Chapter 7 - Automatic General Game Tuning

Diego Perez-Liebana

1 Introduction

Automatic Game Tuning refers to an autonomous process by which parameters, entities or other characteristics of games are adjusted to achieve a determined goal. This goal could be to favour a specific type of player, create games that are more balanced in the case of two-or-more player games, or in order to provide a determined game playing experience to the players. The fact that is *automatic* allows to save some valuable time to game designers and testers, who otherwise need to manually change and play-test the different versions of the game. Furthermore, it can serve as a way of finding new variants of a game that no human had thought before.

As mentioned previously, the fact that the Video Game Description Language (VGDL) is the backbone of GVGAI, allows for a fast tweak-and-test process for game tuning. This chapter studies how VGDL has been modified to facilitate automatic game tuning, by including variables in the language that can be modified from the engine. With this, the GVGAI framework allows not only to create content (levels or rules) for games, as seen in the previous chapter, but also to work on VGDL game *spaces*. This parameterization of GVGAI games (see Section 2 in this chapter) exposes a game space for algorithms to search in; this is, the collection of all possible game variants that the values of the different parameters facilitate.

One possibility to navigate these game spaces is to use optimization techniques. Given an objective (or fitness function), the optimization method explores values for the exposed parameters progressively in order to find games that fulfill better the desired criteria. In this chapter, we propose, in Section 3, to use the N-Tuple Bandit Evolutionary Algorithm (NTBEA) for this purpose. We then focus on automatic game tuning with two different objectives. First, to show how a game can be tweaked to favour one player over another one (see Section 3.2). Secondly, in Section 4, we modify several games to offer different experiences to the player - concretely in the way and time score opportunities are presented along the game.

1.1 Previous Work

One of the first examples of research on game spaces that can be found in the literature [1] defined a parameter space for the game *Flappy Bird*. In this game, the

player has two actions to execute at every frame (*tap* or *no tap*). Each tap makes the player (a bird) to flap its wings and gain some height, while no tapping makes it fall due to the force of gravity. The player must travel to the right of the screen, going through a series of gaps between pipes without touching them. The game ends when the bird touches one of these pipes. Evolutionary algorithms were used to explore the values of the game parameters and the different games that resulted of changing them. Examples of these parameters are the pipe lengths and widths, the (horizontal) distance between pipes, player size, force of gravity, etc. The authors identified four different settings, each one providing a unique gameplay experience. A follow-up work aimed at finding different difficulty levels [2] showed that all playable games are clustered in a certain part of the search space.

In a similar way, Liu et. al. [4] proposed game parameterezation for the two-player game *Space Battle*, in which two ships move in real-time in a 2-dimensional and wrapped space while shooting at each other. The authors used a Random Mutation Hill Climber (RMHC) to evolve game parameters such as the maximum ship and missile speeds, cooldown time (between shootings), missile cost, ship radius and thrust power. The objective of this work is to find game variants in which an MCTS player would defeat Rotate And Shoot (RAS), which is a very simple but powerful strategy in this game. Using UCB1 to select which parameter RMHC should mutate next resulted to be an efficient way to explore the search space and find games with an interesting skill-depth.

A later work by Kunanusont et. al. [3] used a novel Evolutionary Algorithm, the *N-Tuple Bandit EA* (or NTBEA [5]) to explore an even larger space in a new version of *Space Battle*, which counted on 30 in-game parameters. The aim was to find games that would favour skilled players against weak ones, using RAS and two GVGAI agents for training: MCTS (strongest player) and One-Step Lookahead (1SLA, the weakest). The fitness of each game was calculated as the minimum gap of performance between the pairs MCTS-RAS and RAS-1SLA.

2 GVGAI Parameterization

The first step required to explore game spaces in VGDL games is to adapt the language to account for game parameterization. In order to do this, we enhanced VGDL in two ways: First, defining a new section for the language (*ParameterSet*) that would define the types and values of all the parameters that can exist in a VGDL game. Then, allowing the possibility of defining *variables* as values for the properties listed in the *SpriteSet*, *InteractionSet* and *TerminationSet*.

```
1  GameSpace square_size=32
2   SpriteSet
3     background > Immovable img=oryx/space1 hidden=True
4     base     > Immovable img=oryx/planet
5     avatar   > FlakAvatar   stype=sam img=oryx/spaceship1
6     missile > Missile
7       sam   > orientation=UP speed=SSPEED singleton=IS_SAM_SINGLE img=
            oryx/bullet1
8       bomb > orientation=DOWN speed=BSPEED img=oryx/bullet2
9     alien    > Bomber stype=bomb prob=APROB cooldown=ACOOL speed=ASPEED

10      alienGreen > img=oryx/alien3
11      alienBlue > img=oryx/alien1
12    portal  > invisible=True hidden=True
13      portalSlow > SpawnPoint stype=alienBlue cooldown=PCOOL total=
            PTOTAL
14
15   ParameterSet
16     #{Name} > {values(min:inc:max)/(bool)} {desc_string} {[opt]val}
17
18     SSPEED    > values=0.1:0.1:1.0      string=Sam_Speed       value=0.5

19     BSPEED    > values=0.1:0.1:1.0      string=Bomb_Speed
20     APROB     > values=0.01:0.05:0.75   string=Alien_Bomb_Probability
21
22     ACOOL     > values=1:1:5            string=Alien_Cooldown
23     ASPEED    > values=0.5:0.1:1.0      string=Alien_Speed
24     PCOOL     > values=1:1:5            string=Alien_Portal_Cooldown
25     PTOTAL    > values=10:5:60          string=Alien_Portal_Total
26
27     IS_SAM_SINGLE > values=True:False string=Is_Sam_Singleton
```

Listing 1: VGDL Definition of the game *Aliens* enhanced with a ParameterSet. InteractionSet and LevelMapping are unchanged with respect to the default VGDL code for this game (see Chapter 2).

Listing 1 shows an example of a VGDL game, Alens, that has been parameterized. In this case, only the *SpriteSet* has been modified (see the default VGDL code for this game in Chapter 2) and a *ParameterSet* has been added to define the new parameters. Finally, it is important to notice that the first keyword of the VGDL game description file must be `GameSpace`[1]

Each line of the *ParameterSet* (lines 15 to 27) define a new parameter that can be used in the other sets of the description. Each one of these parameters can have up to four different fields.

[1] Note that, in a fully defined VGDL game, this is *BasicGame*.

- **Name:** this is the variable name, a single word that must be used in the other sets to reference this parameter.
- **Values:** this field defines the possible values this parameter may take. Three types are allowed and they are implicitly defined when providing the values, two numeric (`int` and `double`) and a `boolean`.

  - **Numeric:** values are indicated in a tuple of three numbers, $a : b : c$, where a is the minimum, b the increment and c the maximum. For instance, line 19 defines the parameter *BSPEED*, which can take values as defined in $0.1 : 0.1 : 1.0$, where the minimum value is 0.1, the maximum 1.0 and the increment is 0.1. This determines that all possible values for this parameter are 10: $0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9$ and 1.0.
  - **Boolean:** these parameters are always defined as $True : False$ and can take only these two values. An example of a boolean parameter can be seen in line 27.

- **Descriptive String:** an easy to read expression that describes (better than the variable name) what the parameter does.
- **Value:** this is an optional field that allows the designed to initialize a value for the parameter before running the game (see line 18 for an example). If this is present, the values field is ignored. The objective of this field is to facilitate play-testing with the parameters.

Once the *ParameterSet* is defined, it is possible to add variables to the other sets. Listing 1 shows a few examples of this. For instance, line 9, which defines the properties of the aliens (enemies) of this game, uses three variables defined in the *ParameterSet*: *APROB* (which determines the probabilty of an alien dropping a bomb at each frame), *ACOOL* (which decides the number of consecutive frames while the aliens do not execute any action) and *ASPEED* (speed of the aliens, measured in grid cells per movement action).

The framework permits creating games from game space definitions by providing an array of integers for the values of the parameters. Listing 2 shows an example in GVGAI to initialize game spaces. Each one of the values in the array (i) maps one parameter, so the value of this parameter will be $lower_bound + i * increment$. The order of the elements of this array is the same as the order listed when using the function `printDimensions()`, from the object `DesignMachine`.

[t!]

```
//Reads VGDL and loads game with parameters.
String game = "game_filename.txt";
DesignMachine dm = new DesignMachine(game);

//1) Creating a new instantiation of the game space at random:
int[] individual = new int[dm.getNumDimensions()];
for (int i = 0; i < individual.length; ++i)
   individual[i] = new Random().nextInt(dm.getDimSize(i));

//We can print a report with the parameters and values:
dm.printValues(individual);

//Play the game (a human in control)
dm.playGame(individual, game, level1, seed);


//2) Creating a new individual with specific values:
//   Each parameter will take a value = "lower_bound + i*increment"
individual = new int[]{2, 2, 0, 4, 8, 3, 9, 4};

dm.playGame(individual, game, level1, seed);
```

Listing 2: GVGAI Code example (in Java) game space initialization.

For the game space of Aliens defined above, the call to `printDimensions()` provides the following output, which also include the size of the space of possible instantiations of this game and of each dimension:

```
Individual length: 8
Value       S(D)   Range              Description
0.3         10     0.1:0.1:1.0        Bomb_Speed
0.7         6      0.5:0.1:1.0        Alien_Speed
false       2      True:False         Is_Sam_Singleton
5           5      1:1:5              Alien_Cooldown
50          11     10:5:60            Alien_Portal_Total
0.4         10     0.1:0.1:1.0        Sam_Speed
0.46        15     0.01:0.05:0.75     Alien_Bomb_Probability
5           5      1:1:5              Alien_Portal_Cooldown
Search Space Size: 4.950E6
```

Listing 3: Game Space for the game aliens, as indicated in the GVGAI output. Columns are, from left to right: Final value, Dimension size, Range of values and Comprehensive description.

This feature of GVGAI is available for both single- and two-player games. In the rest of this chapter, we show two different use cases in which evolution is used to find different instantiations according to certain search criteria.

3 Evolving Games for Different Agents

In this section, we propose the following problem: given two agents with different skill capabilities, is it possible to find instances of one game (Aliens) in which one agent performs better than the other? And viceversa?

First, we analyze a new Evolutionary Algorithm, the N-Tuple Banding Evolutionary Algorithm [6], which will be used to search the space of possible Alien games. Secondly, we will observe the different results that can be achieved with this method in this environment.

3.1 The N-Tuple Bandit Evolutionary Algorithm

The N-Tuple Bandit Evolutionary Algorithm (NTBEA) [6] is an optimization algorithm specially suited for large search spaces in which the evaluation of each one of its points is computationally very expensive. NTBEA counts on an N-Tuple system that captures the statistics of the inherent model and the combination of the values of its discrete parameters.

NTBEA is formed of three different parts: a bandit landscape model, an evolutionary algorithm and a fitness evaluator subject to noise. In principle, we can assume that the execution of querying the landscape model is negligible in comparison with evaluating a potential solution. Figure 1 depicts the three components of the algorithm.

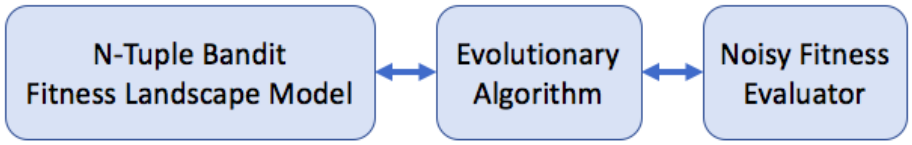

Fig. 1: Key components of NTBEA.

NTBEA works as follows. Search starts from a single point in the search space, chosen uniformly at random. We refer to this point as the *current point*. This point is evaluated once in the target problem, using the noisy evaluator. There is no need for resampling (even in noisy problems) directly. The bandits built in the model may require to re-evaluate a candidate solution in future steps. Note that the algorithm also works for noise-free problems, but without loss of generality it assumes the harder case (where the problem is noisy).

The current point and its fitness value is stored in the bandit landscape model (for brevity, referred to as *the model* from now on). Then, the algorithm advances to

select the next current point. In order to do this, the model is then searched around the neighborhood of the current point. This neighborhood is defined by a number of neighbours and the proximity distribution to the current point. This distribution is controlled directly by the mutation operator. The next point will be such point in the neighbourhood with the highest estimated UCB1 value. This process continues until a the termination condition is met (i.e. evaluation budget or some other criterion).

Estimating UCB Values: an *N*-Tuple Approach One of the key parts of this algorithm is how to estimate the UCB values when sampling a large search space. With large, we assume that it is impossible to evaluate all possible points in the solution space, because the number of fitness evaluation allowed by the budget is smaller than the search space size. The relationship between the points sampled and their neighbours in the search space needs then to be modeled properly.

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

UCB1, shown again in Equation 1, combines a exploitation term ($Q(s,a)$ on the left) and an exploration expression (on the right) balanced by a control parameter C (higher values lead to a more exploratory search and lower values produce more greedy selections). $N(s)$ is the total number of times a bandit has been played, while $N(s,a)$ indicates the number of times the arm a has been played.

For NTBEA, each dimension of the search space is modelled as an independent multi-armed bandit and each arm represents a possible value. The standard UCB formula does not contain ϵ, in order to guarantee that all arms are pulled at least once. In our case, this would be impractical, as it would force an exhaustive exploration of the search space. ϵ in Equation 1 relaxes this requirement.

Additionally, combinations of arms are also modeled as *super-bandits*. In a d-dimensional search space where each dimension has n possible values, the largest super-bandit (which combines all dimensions) would have n^d arms. Rather than using such large bandit, we aggregate over all the N-Tuples in the N-Tuple System model. Let N be the N-Tuple indexing function such that $N_j(x)$ indexes the j^{th} bandit for a point x in the search space. The aggregate UCB value for solution point x is computed as the unweighted average, as defined in Equation 2, where m indicates the total number of bandits defined.

$$v_{UCB}(x) = \frac{1}{m} \sum_{j=1}^{m} \mathrm{UCB}_{N_j(x)}, \tag{2}$$

The N-Tuple bandit system sub-samples the dimensions of a d-dimensional search space with a number of N-tuples. N can takes any values from 1 to d, reaching 2^d bandits if all tuples are considered. Each N-Tuple is assigned a look-up table (LUT) that stores statistical summaries of the values associated with it. These statistics contain the number of samples, its sum and the sum of the square of the fitness of these samples. These values allow for the computation of the mean, standard deviation and standard error fore each N-Tuple. These measurements not only provide a great insight into the system that is being modeled, but they also provide all components required for Equations 1 and 2.

NTBEA is described in Algorithm 1. First, it chooses a random point in the serach space (*current point*). Each one of these points is represented with a vector of integers, where each element is an index to a value in that dimension. There is no restriction towards the type of value it refers to (i.e. integer, double, boolean, etc.).

Given an evaluation budget, the following steps are repeated until the search finishes.

1. A (noisy) fitness evaluation of the current point is made, to then store its fitness in the N-Tuple Fitness Landscape Model. This is the value given to that solution point (lines 6 and 7 in Algorithm 1).
2. From the current solution, a set of unique neighbours is generated using the mutation operator. This represents the population of the current iteration (line 8).
3. The fitness landscape model calculates the (ucb) value for each one of these neighbours, using Equations 1 and 2. The new current solution is set as the neighbour with the highest UCB value.

Once the budget has been exhausted, NTBEA recommends a point among the evaluated solutions and their neighbours, in which each dimension is set to the value with maximal approximate value defined in (2).

An Illustrative Example We take a 5-dimensional space and model it using five 1-tuples and one 5-tuple. In this model, we have four sample points (three of them unique) and their fitness as shown in Table 2 (left). Given these fitness values, the first 1-tuple (i.e. corresponding to the first dimension) has a LUT entry with two entries with a mean of 1 for $LUT[0 * * * *]$ and $\frac{2}{3}$ for $LUT[1 * * * *]$. The 5-tuple has three non-empty entries: $LUT[12340]$ has a mean of 0.5, and $LUT[11111]$ and $LUT[00110]$ both have means of 1. Some other statistics (only for the non-null table entries) that can be found in the system are shown in Table 2 (right). Note that, as indicated above, other measurements such as the standard deviation and standard error are also available for each N-tuple entry.

Algorithm 1 The N-Tuple Bandit Evolutionary Algorithm. This description outlines the simplest case (1 current point), but a population-based version is also possible.

Input: $n \in N^+$: number of neighbors
Input: $p \in (0, 1)$: mutation probability
Input: $flipOnce \in \{true, false\}$ indicates if flip at least once or not during mutation
Oputput: $LModel$: landscape model of the problem.

function NTBEA($n, p, flipOnce$)
  $t = 0$ Counter for fitness evaluations
  $LModel \leftarrow$ Initialise the fitness landscape model
  $current \leftarrow$ random point $\in S$
  while $t < nbEvals$ **do**
    $value \leftarrow$ FITNESS($current$)
    add $< current, value >$ to $LModel$
    $Population \leftarrow$ NEIGHBORS($LModel, current, n, p, flipOnce$)
    $current \leftarrow \arg\max_{x \in Population} v_{UCB}(x)$
    $t \leftarrow t + 1$
  return $LModel$

function NEIGHBORS($model, x, n, p, flipOnce$)
  $Population \leftarrow \{\}$ ▷ Initialise empty set
  $d \leftarrow |x|$ ▷ Get the dimension
  for $k \in \{1, \ldots, n\}$ **do**
    $neighbor \leftarrow x$
    $i \leftarrow 0$
    if $flipOnce$ **then**
      $i \leftarrow$ randomly selected from $\{1, 2, \ldots, d\}$
      for $j \in \{1, \ldots, d\}$ **do**
        if $i == j$ **or** $RAND < p$ **then**
          Randomly mutate value of $neighbor_j$
    Add $neighbor$ to $Population$
  return ($Population$)

3.2 Variants of Aliens for Agents with Different Look-aheads

This section shows an example of usage of NTBEA for tuning parameters of the game Aliens. First, a Game Space is created for this game in VGDL, as described above. The resultant search space is described in Table 1, including the total search space.

The objective set for this experiment is to find games where an agent A achieves a victory rate as highest as possible, while an agent B loses as many games as possible. Both agents are based on the Monte Carlo Tree Search (MCTS) agent distributed with the GVGAI framework and the difference between the two resides on the depth

Solution	fitness
$[1, 2, 3, 4, 0]$	1
$[1, 1, 1, 1, 1]$	1
$[0, 0, 1, 1, 0]$	1
$[1, 2, 3, 4, 0]$	0

N-tuple	Pattern	Mean	Nb. of eval.
1-tuple	$[0, *, *, *, *]$	1	1
	$[1, *, *, *, *]$	$\frac{2}{3}$	3
	$[*, 0, *, *, *]$	1	1
	$[*, 1, *, *, *]$	1	1
	$[*, 2, *, *, *]$	$\frac{1}{2}$	2
	$[*, *, 1, *, *]$	1	2
	$[*, *, 3, *, *]$	$\frac{1}{2}$	2
	$[*, *, *, 1, *]$	1	2
	$[*, *, *, 4, *]$	$\frac{1}{2}$	2
	$[*, *, *, *, 0]$	$\frac{2}{3}$	3
	$[*, *, *, *, 1]$	1	1
5-tuple	$[0, 0, 1, 1, 0]$	1	1
	$[1, 1, 1, 1, 1]$	1	1
	$[1, 2, 3, 4, 0]$	$\frac{1}{2}$	2

Fig. 2: Sample points and their corresponding fitness values (left) and some non-null table entries stored in the system (right).

Name	**Description**	**Possible Values**	**Size**
BSPEED	Speed of the aliens' bombs	0.1, 0.2, ..., 1.0	10
ASPEED	Speed of the aliens	0.1, 0.2, ..., 1.0	10
IS_SAM_SINGLE	Are sams (player's bullet) singleton?	True / False	2
ACOOL	Cooldown for aliens' movement	1, 2, 3, 4, 5	5
PTOTAL	Number of aliens to be spawned	10, 15, 20, 25, ..., 60	11
SSPEED	Speed of the avatar's sams	0.1, 0.2, ..., 1.0	10
APROB	Probability of alien dropping a bomb	0.01, 0.05, 0.1, 0.15, ..., 0.75	15
APCOOL	Cooldown for alien spawning portal	1, 2, 3, 4, 5	5
Total search space size		4.95×10^6	

Table 1: *Aliens*' parameter set search space

of their Monte Carlo simulations. Two depths are used in this study: 20 and 5. It is worth noting that MCTS, using depth 10 and $40ms$ for decision time (the same budget has been used for these experiments), achieves 100% victories on the default game of Aliens.

Two different experiments have been conducted.

- Experiment `I`: Agent A (the one whose number of victories must be maximized) has a simulation depth $d_A = 20$, while agent B has a maximum depth of $d_B = 5$.
- Experiment `II`: $d_A = 5$ and $d_B = 5$.

Note that these experiments are searching for two different objectives: experiment `I` favours agents with long look-aheads, while experiment `II` favours those with shorter ones.

Both agents A and B are used to evaluate all points in the search space. Each game is played twice, one per agent. The final outcome ($o_A, o_B \in \{0, 1\}$, where 0 means a loss and 1 a win) and the final score (s_A, s_B) are recorded. The performance of each agent on each game is computed as $V_i = s_1 \times (1 + C \times o_i)$. Note that this expression rewards agents winning the game by a factor C, which multiplies the score obtained by the agent. The fitness of the individual (instantiation of the given game) is defined be the logistic function shown in Equation 3.

$$Fitness = \frac{1}{1 + e^{-(V_A - V_B) \times K}} \tag{3}$$

where K smooths the steepness of the logistic function. Therefore, a fitness value of 0.5 would indicate a similar performance of both agents in the game. A value close to 1 indicates that the agent A performs much better than B and the contrary is true for values close to 0. 10 repetitions were run for each experiment, lasting for 10000 generations each, each one with a random seed for the game kept constant during the ran, chosen uniformly at random.

Figure 3 show the progression of the fitness as defined in Equation 3 for both experiments. Note that these images plot the cumulative average of the fitness from the start until the given generation. Given the noisy nature of the agents and the exploratory component of NTBEA, individual fitness measurement is very noisy and their plot does not offer much information.

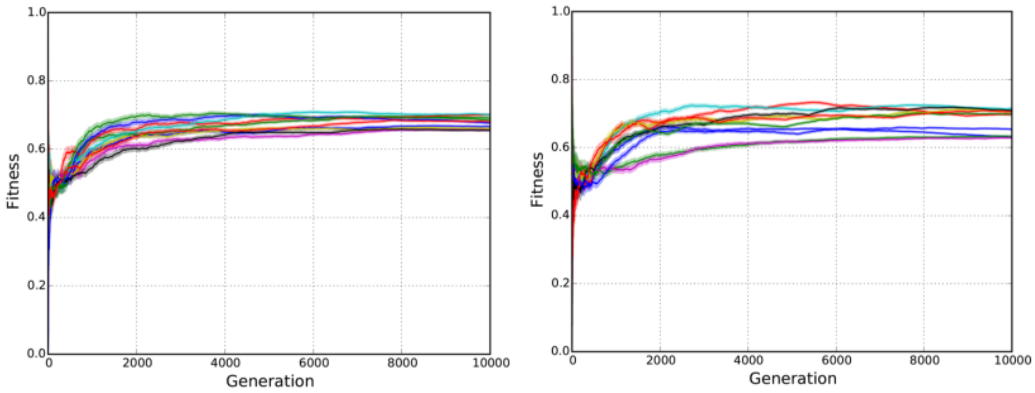

Fig. 3: NTBEA Fitness progression on Aliens games. On the left, experiment `I`. On the right, experiment `II`. All games played by two MCTS agents with simulation depths 5 and 20.

As can be seen, all fitness values average around 0.5 at the start of the runs, in which represents games being explored for which both agents perform similarly. In both experiments (`I` on the left, `II` on the right), all runs progressively find games in which fitness is higher (respectively, agent A performing better than B on the left, vice-versa on the right).

Each generation evaluates one point (with no re-sampling) of the search space. It is therefore worth validating that the final games recommended by NTBEA have the desired properties. In order to check this, a validation experiment has been performed by playing 20 times each one of the recommended games (per run, thus 10 in total) and averaging the fitness values achieved. Furthermore, for this validation games are played with different random seeds to the ones used during their evolutionary runs. Figure 4 shows that all evolved games achieve a greater than 0.5 fitness, which means that all games are played better by one agent than the other.

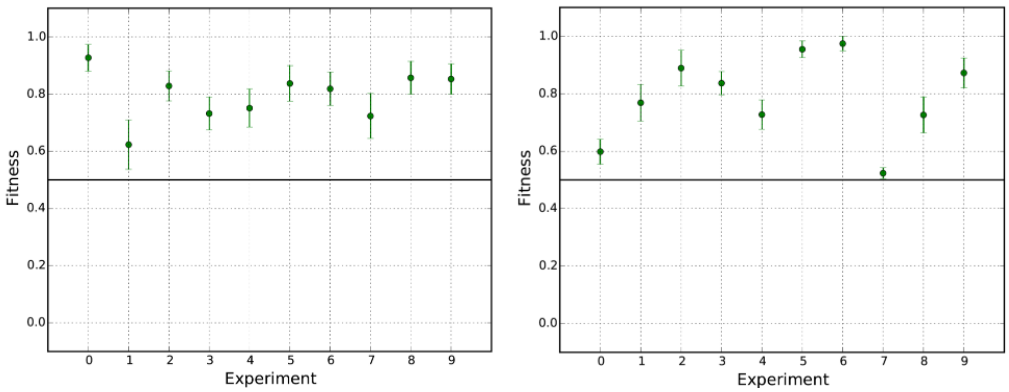

Fig. 4: Validation of the games recommended by NTBEA after evolution. Each bar is the average of fitness values achieved fruit of 20 evaluations of each game. On the left, experiment `I`. On the right, experiment `II`. Games are played by MCTS with simulation depths 5 and 20.

Finally, an extra set of games have been played in these recommended games using different agents and an extra set of seeds for the random generators. A Rolling Horizon Evolutionary Algorithm (RHEA, see Chapter 3) has been used with individual lengths 5 and 20, same as used in the experiments, in order to test that the games evolved do not only have the desired properties when playing with the agents used during evolution. Table 2 shows the percentage of victories of each agent, the

average score and time steps. As can be seen, the evolved games are robust to the type of agent used to play them and the seeds used.

Agent	% Victories	Average Score (std error)	Time Steps (std error)
Results obtained in games evolved for Experiment `I`			
RHEA (d=20)	100	61.65 (0.92)	101.85 (4.45)
RHEA (d=5)	5	65.45 (1.17)	133.65 (1.79)
Results obtained in games evolved for Experiment `II`			
RHEA (d=20)	0	29.05 (4.19)	98.8 (8.28)
RHEA (d=5)	100	124.7 (0.84)	344.55 (14.38)

Table 2: Test performed with RHEA agents on the games recommended by NTBEA. First two row of results refer to those games in which longer depth is preferred ($d = 20$). Last two row refer to results in games where shorter depth is preferred ($d = 5$).

Examples of generated games Figure 5 shows a screenshot of one of the games recommended for experiment `I` (favouring longer look-aheads). Table 4 shows the final parameters found by NTBEA for this game.

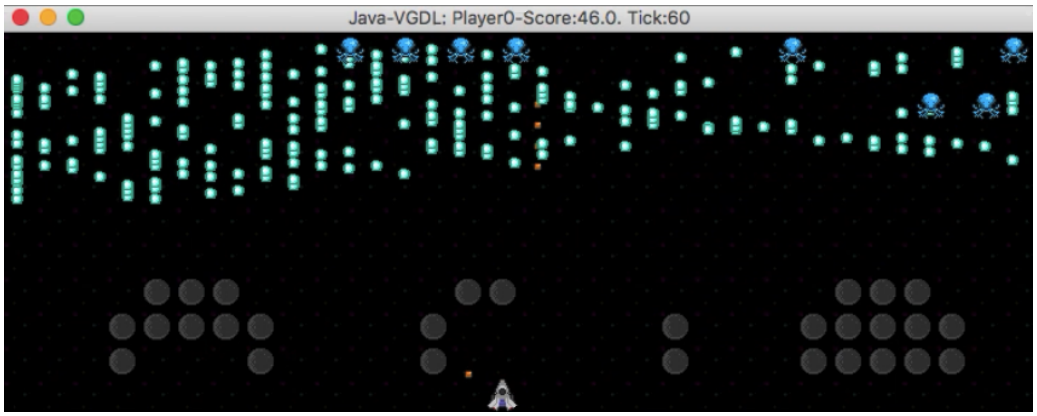

Fig. 5: Screenshot of the game *Aliens* evolved for the Experiment *I*, where longer look-aheads are preferred. A video of this game, played by RHEA ($d = 20$), can be found at: `https://www.youtube.com/watch?v=PWHGM_Bd6Jw`

Parameter	Value	Parameter	Value	Parameter	Value	Parameter	Value
BSPEED	0.1	ASPEED	0.8	IS_SAM_SINGLE	True	ACOOL	1
PTOTAL	25	SSPEED	0.6	APROB	0.3	APCOOL	2

Table 3: Parameters tuned by NTBEA for one of the games of Experiment *I*.

As can be seen, the game looks much different than the original one (see Chapter 2). There are many more alien bullets on screen (note parameter BSPEED in Table 4 is the minimum non-zero possible speed for the bullets) and with bullet being very close to each other on the y-axis, product of a relatively high probability of shooting ($APROB = 0.3$). This creates a type of games where a *cascade* of bullets is thrown upon the player. The only chance for the player to win the game is to destroy all aliens quickly, before the unavoidable waterfall of bullets reaches the player first. Precision for shooting at the enemies is needed for this, and an algorithm with a longer look-ahead can reach the rewards available by killing aliens in its planning horizon better than an agent with a shorter simulation depth.

Conversely, Figure 6 shows a screenshot of a game evolved for experiment `II` and Table 4 the final parameter values.

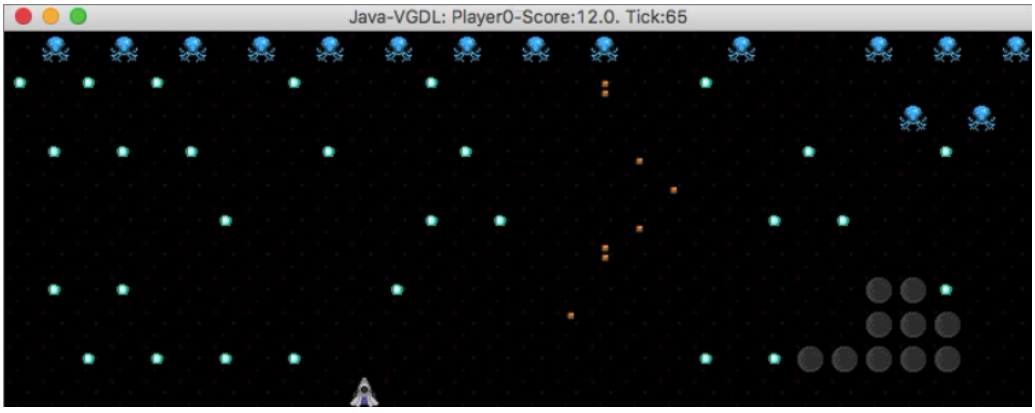

Fig. 6: Screenshot of the game *Aliens* evolved for the Experiment *II*, where shorter look-aheads are preferred. A video of this game, played by RHEA ($d = 5$), can be found at: `https://www.youtube.com/watch?v=t6usa_f9jig`

As can be seen, the game looks again very different to previous versions of Aliens shown in this book. In this case, the enemies move faster ($ASPEED = 0.1$) and they are more separated from each other ($APCOOL$ higher in this case than in Table 4).

Parameter	Value	Parameter	Value	Parameter	Value	Parameter	Value
BSPEED	1.0	ASPEED	1.0	IS_SAM_SINGLE	True	ACOOL	2
PTOTAL	55	SSPEED	0.3	APROB	0.5	APCOOL	4

Table 4: Parameters tuned by NTBEA for one of the games of Experiment *II*.

Bullet speed is also higher than in the previous case, with $BSPEED = 1.0$ as a parameter. The resultant game is much faster paced than the example of experiment *I* but in this case, with great skill, it is possible to dodge bullets. This maneuver is easier for an algorithm with shorter look-ahead[2] and allows this type of agent to survive long enough until the enemies reach the simulation horizon, when they can be killed. Longer-sighted agents are typically killed by these bullets as they are less precise on the shorter range.

We found really interesting that the same algorithm, NTBEA, is able to evolve and create new games (with dynamics and relationships between the game parameters that were never thought of before) that respond to opposed objectives in a stable and robust (to noise and agents) way. This result opens the path to a line of research that explores game spaces automatically by using agents to evaluate them.

4 Modelling Player Experience

In this section we describe our work on using NTBEA to tweak VGDL games with the objective of adjusting the game experience of the players. In particular, the aim is to modify the game parameters so the score progression that their playing agents achieve follows a pre-determined curve.

4.1 Designing the Search Space

In the original VGDL, all sprites that are spawned from *portal* sprites are created at a constant rate during the whole game. in order to enrich the space where interesting games can exist, these portals have been modified to establish a two limits, lower and upper, which determine when the portal is allowed to spawn sprites. These limits are now part of the possible parameters that can be tweaked by any optimization algorithm. It is worth noting that the original version of the games with portals are still valid points in the search space using these limits (concretely, setting the lower limit to 0 and the upper limit to 2000).

For this work, three games were chosen and parameterized: *Defender*, *Waves* and *Seaquest*. Figure 7 shows screenshots of the three games, as in their original

[2] Having shorter depths and the same budget, more certain analysis can be done of the near future events.

implementation. In Defender, the player controls an aircraft that aims at destroying some aliens that are bombing a city. The player can shoot missiles but requires ammunition, which is provided via supply packs that fall from the sky. Aliens move from their spawn points horizontally to the left and are harmless to the avatar.

In Waves, the player must again try to fight aliens but in this case the objective is survival. Aliens are spawn from the right and move towards the player, located initially on the left part of the screen. Aliens shoot missiles at the player which this must avoid but that can be destroyed by the player's own bullets. When this happens, a shield drops that can be picked up for extra protection. Finally, Seaquest is a port of the original game with the same name, in which the player controllers a submarine that must rescue divers spawned at the bottom of the sea. The submarine can stay under water for a certain amount of time before oxygen runs out, before which the player must come to the surface or game is over. Different types of animals (whales, sharks and piranhas) move horizontally and kill the player upon contact.

These games have been chosen due to the possibility of designing large game spaces in them, which can provide multiple different instances of the same game. Tables 5, 6 and 7 describe the search spaces designed for these games (respectively) using the VGDL parameterization model shown in this chapter. All these search spaces reach a size of 10^{10}.

Name	**Description**	**Possible Values**	**Size**
BSPEED	Bomb speed	0.1, 0.3, 0.5, 0.7, 0.9	5
ASPEED	Alien speed	0.2, 0.4, 0.6, 0.8, 1.0	5
SUPSPEED	Supply falling speed	0.05, 0.25, 0.45	3
APROB	Alien's probability to shoot a bomb	0.01, 0.02, 0.03, 0.04, 0.05	5
SLOWPPROB	Slow portal's probability to spawn an alien	0.05, 0.1, 0.15, 0.2, 0.25	5
FASTPPROB	Fast portal's probability to spawn an alien	0.3, 0.5	2
AMPROB	Supply portal's probability to spawn a supply	0.05, 0.15, 0.25	3
ACOOLDOWN	Alien's bomb shooting cooldown	2, 4, 6, 8, 10	5
PCOOLDOWN	Alien portal's cooldown	5, 10, 15, 20	4
AMCOOLDOWN	Supply portal's cooldown	5, 10, 15, 20,	4
BLIMIT	Avatar's maximum ammo supply	5, 10, 15, 20	4
ADDSUP	Amount of ammo a supply pack contains	1, 2, 3, 4, 5	5
LOSSCITY	Score lost when a city is destroyed	-4, -3, -2, -1	4
AREWARD	Score gained when an alien is shot	1, 3, 5, 7, 9	5
DELAY	The time step that all portals start spawning	0, 50, 100, 150, 200, 250, 300	7
CLOSE	The time step that all portals stop spawning	350, 400, 450, 500	4
Total search space size		1.08×10^{10}	

Table 5: Defender's parameter set search space

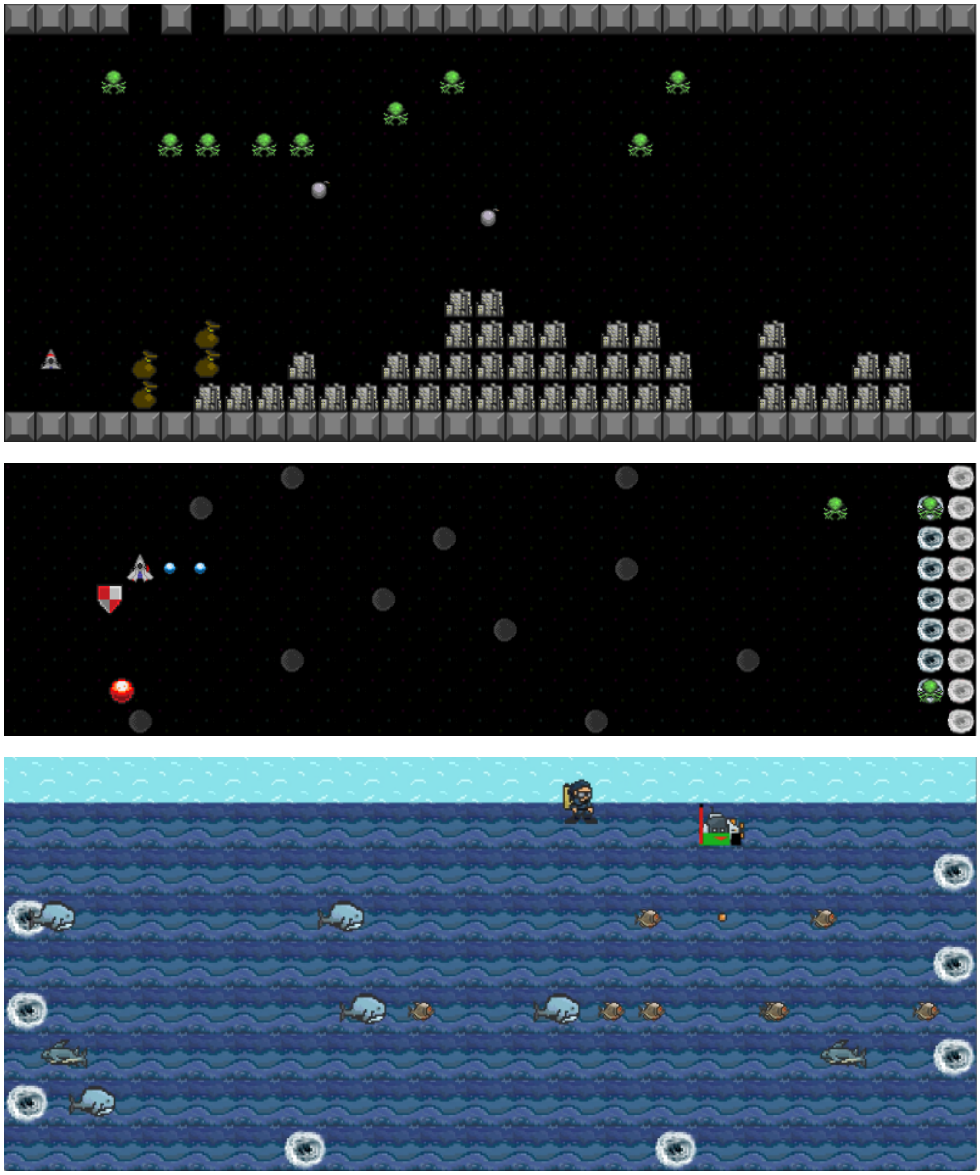

Fig. 7: Screenshots of the games, from top to bottom: *Defender*, *Waves* and *Seaquest*.

Selected Target Score Functions The objective of this work is to find, in the search space of these games, particular instances of them where the players are exposed to certain predetermined target score functions. All functions designed for this experiment are positive-definite ($f(x) > 0$ for all $x > 0$). We have defined four functions: *linear* (Equation 4, with $m \in \{0.2, 0.4, 1\}$), *shifted sigmoid* (Equation 5,

Name	Description	Possible Values	Size
RSPEED	Rock's speed	0.45, 0.95, 1.45, 1.95, 2.45	5
SSPEED	Avatar missile's speed	0.5, 1.0, 1.5, 2.0	4
LSPEED	Laser's speed	0.1, 0.2, 0.3, 0.4, 0.5	5
ACOOLDOWN	Alien portal's cooldown	2, 6, 10, 14	4
RCOOLDOWN	Rock portal's cooldown	2, 6, 10, 14	4
APROB	Alien portal's probability of alien spawn	0.01, 0.05	2
RPROB	Rock portal's probability to spawn a rock	0.15, 0.2, 0.25, 0.3, 0.35, 0.4	6
PSPEED	Avatar's speed	0.5, 1.0, 1.5	3
ASPEED	Alien's speed	0.05, 0.1, 0.15, 0.2, 0.25, 0.3	6
SLIMIT	Avatar's maximum health point	2, 4, 6, 8 , 10	5
ASPROB	Alien's probability to shoot a laser	0.005, 0.01, 0.015, 0.02	4
SPLUS	Avatar's health increase when picking a shield	1, 2, 3, 4, 5	5
APEN	Score lost when avatar collides with alien	-4, -3, -2, -1	4
LASERPEN	Score lost when avatar hit by a laser	-4, -3, -2, -1	4
SREWARD	Score gained when an alien is shot	1, 3, 5, 7 , 9	5
DELAY	Time step at which portals start spawning	0, 50, 100, 150, 200, 250, 300	7
CLOSE	Time step at which portals stop spawning	350, 400, 450, 500	4
Total search space size		7.741×10^{10}	

Table 6: Waves' parameter set search space

with $K_3 \in \{3, 12\}$), *logarithmic* (Equation 6, $L = 15$) and *exponential* (Equation 7). Note that the last two functions require a fast increase in the score rate either at the start or at the end of the games. These functions are plotted in section 4.2 together with the results, for the shake of space.

$$f(x) = mx \tag{4}$$

$$f(x) = 150 \times \left(\frac{1}{1 + \exp(-\frac{x}{30} + K_3)}\right) \tag{5}$$

$$f(x) = L \log_2 x \tag{6}$$

$$f(x) = 2^{\frac{x}{70}} \tag{7}$$

It is important to highlight that these functions are targets, and as such they may be impossible to achieve by a playing agent in the game space defined. However, our objective is to use NTBEA to tune games so the recommended instances provide an experience, as defined by the score trend, that approximate these ideal progression curves as much as possible.

Name	Description	Possible Values	Size
SSPEED	Shark's speed	0.05, 0.2, 0.35, 0.5	4
WSPEED	Whale's speed	0.05, 0.2	2
PSPEED	Piranha's speed	0.05, 0.2, 0.35, 0.5	4
DSPEED	Diver's speed	0.1, 0.3, 0.5, 0.7, 0.9	5
SHPROB	Shark portal's probability of shark spawn	0.01, 0.06, 0.11, 0.16	4
WHPROB	Whale portal's probability of whale spawn	0.005, 0.025, 0.045, 0.065, 0.085	5
DHPROB	Normal diver portal's probability of spawn	0.005, 0.015, 0.025, 0.035, 0.045	5
OFDHPROB	Fast diver portal's probability of diver spawn	0.05, 0.07, 0.09	3
WSPROB	Whale's probability to spawn a piranha	0.01, 0.04, 0.07, 0.1	4
HP	Avatar's initial oxygen amount	9, 17, 25, 33	4
MHP	Avatar's maximum oxygen amount	10, 20, 30, 40	4
HPPLUS	Oxygen gained per time step at the surface	1, 2, 3, 4	4
TIMERHPLOSS	Oxygen amount lost per time step underwater	5, 10, 15, 20	4
WHALESCORE	Score increased when a whale is shot	5, 10, 15, 20	4
DCONS	Consecutive tiles a diver can move per step	1, 2, 3	3
CRLIMIT	Max divers the avatar can rescue in one dive	1, 3, 5, 7	4
DELAY	The time step that all portals start spawning	0, 50, 100, 150, 200	5
SHUTHOLE	The time step that all portals stop spawning	200, 250, 300, 350, 400	5
Total search space size		5.892×10^{10}	

Table 7: Seaquest's parameter set search space

Fitness Calculation Potential solutions explored by NTBEA are evaluated using the RHEA agent available in the GVGAI framework during evolution. Each game played records the score at every game tick and the final outcome (win or loss). We use a Normalized Root Mean Square Error (NRMSE) to compute the deviation between the score obtained during the game and the target trend. Let $\hat{s}$ be the vector of scores achieved from timestep 1 to n (the last game tick), and $\hat{y}$ the vector of target scores for a given function. RMSE is calculated as shown in Equation 8, which is also the loss function on the target.

$$Loss(\hat{s}, \hat{y}) = NRMSE(\hat{s}, \hat{y}) = \frac{\sqrt{\sum_{i=1}^{n}(\hat{y}_i - \hat{s}_i)^2}}{n(\hat{y}_{max} - \hat{y}_{min})} \tag{8}$$

We then define $1 - Loss(\hat{s}, \hat{y})$ as the fitness function to be maximized by NTBEA.

4.2 Evolving Games for Player Experience

Experimental setup NTBEA has been used to evolve parameters on the three VGDL games described in the previous section: Defender, Waves and Seaquest, aiming to fit the score progression of an RHEA agent to different variations of the four target functions described above. Furthermore, an MCTS agent has been used to validate the games finally suggested by NTBEA, playing 20 times each one of them. In total, 21 different experiment settings, resulting of testing 7 variations of these target functions, have been tested:

- Linear function, Equation 4, $m = 0.2$.
- Linear function, Equation 4, $m = 0.4$.
- Linear function, Equation 4, $m = 1.0$.
- Sigmoid function, Equation 5, $K_3 = 3$ (shifted left).
- Sigmoid function, Equation 5, $K_3 = 12$ (shifted right).
- Logarithmic function, Equation 6.
- Exponential function, Equation 7.

Ten runs have been performed for each one of these settings and outcomes have been averaged to present their results. RHEA was executed using a population size of 20 with an individual length $l = 10$ and mutation rate $1/l$. The C value for the tree policy of the MCTS agent is set to $\sqrt{2}$ and the simulation depth $d = 10$. Both agents count on the same value function to evaluate a state, which follows Equation 9. In this scenario, the value would be the score of the game unless the game is over and has been won (1000) or lost (-1000). It is important to highlight, thus, that the agents are always aiming to maximize score. Hence, the score trend shown in a game depends mostly on the characteristics of the game itself, with noise introduced by the inherent stochasticity of the games and agents employed.

$$V(s) = \begin{cases} score(s), & \text{otherwise} \\ 1000, & \text{if game won in state } s \\ -1000, & \text{if game lost in state } s \end{cases} \tag{9}$$

Finally, C for the NTBEA bandits is also set to $\sqrt{2}$ and the number of neighbours for NTBEA is established at 100.

Evolving for a linear score progression Figure 8 shows the average fitness trend over 500 generations of NTBEA in the three games studied in this work, when using the linear function $y = 0.2x$ as the target score trend. In this curve (as for all the

others), x represents the time step and y the score at time step x. For these plots, the optimal value is 1 (as it minimizes the loss - Equation 8 - to 0). The plots represent an average of the 10 runs performed per experimental setting, and the light blue shaded area indicates the standard deviation of the values.

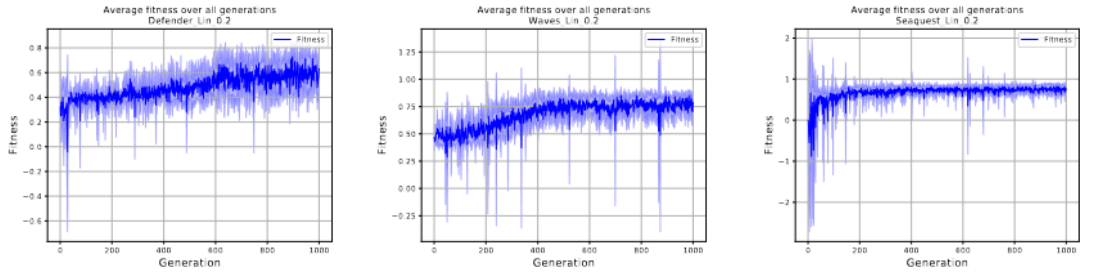

Fig. 8: Average fitness throughout evolutions for $y = 0.2x$ on the games of this study (left to right: *Defender*, *Waves* and *Seaquest*.

As can be seen, the fitness progression in the three games approximates to 1, which translates to explored points in the search space that represent games for which the score trend that RHEA achieves is close to $y = 0.2x$. Fitness values seem to stabilize at generation 1000 for all games. In Seaquest (8-right), convergence was achieved quickly (around generation 200), while Waves stabilized a bit later (8-center, 500 generations). Defender (8-left) took more time to reach a stable fitness (albeit with a higher standard deviation), after close to 800 generations.

Another interesting way of analyzing this progression is to plot the actual score trend that the games achieve per generation. As that would be quite difficult to include in a single figure, we have taken average of score trends in consecutive generations. In order to be able to see this progression, segments are created of N generations each, and an average is plotted for all of them. The value of N is adjusted per game for a better visualization, as different games evolved at different speeds (but N is kept constant though the experiment for all runs). If NTBEA is progressing in the right direction, it is expected that consecutive segments approximate better the target function. Figure 9 plots these curves per generation segment for the three games evolved to fit $y = 0.2x$. Note that the red function is the target sought.

As can be observed, the first curves (blue lines) diverge considerably to from the target progression. These are the initial game instances explored by NTBEA, where the algorithm selects points in the search space with little to no information. See, for instance, how in Seqauest (9, right) the first average score trend is above the target

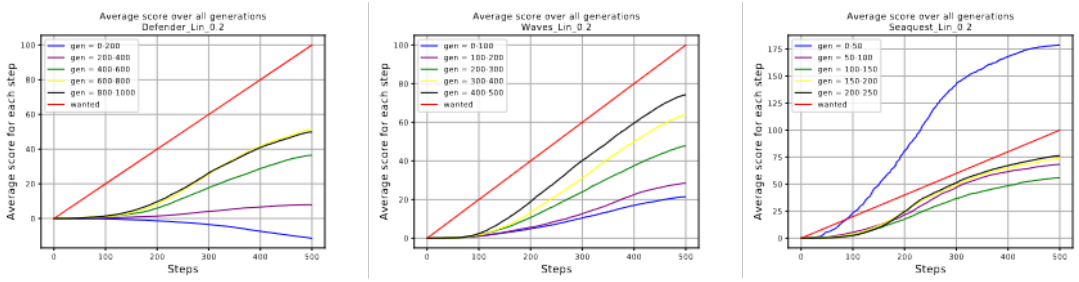

Fig. 9: Average score trend throughout evolution for the linear function $y = 0.2x$ on the games tested in this study (left to right: *Defender*, *Waves* and *Seaquest*).

by a significant gap. In the other games, the initial score progression lies below the target in Waves (Figure 9 , center), and it is negative (i.e. the player loses points) in *Defender*.

As the landscape model of NTBEA becomes more accurate, the individuals explored show a better performance and the progression curves approximate the target better. In general, the black curves (last generation segment) achieves the smallest error with respect to the target for all games and aimed trends. For all games, NTBEA manages to find parameter sets that are closer to the target. Furthermore, these results are consistent for all linear functions chosen as targets.

As mentioned above, we played the suggested games with MCTS to verify that the games suggested by NTBEA provide the desired target score trends not only for the agent that was used during evolution (RHEA) but also for a different one. This is analogous to the procedure done for Aliens in Section 3.2. For doing this validation, the best individual of each run was selected and played 10 times. The score from these games were recorded, averaged for the same parameter set, and plotted along with others in the same evolution configuration. Figure 10 shows the score trends of the best individuals found in all evolutionary runs for the three games. To save space, we are only showing the plots for the target function $y = x$ (the figures of the other linear functions are very similar).

It is worth highlighting that most of the recommended individuals for Defender (Figure 10, right) are games for which MCTS achieves a positive score on average (in contrast with the initial negative score trend seen in the first generations during evolution). The target $y = mx$ with $m = 1$ seems to be, however, hard to approximate in this game. in contrast, games suggested for Waves and Seaquest provide better results, showing score trends in which the MCTS agent approximate the target progression.

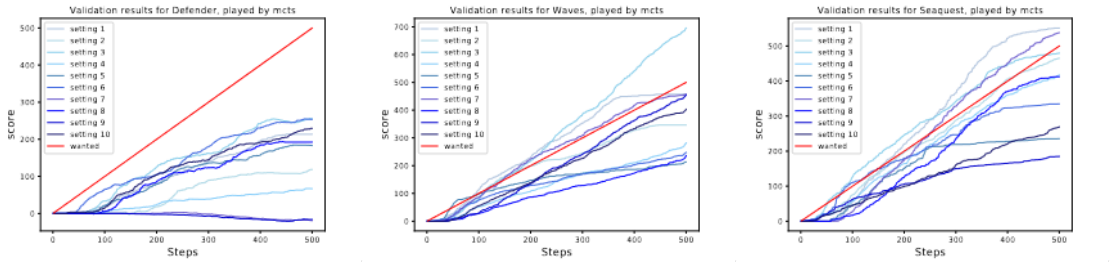

Fig. 10: Average score trend on validation for $y = x$ on the games of this study (left to right: Defender, Waves and Seaquest).

Fitting advanced score trends Figures 11 and 12 shows the fitness progression and score trends when the left-shifted (respectively right-sifthed) sigmoid functions are used as targets.

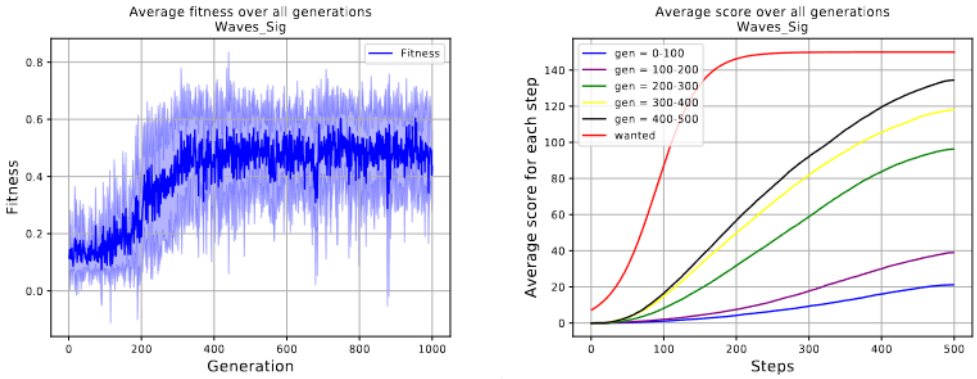

Fig. 11: Fitness and average score trends for the (left) shifted sigmoid function $y = \frac{150}{1+\exp(-\frac{x}{20}+3)}$ in Waves.

Both figures show, on the left, the fitness progression of the NTBEA runs and, on the right, the curve progression as depicted in generation segments. The fitness progression suggests that the left-shifted target is more difficult to adjust that the right-shifted one: the initial fitness of the former is worse and it reaches a constant value of 0.5 (optimum is 1.0), while the latter converges to 0.8.

The left-shifted function requires a game that progresses from providing no score chances to many very early in the game, while the right version of this function shifts this score opportunity change to later in the game. This causes that, in the former

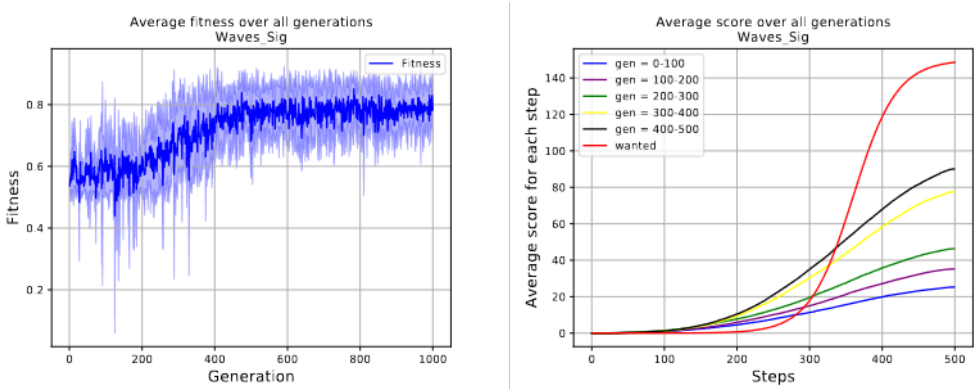

Fig. 12: Fitness and average score trends for the (right) shifted sigmoid function $y = \frac{150}{1+\exp(-\frac{x}{20}+12)}$ in Waves.

case, the average score achieved is higher than in the latter trend, as more game ticks are available to score. This can be clearly seen in both figures 11 and 12, right plots. For instance, in the middle game (time step around 250), the score achieved in the left-shifted case is higher than its counterpart. Similarly, the individuals of the right-shifted sigmoid case evolved to provide fewer score opportunities in the first half of the game.

It can be seen how, starting from a similar trend in the first generation segment (blue lines), the left-shifted trend evolves progressions with higher scores, while the right-shifted version stays at low values. The similar starting point is expected (initial random parameters) and it is clear that NTBEA manages to find parameter sets with very differentiated trends in the end, suggesting that this method is general and can adapt to different target functions from the same starting point.

Validation of these games is again performed using MCTS to play the games suggested by NTBEA. Figure 13 shows the score trends achieved by MCTS in these games for the left-shifted (left figure) and right-shifted (right) target functions.

As can be seen, NTBEA is more successful at evolving games for the right-shifted function. Acknowledging some evolved games fail dramatically at providing the desired score trend in both cases (horizontal blue lines in Figure 13), most of the games evolved for the right-shifted progression adjust better to the desired progression than their counterparts of the left-shifted curve. For this one, the agents achieve (in the best case) close to linear score trends. This suggests again that the left-shifted sigmoid target is harder to approximate, and this actually is understandable: the game must start with no score opportunities to quickly provide more, but stabilizing again in less than 200 game steps at a maximum score. Again, it is worth remembering that

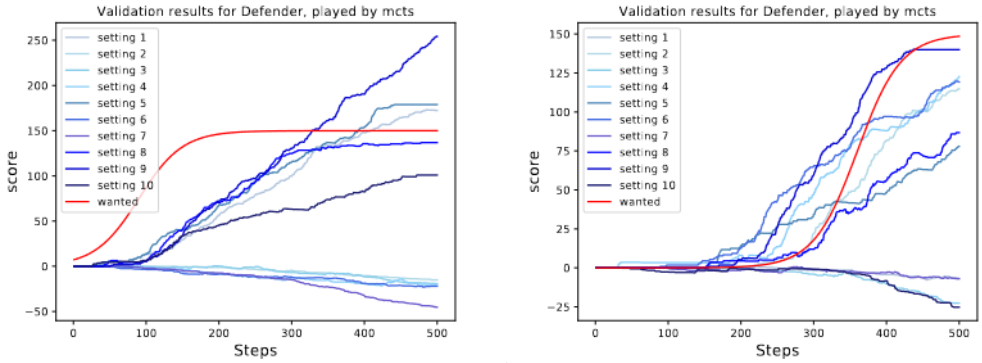

Fig. 13: Average score trend on validation for Defender, shifted sigmoid target functions $y = \frac{150}{1+\exp(-\frac{x}{20}+3)}$.

the target trends may be not achievable by the agents in the game spaces available - they must be understood as guides for evolution.

The results for the logarithmic and exponential target functions are very similar to the ones obtained for the sigmoid ones. Again, we plot the validation results in the game Defender, which has shown to be the most challenging game of the three. Figure 14 shows the results of average score for Defender, targeting the logarithmic (left) and exponential (right) score progression functions.

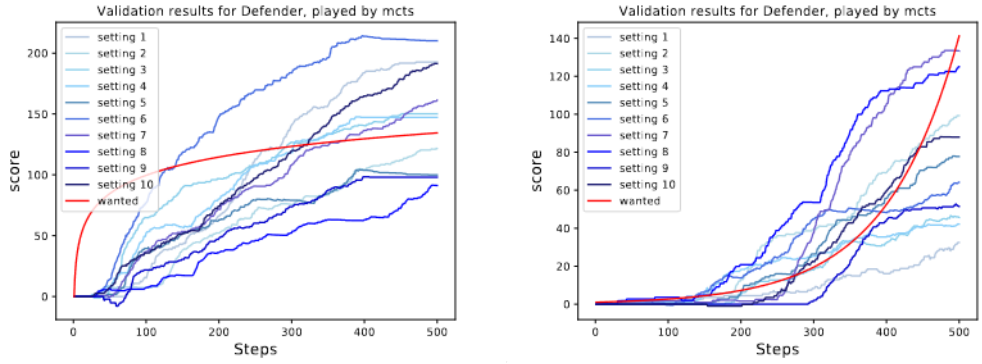

Fig. 14: Average score trend on validations for the game Defender, logarithm and exponential target functions ($y = 15\log_2(x)$ and $y = 2^{\frac{x}{70}}$, respectively).

In this case, the logarithmic trend again poses more problems to find games that adhere to the target curves than the exponential one. The reason for this is the quick and sudden requirement for score in the first steps of the game (it requires to achieve 35 points in the first 5 game steps). However, it is clear that the evolved games behave similar under the same MCTS agent that plays them: the trends shown in both images from Figure 14 are noticeably different.

We observed the values of the parameters evolved by NTBEA in Defender, the game that has shown to be the hardest of the three studied here. In linear functions, a higher slope ($m = 1$) in the trend tended to produce games with a higher supply limit, slower bombing speed and alien spawn probability. Also, the alien spawn portal stops creating enemies earlier on higher slopes than in the others.

Observing values evolved by NTBEA for the advanced target functions in Defender, we can see that the supply amount, alien movement speed and alien spawning rate are higher in the right-shifted sigmoid function than in the left-shifted one. Portals used for spawn aliens open and closed later in the exponential version, providing a slower supply speed and faster alien movement.

Examples of the recommended Defender games can be found in an online video[3]. These show some games evolved for each one of the target function families (linear, sigmoids, logarithmic and exponential), where the differences can be easily observed for each setting.

References

1. A. Isaksen, D. Gopstein, J. Togelius, and A. Nealen, "Discovering Unique Game Variants," in *Computational Creativity and Games Workshop at the 2015 International Conference on Computational Creativity*, 2015.
2. A. Isaksen, D. Gopstein, and A. Nealen, "Exploring Game Space Using Survival Analysis," in *FDG*, 2015.
3. K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas, "The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement," in *IEEE Proceedings of the Congress on Evolutionary Computation (CEC)*, 2017, pp. 2201–2208.
4. J. Liu, J. Togelius, D. Pérez-Liébana, and S. M. Lucas, "Evolving Game Skill-Depth using General Video Game AI Agents," in *IEEE Proceedings of the Congress on Evolutionary Computation (CEC)*, 2017.
5. S. M. Lucas, J. Liu, and D. Perez-Liebana, "The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation," *arXiv preprint arXiv:1802.05991*, 2018.
6. ——, "The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation," *arXiv preprint arXiv:1802.05991*, 2018.

[3] `https://youtu.be/GADQLe2TiqI`