn anticlockwise, thrust (move forward) and shoot a missile. Rotation, acceleration and shooting actions can be performed together in one game tick. Turn actions simply rotate the ship into the corresponding directions, without moving it from its current position. Therefore, the ship can only move forward when the thrust action is operated. When a player chooses to shoot, a round-shaped missile appears at the players ship location and moves into the ship's forward direction with a specific velocity.

Each player has 1000 lives, which decrease by 1 if their ship is hit by one of the opponents missiles. They each start with only 100 missiles. The game ends when one of two constraints is met: one players lives dropping to zero or the game-end being reached after a set amount of game ticks (2000 by default). At the end of the game, the player with most points is declared the winner.

The framework uses the same interface as the Two-Player General Video Game AI (GVGAI) framework [13], [14], making it easy to plug in agents submitted to the GVGAI competition and use them for game testing in this new problem. All AI controllers have access to a Forward Model (FM), which allows the agents to simulate possible future game states by providing an in-game action. Additionally, as the game is real-time, the agents only have 40ms for making a decision during a game step, with 1s for initialization.

This game appears in recent literature as an interesting challenge for AI agents. Liu et al. [15] analyse a novel co-evolution approach to two player games and test it on a simplified Space Battle game, in which the shooting action is removed and instead players score by positioning behind the enemy ship. The game states are evaluated by calculating the distance the players are from the ideal position.

B. AI controllers

Six different game playing agents were used in this experiment for game testing purposes. These are separated into two sets: first, the enemy player set, containing all agents mentioned in this section. The enemy player is an evolvable parameter (see Section IV-B), therefore the EA has access to all possible enemies when evolving the game.

In order to analyse the skill-depth of games produced by the Evolutionary Algorithm, a second set is used, containing

only three players: One Step Look Ahead (1SLA), which delegates a non-skill player, Rotate and Shoot (RAS), portraying an intermediate-skill player and Monte Carlo Tree Search (MCTS), representing a skillful player.

1) Do Nothing: Do Nothing is a dummy controller that returns no action in every game tick.

2) Random: The random controller returns a randomly chosen action in every time step, out of all those available to it. This agent can prove to be challenging to beat due to its unpredictable nature.

3) One Step Look Ahead (1SLA): This is the most simple AI algorithm which accesses the Forward Model. One Step Look Ahead (1SLA) simply simulates the next state for all available actions and chooses for execution the most promising action. The agent uses a simple heuristic to quantify the value of a game state, by considering the game score and, in cases of end-game states, whether it won or lost (receiving either a large bonus or a large penalty, respectively). The general behaviour of this agent observed in the game used for this experiment is moving randomly without shooting, unless the missile is due to hit the other player in the next game tick. It was considered a low skilled player for this study because it is fairly easy to beat, as long as the other player keeps its distance and avoids being right in the missile spawn point.

4) Rotate and Shoot (RAS): As the name suggests, this controller returns a combination of actions at each time step, clockwise turning and shooting. This strategy has proven to be unbeatable in a game which allows unlimited or a large number of missiles. In the game used in this paper, the players only have 100 missiles available in the beginning. Therefore, in theory, RAS should not perform well after the first 100 time steps. It was chosen as a mid-skilled AI player in this experiment, as it has a potential to survive and shoot the enemy at least in at the beginning of the game.

5) Monte Carlo Tree Search (MCTS): Monte Carlo Tree Search (MCTS) is a well-known tree search algorithm for game AI controllers. The main strength of MCTS is its ability to deal with huge search space by balancing between known and non-explored states using UCB1 equation (See Equation 1). For more information on this technique, the reader is referred to [16]. MCTS was chosen to represent a skillful player because its behaviour in the game was mostly unpredictable, unlike 1SLA and RAS, and able to perform well for the whole duration of the game.

$$a^* = \underset{a \in A(s)}{\arg\max} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \qquad (1)$$

6) Microbial Evolutionary Algorithm (MEA): This controller picks the first action of the best action plan generated by a Evolutionary Algorithm. It starts with a population of random individuals encoded as sequences of in-game actions. It then selects individuals through a microbial tournament, from which an offspring is generated via crossover. The newly generated individual is mutated, its fitness evaluated (using the same heuristic as 1SLA on the game state reached after playing the actions in one individual, in order) and the best individuals are carried forward to the next generation. This process is reiterated until the budget limit is reached (e.g. time, memory or specific number of iterations).

C. Random Mutation Hill Climber (RMHC)

A Random Mutation Hill Climber is the simplest version of an evolutionary algorithm, with only one individual in the population. It starts by randomly assigning values to each gene of the individual. One gene is selected for mutation uniformly at random, the fitness value of the resulting individual is calculated (similarly to the MEA evaluation) and compared with the previous one. The better individual is kept for the next iteration of the algorithm, repeated while allowed by the budget offered. In the implementation used for this paper, both the parent and the offspring are evaluated in each generation.

This algorithm is often used as a game playing agent in literature due to the great results produced while keeping simplicity. Buzdalov et al. [17] use an RMHC algorithm combined with Q-Learning for adaptive behaviour and analyse its runtime complexity on a modified OneMax problem (with an obstructive fitness function meant to lead the algorithm in the wrong direction), reporting good results. Liu et al. [2] explore modifications of the vanilla RMHC method, by using Upper Confidence Bounds (UCB) to guide evolution.

IV. Approach

Three EA algorithms were used to evolve *Space Battle Evolved* game parameters, using a fitness function meant to distinguish between good and bad players and optimize games in order to maximise the skill depth (See Section IV-F).

For each algorithm, 50 trials were experimented, due to the noisy game environment, as the same parameter set might return different fitness values in different runs. Each iteration of the evaluation process was carried out for 100 evaluations. The final games were analysed statistically by their fitness, as well as tested by a number of human players who offered their subjective opinions.

A. Space Battle Evolved

Space Battle Evolved (See Figure 1 for an example) is a variation of the simple Space Battle game that was designed for this project. Following on from the original rules of Space Battle, there are three main changes made to produce this variant.

Firstly, black holes were created, which have a set range and add forces to nearby objects in order to drag them towards their center. The players receive a penalty for each game tick in which they remain within a certain distance to a black hole center. There are, however, areas inside the black holes where no penalty is applied, called safe zones.

Secondly, two additional types of missiles were included in this version: a twin shot type, which fires two normal missiles at 45 and -45 degree angles from its direction and a bomb type which explodes in a large radius after a set time or upon collision with another object.

Lastly, due to the limited missiles available, the players have now at their disposal collectible packs of 20 missiles, which spawn on the map in a random position, disappear after a specified time (or after being collected by a player) and re-spawn again after a certain amount of time.

B. Evolvable game parameters

There were 30 evolvable game parameters in total, as summarized in Table I. These can be divided into 4 categories: missiles, black holes, resources and enemy.

1) Missile related: 6 of the parameters refer to missiles, including the type of the missile, its maximum speed, its cooldown (how many game ticks until the player is allowed to shoot a new missile), its radius, its time to live and, finally, the bomb explosion radius (for bomb type missile only). As the primary way of obtaining a score advantage over the opponent (and thus possibly ultimately winning the game), these could be considered key parameters.

2) Black hole related: 21 of the game parameters involve black holes, 17 relating to black hole locations and the rest specifying black hole characteristics. We divided the game map into a grid and allowed the evolutionary algorithms to decide whether to include a black hole in the center of each grid cell or not. The grid size varies between 1 and 4 (with a step of 1), therefore there could be up to 16 (4×4) black holes in a game. The black holes layout depict the main environment of the game. Other parameters needed for black hole mechanics include radius, force, penalty score (negative score given when the player is inside the black hole) and the safe-zone radius (non-penalization area around the inside border of the black hole).

3) Resource related: This game object refers to a pack of 20 missiles which eventually spawn on random positions on the map for collection. 2 resource-related parameters are time to live (number of game ticks before the resource disappears) and cooldown (number of game ticks for the resource to be re-spawned). Due to the limited missiles in *Space Battle Evolved*, the resources dictate whether the players may gain more than the maximum possible 10000 points which can be achieved with the initial missile budget.

4) Enemy related: The algorithms presented in this paper evolve the AI enemy as part of the game parameters, a novel aspect which greatly impacts the resulting gameplay. This parameter can refer to either of the AI controllers mentioned in Section III-B.

C. Baseline algorithm

We applied Random Mutation Hill Climber as a baseline algorithm. The algorithm uses an array of 30 parameters for evolution, initializing each one to a random value. One parameter is then chosen uniformly at random and mutated (1 random gene being changed to a different value). The fitness value of this mutated game is calculated by playing three games with the three AI controllers of different skill levels and following the method in Section IV-F. If the mutated game ends up with a higher fitness than its parent,

---

Algorithm 1 Random Mutation Hill Climber (RMHC)

```
 1: Input: game parameter list params, number of trials ntrials,
 2:        number of evaluations nEvals
 3: Output: evolved parameter sets
 4: BEGIN
 5:     paramList ← ∅
 6:     REPEAT ntrials times
 7:         params ← randomly assign each value
 8:         bestSoFar ← fitness(params)
 9:         REPEAT nEvals times
10:             p ← randomly select one parameter
11:             mutatedParams ← mutate(params,p)
12:             newFitness ← fitness(mutatedParams)
13:             IF newFitness ≥ bestSoFar
14:                 params ← mutatedParams
15:                 bestSoFar ← newFitness
16:         add params to paramList
17:     RETURN paramList
18: END
```

the offspring is the individual carried forward to the next generation. RMHC algorithm implementation details can be observed in Algorithm 1.

D. Biased Mutation RMHC

Biased Mutation RMHC (B-RMHC) was inspired by the idea that different parameters affect the change in fitness values at different rates. That is, modifying one parameter might significantly affect the fitness value more than others. Therefore, a biased mutation towards more interesting parameters was used to obtain more diverse games and speed up evolution. The algorithm is made up of two parts: parameter pre-processing and actual evolution.

1) Pre-processing: The parameters were divided into two groups, separating the black hole cells (Group B) from the rest (Group A). For Group A's pre-processing, the parameters received random values to start with. Then, for each parameter, the *importance* metric was calculated by using the standard deviation from the fitness in N tests, where N is the total number of values the parameter tested can take. For each value, the game taking the new parameter list was evaluated using the same fitness function employed during evolution. This assessment is based on the assumption that larger differences in fitness lead to larger standard deviation values.

For Group B, the parameters were analyzed separately for each possible grid size value, starting from all the black hole cells being empty and evaluating the effect of enabling a black hole in each cell. Similarly to Group A, the standard deviation of the fitness values resulted from each cell's evaluation was used to rank these parameters.

Pre-processing step outputs two list of parameters sorted by how much they affect the game fitness value. Details of this algorithm can be seen in Algorithm 2. This ordering was then used in the evolution step.

2) Evolution: A softmax function was employed to do a biased parameter selection at the beginning of the evolution process. This ensures that more important parameters are more likely to be selected. After that, the algorithm follows the

TABLE I: Evolvable parameters, their value ranges and step.

Parameter	Value Range	Step	Parameter	Value Range	Step	Parameter	Value Range	Step
MISSILE_MAX_SPEED	1 - 10	1	BLACKHOLE_CELL x 16	0 or 1	1	BOMB_RADIUS	10 - 50	10
MISSILE_COOLDOWN	1 - 9	1	BLACKHOLE_RADIUS	25 - 200	25	MISSILE_TYPE	0 - 2	1
MISSILE_RADIUS	2 - 10	2	BLACKHOLE_FORCE	0 - 3	1	RESOURCE_TTL	400 - 600	100
MISSILE_MAX_TTL	40-160	20	BLACKHOLE_PENALTY	0 - 9	1	RESOURCE_COOLDOWN	200 - 300	50
GRID_SIZE	1 - 4	1	SAFE_ZONE	0 - 20	10	ENEMY_ID	0 - 5	1

Algorithm 2 Biased Mutation Pre-processing (MutPrep)

1: Input: game parameter list *params*
2: Output: sorted lists of important parameters
3: BEGIN
4:     *PriorityQParams* ← ∅
5:     *PriorityQBH* ← ∅
6:     *ParamsN* ← *GroupA* parameters
7:     FOR EACH *p* in *paramsN*
8:         *value* ← ∅
9:         *rand*← randomly assign other parameter values
10:        FOR EACH possible value *v* of *p*
11:            *rand[p]* ← *v*
12:            add *fitness(rand)* to *value*
13:        *PriorityQParams[p]* ← *SD(value)*
14:    *rand*← randomly assign other parameter values
15:    *rand*← disable all black holes
16:    FOR *gSize* ∈ {0, 1, 2, 3, 4}
17:        *bhpriorityQ* ← ∅
18:        *fitnessOff* ←*fitness(rand)*
19:        FOR *b* = 1 to *gSize*^{2}
20:            enable black hole at position *b* in *rand*
21:            *bhpriorityQ[b]* ← *fitnessOff* − *fitness(rand)*
22:        *PriorityQBH[gSize]* ← *bhpriorityQ*
23:    RETURN *PriorityQParams*, *PriorityQBH*
24: END

Algorithm 3 N-Tuple Bandit Mutation (NTuple)

1: Input: game parameter list *params*,
2:     number of evaluations *nEvals*
3: Output: the best parameter set
4: BEGIN
5:     *current* ← randomly assign each value
6:     *LModel* ← ∅
7:     REPEAT *nEvals* times
8:         *value* ← *fitness(current)*
9:         add <*current*, *value*> to *LModel*
10:        *neighbors* ← generate neighbours from *LModel*
11:        *current* ← *n* in *neighbors* with *Max(UCB(n))*
12:    RETURN *n* in *LModel* with highest average value
13: END

---

design of simple RMHC and the fitness is evolved for a fixed number of iterations.

E. N-Tuple Bandit Evolutionary Algorithm

The N-Tuple Bandit Evolutionary Algorithm is an algorithm we developed to be particularly useful for evolving game designs and game parameters, especially when using agent-based evaluation methods. The evaluation function used in this paper will be noisy if the game is played by stochastic agents, such as MCTS, and fairly expensive in CPU time to run each game. Hence, it is desirable to have an evolutionary algorithm that is able to operate very efficiently, making the best possible use of the available fitness evaluation budget, and also one that is robust to noise. The N-Tuple Bandit EA satisfies these criteria.

1) Algorithm: The algorithm operates as follows. It begins by choosing a random point in the search space, which is called the current point. It then makes a noisy fitness evaluation and stores it in the N-Tuple Fitness Landscape Model as the value for that point. Using a mutation operator to generate a set of unique neighbours of the current point, and using the fitness landscape model, the algorithm gets the estimated Upper Confidence Bound (UCB) value of each point (see Equation 1). Finally, it sets the current point as the neighbour from the previous step with the highest UCB value.

These steps are displayed in Algorithm 3. When the fitness evaluation budget has been exhausted, the method searches a set of neighbours of all of the evaluated points and returns the one with the highest mean value ($Q(s, a)$).

2) N-Tuple Fitness Landscape Model: N-Tuple systems have ideal properties for use as fitness landscape models, in that they offer super-fast one-shot training and good accuracy. While their use for optical character recognition dates back to the 1950s, Lucas [18] introduced their use for game position evaluation functions. The concept is as follows. Given an *D*-dimensional search space, we sub-sample its dimensions with a number of *N*-tuples. The value of *N* ranges from 1 up to *D*, though may miss out values in between. The results in this paper are based on using *D* 1-tuples and 1 *D*-tuple.

Each N-Tuple has a look-up table (LUT) that stores statistical summaries of the values it encounters; the basic numbers stored are the number of samples, the sum of the samples, and the sum of the square of the samples. This enables the mean, the standard deviation and the standard error to be calculated for each entry in the table. More details can be found in [18].

F. Fitness evaluation

The fitness value of each game was evaluated with 3 gameplays, by using 1SLA, RAS and MCTS as players. For each game played, both of the players scores were divided by 100 to lower the scale, then a 1000 bonus points were awarded to the winner to prioritize winning result in producing the final score. The difference in the final score between player 1 and player 2 was assigned as the fitness of one game. Equation 2 shows the final score calculation for each gameplay. $W_k = 1000$ if the player *k* won the game and 0 otherwise.

$$T_g = (\frac{S_1}{100} + W_1) - (\frac{S_2}{100} + W_2) \tag{2}$$

After the total score T_g for every game g is computed, it was brought into the final fitness calculation as depicted in Equation 3, where T_1 is the weak player's game fitness (1SLA), T_2 is the mediocre player's game fitness (RAS) and T_3 is the strong player's game fitness (MCTS).

$$Fitness = Min(T_3 - T_2, T_2 - T_1) \qquad (3)$$

Equation 3 is similar to that used for the Physical Traveling Salesman Problem by Perez et. al. [7]. Based on this fitness evaluation, the aim of the algorithms is to maximize the smallest gap between final scores of each game in the order $T_3 > T_2 > T_1$, which would result in the maximum skill-depth.

V. EXPERIMENTAL RESULTS

We apply the RMHC, the Biased Mutation RMHC (denoted as B-RMHC) and the N-Tuple Bandit Mutation algorithm (denoted as N-Tuple) independently 50 times to evolve game instances, thus 150 games are designed in total. 100 game evaluations are allocated to each of the algorithms during the evolution. Then we pick up some of the evolved game instances for human players to test and analyse their feedback.

A. Selection of designed games by reevaluation

To select the game instance for human testing, each of the evolved game instances is then evaluated 100 times, where each evaluation takes into account the outcomes of three games played by the 1SLA, RAS and MCTS controllers (detailed in Section IV-F). The sorted average fitness values over 100 evaluations and standard errors are presented in Figure 2.

The N-Tuple algorithm (green markers) outperforms both the RMHC and its variant. Moreover, N-Tuple is more robust and has a more stable performance (negligible standard error). Among 50 game instances evolved by the N-Tuple Bandit Mutation algorithm, only a few of them (very left part in Figure 2) have an average fitness below zero. Nevertheless, the lowest average fitness is still much higher than most of the games evolved by both RMHC and B-RMHC.

A two-tailed Mann-Whitney U test shows that the results are significant when comparing the worst average games of each algorithm ($p \ll 0.0001$ for N-Tuple over RMHC and B-RMHC). The differences between RMHC and B-RMHC are not statistically significant ($p = 0.5774$). If taking into account all 50 trials, then N-Tuple remains significantly better than the other two algorithms ($p \ll 0.0001$). However, B-RMHC is not significantly better than RMHC ($p = 0.6080$).

Table II provides the parameters of the games, optimized by the RMHC, the B-RMHC and the N-Tuple algorithm, with the highest and lowest average fitness.

B. Evaluation by human players

We picked up the games with the highest and lowest average fitness designed by the 3 algorithms and invited two human players to evaluate them. The human players were asked to play the 6 games and provide feedback without being told the fitness level of each game. One screenshot of each of the games is presented in Figure 3, as well as the feedback from

TABLE II: Optimized parameters of game instances with the highest or lowest average fitness, designed by three algorithms.

| Parameter | Value optimised by different algorithms | | | | | |
| | RMHC | | B-RMHC | | N-Tuple | |
	High	Low	High	Low	High	Low
MISSILE_MAX_SPEED	6	1	10	1	9	10
MISSILE_COOLDOWN	9	5	5	3	2	5
MISSILE_RADIUS	2	10	10	4	4	4
MISSILE_MAX_TTL	140	60	40	80	40	140
GRID_SIZE	4	3	1	1	3	1
BLACKHOLE_CELL(1,1)	0	1	0	1	1	1
BLACKHOLE_CELL(1,2)	0	0	1	1	0	1
BLACKHOLE_CELL(1,3)	0	1	1	0	0	1
BLACKHOLE_CELL(1,4)	1	0	0	0	0	0
BLACKHOLE_CELL(2,1)	0	0	1	1	0	1
BLACKHOLE_CELL(2,2)	1	1	0	1	0	0
BLACKHOLE_CELL(2,3)	1	0	1	0	0	1
BLACKHOLE_CELL(2,4)	1	1	1	0	1	0
BLACKHOLE_CELL(3,1)	1	1	1	0	0	0
BLACKHOLE_CELL(3,2)	1	0	1	0	1	1
BLACKHOLE_CELL(3,3)	1	0	0	0	1	1
BLACKHOLE_CELL(3,4)	1	1	0	0	1	1
BLACKHOLE_CELL(4,1)	1	0	0	1	0	0
BLACKHOLE_CELL(4,1)	1	0	0	0	0	1
BLACKHOLE_CELL(4,3)	1	0	0	1	0	1
BLACKHOLE_CELL(4,4)	0	1	0	0	1	1
BLACKHOLE_RADIUS	200	75	100	100	150	25
BLACKHOLE_FORCE	2	1	3	3	3	1
BLACKHOLE_PENALTY	3	4	0	7	7	8
SAFE_ZONE	20	0	20	20	10	10
BOMB_RADIUS	10	50	20	40	20	20
MISSILE_TYPE	2	1	2	0	2	0
RESOURCE_TTL	400	500	500	500	400	500
RESOURCE_COOLDOWN	200	250	250	200	200	200
ENEMY_ID	0	2	1	0	0	5

both players. The two human players evaluated the games differently, according to their playing preference. Player A cares more about the challenging aspect of the game and is attracted more towards uncommon game scenarios; Player B is less easily satisfied and found most of the games boring. Interestingly, though they have ranked the games differently, they both have a preference for the game $G3_H$ (with the highest average fitness value, optimised by N-Tuple) and dislike the games $G1_H$ (with the highest average fitness value, optimised by RMHC) and $G2_H$ (with the highest average fitness value, optimised by B-RMHC).

C. Manual tuning of the evolved game

It's notable that the game $G2_H$ uses a *doNothing* opponent. We manually edited the ENEMY_ID parameter to use an MCTS controller as opponent instead and asked the two human players to play the edited game. Player A found the new game improved but still a basic game with big missiles, not very interesting compared to the previous games. However, Player B found the new game better with the agent now moving around the map and even increased its position in their personal ranking.

VI. CONCLUSION

One of the big challenges in Game Design is the tuning of game parameters. Given a set of parameter values, a new game instance is created. The difficulty of a game could change significantly when varying one single parameter of a game. The behavior of a human player or an AI agent and the fun
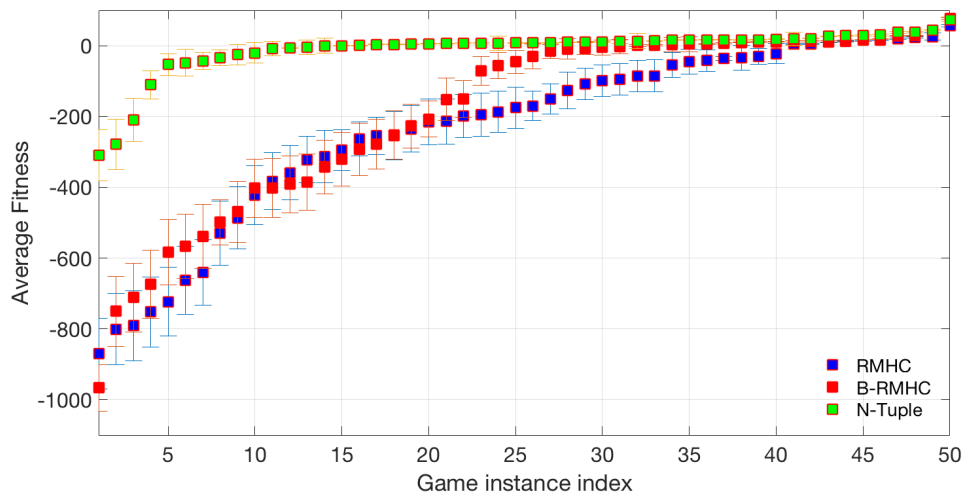
Fig. 2: Sorted average fitness values over 100 evaluations of 50 game instances evolved using three different algorithms. The x-axis shows the game indices after sorting. The standard errors are shown as well.

level of the game will also be affected. For instance, doubling the gravity in *Flappy Bird* will expect to increase the frequency of calling the "jump" actions. However, the selection of game parameters and tuning are not trivial due to the number of parameter to be tuned and the number of possible values of each of the parameters, resulting in a large search space. This motivates the research presented in this paper.

The authors applied the Random Mutation Hill Climber (RMHC) and two new algorithms, the Biased Mutation RMHC (B-RMHC) and the N-Tuple Bandit Evolutionary Algorithm (N-Tuple), to evolving game instances based on a real-time continuous 2-player competitive game called *Space Battle Evolved* (detailed in Section IV-A).

The Biased Mutation RMHC exploits some particular parameters which are considered to be more important after some pre-selection process. The N-Tuple Bandit Evolutionary Algorithm uses a bandit approach to balance the exploration and exploitation of the search space of every game parameter and a model to estimate the quality of unsampled game instances. The statistical results based on the final fitness of the solutions found by the three algorithms suggest the N-Tuple to be significantly better than the other two methods, being able to produce high fitness games.

Two human players have tested some of the evolved games and provided valuable reviews. Both players preferred the new game evolved using the N-Tuple Bandit Evolutionary Algorithm, although they offered mixed opinions on the RMHC games. One highlight of this study is evolving the enemy AI as part of the game parameters. The effect of changing the opponent player was explored in the human trials, indicating that even though this aspect has a great effect on the quality of the gameplay, an outstandingly easy or difficult environment reduces this effect slightly.

The experimental results on optimising *Space Battle Evolved* (Section V) illustrate the outstanding and robust performance of the N-Tuple Bandit Evolutionary Algorithm. With this in mind, we can foresee a bright future for the N-Tuple Bandit Evolutionary Algorithm in AI-Assisted Game Design. Further work will look into the benefits of increasing the number of fitness evaluations, meant to reduce the noise in the evolution. Additionally, although the novel approach used in the Biased Mutation RMHC shows promise, improvements should be considered, such as increasing the re-sampling when measuring parameter importance metrics to produce more accurate results. Another possible future work is to apply this evolutionary algorithm with other game framework, such as GVG-AI [13].

References

[1] J. Togelius and J. Schmidhuber, "An Experiment in Automatic Game Design," in *2008 IEEE Symposium On Computational Intelligence and Games*, Dec 2008, pp. 111–118.

[2] J. Liu, D. P. Liebana, and S. M. Lucas, "Bandit-Based Random Mutation Hill-Climbing," *ArXiv*, 2016. [Online]. Available: http://arxiv.org/abs/1606.06041

[3] M. J. Nelson and M. Mateas, "Towards Automated Game Design," in *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*. Springer, 2007, pp. 626–637, lecture Notes in Computer Science 4733.

[4] A. Isaksen, D. Gopstein, and A. Nealen, "Exploring Game Space Using Survival Analysis," in *Proceedings of the 10th International Conference on the Foundations of Digital Games, FDG 2015, Pacific Grove, CA, USA, June 22-25, 2015*, 2015.

[5] A. Isaksen, D. Gopstein, J. Togelius, and A. Nealen, "Discovering Unique Game Variants," in *Computational Creativity and Games Workshop, hosted by The Sixth International Conference on Computational Creativity, ICCC*, 2015.

[6] A. Isaksen and A. Nealen, "Comparing Player Skill, Game Variants, and Learning Rates Using Survival Analysis," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2015.

[7] D. Perez, J. Togelius, S. Samothrakis, P. Rohlfshagen, and S. M. Lucas, "Automated Map Generation for the Physical Traveling Salesman Problem," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 5, pp. 708–720, Oct 2014.

[8] A. B. Safak, E. Bostanci, and A. E. Soylucicek, "Automated Maze Generation for Ms. Pac-Man Using Genetic Algorithms," in *International Journal of Machine Learning and Computing*, vol. 6, no. 4, 2016, pp. 226–240.
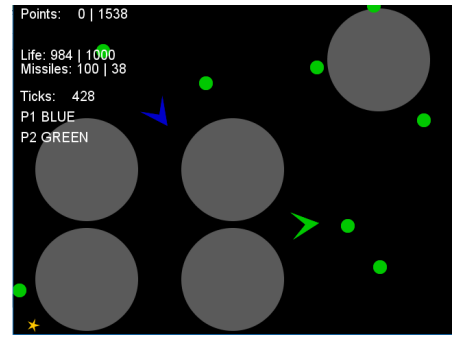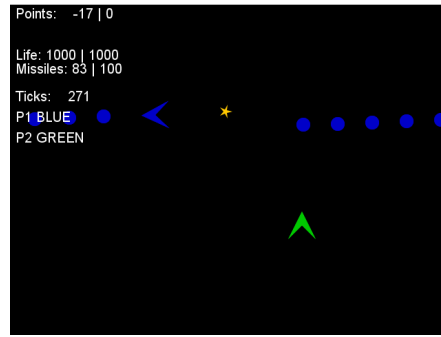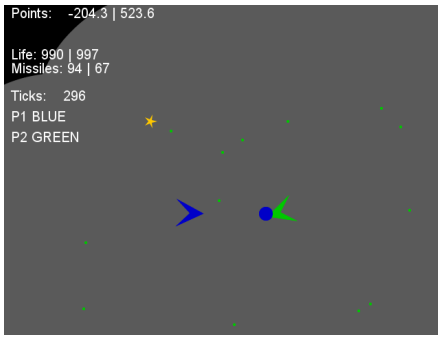
(a) $G1_H$, game with the highest average fitness designed by the RMHC **Player A** complained that although the environment was interesting, there are too many huge black holes overlapping, making it impossible to move. **Player B** found this game annoying, as she was dragged into the center of black holes and couldn't escape, nor get resources.

(b) $G2_H$, game with the highest average fitness designed by the Biased Mutation RMHC. **Player A** found this game terrible (playing against doNothing controller with no black hole and huge missile bombs). **Player B** found this game boring.

(c) $G3_H$, game with the highest average fitness designed by the N-Tuple Bandit Mutation. **Player A** found this game interesting (with 2 big very strong black holes), but it was trivial because the ships could shoot each other when they were in the different black hole centers. **Player B** found this game frustrating, but slightly better than $G3_L$.
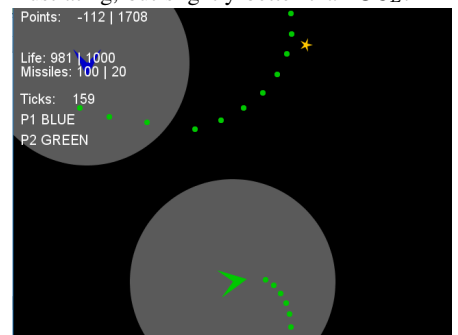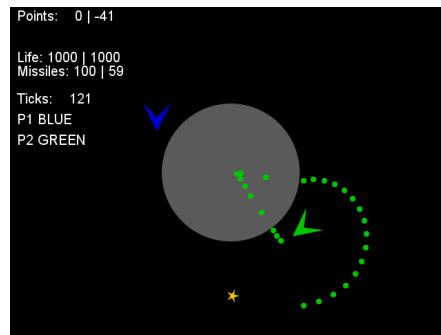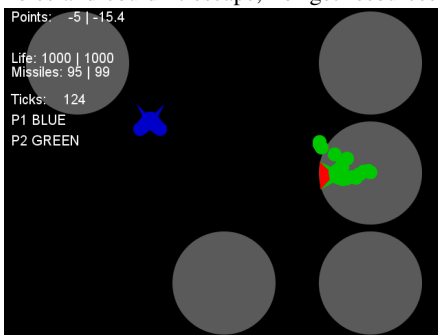
(d) $G1_L$, game with the lowest average fitness designed by the RMHC. **Player A** found it difficult hitting the opponent, but the environment was dynamic and fun. **Player B** found it better than $G1_H$, but still annoying, and suggested to increase the speed of missiles.

(e) $G2_L$, game with the lowest average fitness designed by the Biased Mutation RMHC. **Player A** found this game not challenging at all. **Player B** claimed that this game is her favorite.

(f) $G3_L$, game with the lowest average fitness designed by the N-Tuple Bandit Mutation. **Player A** found this game quite silly because of the missiles flying everywhere. **Player B** found it good, though not as good as $G2_L$.

Fig. 3: Screenshots of the 6 designed games evaluated by human players and their feedback. The players were not told which games they were playing. The game IDs were issued when analyzing the feedback. The **Player A** ranked the games as $G1_L > G3_H > G1_H > G3_L > G2_H > G2_L$ in terms of challenge/fun, while the **Player B** ranked the same games as $G2_L > G3_H > G3_L > G1_H > G1_L > G2_H$.

[9] K. O. S. Julian Togelius, Georgios N. Yannakakis and C. Browne, "Search-based Procedural Content Generation: A Taxonomy and Survey," in *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, vol. 3, no. 3, 2011, pp. 172–186.

[10] F. Rothlauf, *Representations for Genetic and Evolutionary Algorithms*. Heidelberg: Springer, 2006.

[11] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson, "Towards Generating Arcade Game Rules with VGDL," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, Aug 2015, pp. 185–192.

[12] T. L. T. Menezes, T. R. Baptista, and E. J. F. Costa, "Towards Generation of Complex Game Worlds," in *2006 IEEE Symposium on Computational Intelligence and Games*, May 2006, pp. 224–229.

[13] D. Perez-Liebana, 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," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. PP, no. 99, 2015, p. 1.

[14] R. D. Gaina, D. Perez-Liebana, and S. M. Lucas, "General Video Game for 2 Players: Framework and Competition," in *Proceedings of the IEEE Computer Science and Electronic Engineering Conf.*, 2016.

[15] J. Liu, D. Perez-Liebana, and S. M. Lucas, "Rolling Horizon Coevolutionary Planning for Two-Player Video Games," in *Proceedings of the IEEE Conference on Computational intelligence and Games (CIG)*, 2016, p. to appear.

[16] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," in *IEEE Trans. on Computational Intelligence and AI in Games*, vol. 4, no. 1, 2014, pp. 1–43.

[17] M. Buzdalov, A. Buzdalova, and A. Shalyto, "A First Step towards the Runtime Analysis of Evolutionary Algorithm Adjusted with Reinforcement Learning," in *2013 12th International Conference on Machine Learning and Applications*, vol. 1, Dec 2013, pp. 203–208.

[18] S. M. Lucas, "Learning to Play Othello with N-Tuple Systems," *Australian Journal of Intelligent Information Processing*, vol. 4, pp. 1–20, 2008.