Noname manuscript No.
(will be inserted by the editor)

Creating Autonomous Agents for Playing Super Mario Bros Game by Means of Evolutionary Finite State Machines

A.M. Mora · J.J. Merelo · P. García-Sánchez · P.A. Castillo · M.S. Rodríguez-Domingo · R.M. Hidalgo-Bermúdez

Received: date / Accepted: date

Abstract This paper proposes the design and improvement of an autonomous agent based in a mixture between behavioural models (Finite State Machines, FSMs) and evolutionary methods (Genetic Algorithms in this case), consisting on the consideration of FSMs as part of the individuals to evolve. This leads to finally obtain a so-called *bot*, which is able to autonomously complete different scenarios on a simulator of Super Mario Bros. game. The bot should follow *Gameplay* track rules of the international Mario AI Championship. Mono- and multi-seed approaches (evaluation in one play or multiple plays respectively) have been analysed, considering the machine resources consumption, which turns in a bottleneck in some experiments. However, the methods yield agents which can finish several stages of different difficulty successfully, and which play much better than an expert human player. According to the results and considering the competition's restrictions and objectives, these agents would have enough performance to participate in this competition track.

Keywords Videogames · Super Mario Bros · Artificial Intelligence · · Genetic Algorithms · Finite State Machines · Autonomous Agents · Non Player Characters · Bots

A.M. Mora, M.S. Rodríguez-Domingo, R.M. Hidalgo-Bermúdez,
J.J. Merelo, P. García-Sánchez, P.A. Castillo
Departamento de Arquitectura y Tecnología de Computadores.
ETSIIT-CITIC.
University of Granada, Spain.
Tel.: +34-958241778
Fax: +34-958248993
E-mail: {amorag,jmerelo,pgarcia,pedro}@geneura.ugr.es
rosa.hb84@gmail.com, zandri@gmail.com

1 Introduction

Videogames have considerably evolved from their original concept to the complex technological products that they have become, creating a massive industry and market as important as the cinematographic one. Thus, in recent years, most of the game development has been focused on the technical part (graphics and sound), leaving the Artificial Intelligence (AI) aside. However, nowadays artificial intelligence (specifically Computational Intelligence, CI) is becoming more relevant, since videogames players are requesting for opponents which mean a real challenge, and so provide a better gaming experience. This is leading to new research lines on how to create non-playing characters with adapted and unpredictable behaviour, i.e. more human-like in some sense.

Going back to the first modern video games, and focusing on one of the most prolific characters, in 1981 Shigeru Miyamoto[1] created Donkey Kong, in which a plumber named Jumpman tries to rescue his girlfriend from a gorilla. This character was evolved, and in 1983 (thanks again to Miyamoto) Mario Bros. games series appeared. The most famous was platform game Super Mario Bros., which was launched in arcade and also in the Nintendo's home console NES. Several other sequels have been published, including the blockbuster Super Mario World in the 90s, which had a great improvement in graphics, sound and playability.

All these games present the plumber Mario who must rescue the princess of the Mushroom Kingdom, Peach, kidnapped by the king of the koopas, Bowser. The gameplay consists in moving across lateral platforming levels (stages), trying to avoid different types of enemies and obstacles and using some items, such as mushrooms (Mario grows) or fire flowers (Mario can shot fireballs).

Videogames have become one of the most extended research areas regarding the Artificial Intelligence (AI) field, yielding the so-called Computational Intelligence (CI) branch. Several games have become extended frameworks for study new techniques and algorithms in these scopes, such as Quake [13], Unreal [23,16], Pac-Man [14,20], Starcraft [25,11] and, of course, the commented Super Mario Bros. [19,26,3,22].

Thus, there exists a framework developed for the latter called *Mario AI*, a modified version of the game known as Infinite Mario Bros.[2], an open source application where the researchers can implement, for instance interactive and autonomous behavioural routines (or agents), using the set of functions and variables which it offers. Moreover, there is an additional motivation for the scientific community to perform these studies, which is the competition *Mario AI Championship*[3], held three times a year, inside several famous CI conferences and workshops. There are some tracks in it: Learning, Level generation, Turing test, and Gameplay. The latter is devoted to create an autonomous

[1] Designer and producer of Nintendo Ltd., and winner of the 2012 Príncipe de Asturias Prize in Humanities and Communication

[2] `http://www.mojang.com/notch/mario/`

[3] `http://www.marioai.org/`

agent, also called *bot*, which must perform the best as possible in automatically playing and pass sequential levels (stages) with a growing difficulty.

This work is enclosed in this track, and presents different approaches of autonomous agents, which follow the behaviour modelled by means of a Finite State Machine (FSM) [4]. This behavioural engines are generated as part of the individuals in the population of an Evolutionary Algorithm (EA) [1], starting from simple states and combining them to create wide sets of outputs in every state. Thus, this is a shape of evolutionary FSMs, which is, to our knowledge, a novel approach inside this scope.

This EA, a Genetic Algorithm (GA) [10], has been applied offline (not during game), as have been done in many CI researches in the videogames scope [24,15,5,7]. Two different approaches have been considered, mainly experimenting with different evaluation schemes: *mono-seed evaluation*, in which every individual plays just one level/stage and just one time; and *multi-seed evaluation*, which considers 30 plays in 30 different stages (generated with a different randomisation seed).

The approaches have been widely tested and analysed, getting an optimal set of parameters for the EA and thus, very competent agents in a number of difficulty levels, which could participate successfully in the competition. However, these approaches (much more i the case of multi-seed) demand a high amount of computation resources and machine memory, which means it is not possible to optimise in any difficulty level, due to the huge amount of inputs, rules and outputs to consider in the evolution.

The rest of the paper is organised as follows. Next section presents some preliminary concepts and background of the work. Section 3 defines the problem itself, describing the Infinite Mario Bros. platform, along with the competition rules, regarding the main features that the agent must consider and its constraints. Then, Section 4 introduces the agents' approaches which will be analysed in the paper, along with the set of experiments conducted to perform this analysis in Section 5. Finally, Section 6 describes the reached conclusions and proposes some future lines of work.

2 Preliminary concepts and background

In this section, the main techniques applied in the development of this work are briefly described.

2.1 Genetic Algorithms

Evolutionary Algorithms (EAs) [1] are a class of direct, probabilistic search and optimisation algorithms gleaned from the model of darwinistic evolution. They have been widely used for solving combinatorial and optimisation problems. An EA works with a population of possible solutions (individuals) for the target problem, a selection method that favours better solutions and a set

of operators that act upon the selected solutions. After an initial population is created (usually randomly), the selection and operators are successively applied to the individuals in order to create new populations that replace the older one. This process guarantees that the average quality of the individuals tends to increase with the number of generations. Eventually, depending on the type of problem and on the efficiency of the EA, an optimal solution may be found. The most extended class of EA are the Genetic Algorithms (GAs) [10]. A GA is composed of a *population of candidate solutions* to a problem that evolve towards optimal (local or global) points of the search space by recombining parts of the solutions to generate a new population. The decision variables of the problem are encoded in strings with a certain length and cardinality. In GAs' terminology, these strings are referred to as *chromosomes*, each string position is a *gene* and its values are the *alleles*. The alleles may be binary, integer, real-valued, etc, depending on the codification (which in turn may depend on the type of problem). As stated, the "best" parts of the chromosomes (or building-blocks) are guaranteed to spread across the population by a *selection mechanism* that favours better (or fitter) solutions. The quality of the solutions is evaluated by computing the *fitness* values of the chromosomes; this fitness function is usually the only information given to the GA about the problem.

EAs have been applied in a wide range of optimisation problems, including in recent years (10 years ago) their application in videogames area [12, 21, 26, 23, 16]. The problem is they usually involve considerable computational cost and thus they are not frequently used in real-time. There is a proposal of use of EAs online (during the game) by Fernández et al. [8], which applied these algorithms in an action game for pathfinding tasks, but it is quite difficult to use them for more complex computations in real-time. Thus, the most successful proposals for using EAs in games correspond to offline applications [24, 15, 7], that is, the EA works (for instance, to improve the operational rules that guide the bot's actions) while the game is not being played (before it), and the results or improvements can be used later during the game. Through offline evolutionary learning, the quality of bots' intelligence in commercial games can be improved, and this has been proven to be more effective than opponent-based scripts. This way, in this work, an offline EA is applied to a parametrised behavioural model for a Super Mario agent, in order to improve its decision engine, which will be used later during game.

2.2 Finite State Machines

A Finite State Machine (FSM) [4] or Finite State Automaton is a computational model which represents a set of *states* and connections between them. Every connection correspond to a *transition* from one state to another one, depending on the state inputs and the so-called transition function.

It is represented as a directed graph (see Figure 1), where the states are nodes labelled with a name, and the transitions are edges with an associated

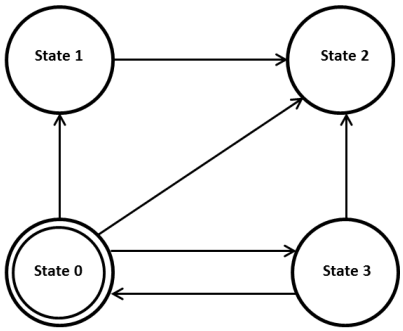

Fig. 1 Example of FSM with four states (circles) and some transitions between them (directed arcs).

value representing the input that should be received to move between states through this transition. The graph can be connected at different degrees (is not always fully connected), depending on the possible transitions to model.

There is usually an initial state, where the model starts working, and a final state, which is the final output of the automaton.

FSMs were initially designed to recognise regular and formal languages, but nowadays they are used as a powerful behaviour modelling tool inside videogames, i.e., the AI of an autonomous character can be defined by means of a FSM. They are based in *sensors*, which receive environmental inputs and *actuators*, which perform the response actions to these inputs. In the FSM model, the possible values received by the sensors are set in the edges of the graph (as transitions), meanwhile the actuators' actions happen in every state. Several videogames have implemented this technique [9,18], from Pac-Man (modelling the ghosts' behaviour) to Unreal Tournament series, which is one of the best considered games regarding its Bots (agents) AI. Also the authors have previously researched in the evolution of FSM-Based behavioural models [6], but in the scope of the Unreal Tournament game.

In the present work, the behavioural model for our Mario agent has been implemented by means of a FSM, since it can easily model (or simulate) the behaviour of an expert player.

3 Mario AI: Environment and Competition

3.1 The Environment

The game consists in moving the character, Mario, through bi-dimensional levels (stages). He can move left and right, down (crouch), run (letting the button pushed), jump and shoot fireballs (when in "fire" mode).

The main goal is complete the stage, whereas secondary goals could be killing enemies and collecting coins or other items. These items may be hidden

Fig. 2 Infinite Mario screen capture showing a level/stage, and the information provided by the Mario AI simulator.

and may cause Mario to change his status (for instance a fire flower placed 'inside' a block). The difficulty of the game lies in the presence of cliffs/gaps and enemies. Mario loses power (i.e., his status goes down one level) when touched by an enemy, and dies if he falls off a cliff or if he is touched in the smaller status. Figure 2 (left) shows a screen capture of the game.

Mario can be in three different modes:

- *Small*: In this mode Mario is smaller than in the others. If an enemy strikes him, Mario dies. He can't crouch.
- *Big*: This is Mario's intermediate mode. Mario reaches this mode if he is in Fire state and an enemy touches him, or devouring a mushroom in the Small state.
- *Fire*: In it Mario can shoot fireballs. Mario can reach this mode if he takes a fire flower (it only appears if Mario is in Big mode).

The simulator provides information about Mario's surrounding areas (See Figure 2 (right)). According to the rules of the competition, two matrices give this information, both of them are 19x19 cells size, centred in Mario. One contains the positions of surrounding enemies, and the other provides information about the objects in the area (items and stage features such as gaps, blocks, platforms, etc).

Every tick (40 ms), Mario's next *action* must be given. This action consists in a combination of the five possible movements that Mario can do (left, right, down, fire/speed, jump). This information is encoded into a boolean array, containing a true value when a movement must be done.

The action to perform depends, of course, of the stage characteristics around Mario, but it is also important to know where the enemies are and their type. Thus, the agent could know if it is best to jump, shoot or avoid them. We have defined four main enemies groups according to what the agent needs to do to neutralise them:

- Enemies who can die by a fireball strike or if Mario jumps on them and crushes them or if Mario is carrying a koopa shell and throws it at the

enemies: Goomba, Goomba Winged, Green Koopa, Red Koopa, Koopa Winged Green, Red Koopa Winged.
- Enemies who only die by a fireball strike: carnivorous flowers that appear in the pipes.
- Enemies who only die when Mario jumps on them and crushes them: the cannon balls.
- Enemies who only die if Mario launches a koopa shell against them: the Spiky and Spiky Winged.

3.2 The Competition

The proposed agents follow the rules of the Mario AI Championship[4] devoted to address different CI tasks (commented in the following paragraph).

Specifically, Mario AI is a simulator, a version of Infinite Mario Bros., which is available in open source and make it easier for the competitors to implement their algorithms, using a set of classes, variables and high-level functions. It includes a wide support for implementing autonomous agents to control Mario character using AI techniques.

The competition consists of four categories:

- Gameplay: looking for the best Mario agent (bot), i.e. the one who can pass a higher amount of stages with growing difficulty.
- Learning: the agent must learn to play a particular stage in a limited time.
- Level Generation: generation of levels/stages which should be as fun as possible (according to human players).
- Turing Test: the agent must behave as a human.

Even being a relatively new competition, it has already been quite successful and had numerous participants with very interesting results and algorithms. For example the work by Bojarski and Bates-Congdon, called REALM [3], in which an agent based on an evolutionary set of rules is developed, specifying some goals to achieve starting from a set of conditions and actions. This agent won the competition in the categories of Gameplay and Learning in the CIG 2010 edition.

In the category of Gameplay the main goal is to complete as many stages as possible. The problem can be interpreted as a pathfinding one, as Baumgarten [2] did. He solved the objective of finding the fastest way through the surroundings in which Mario stays alive by means of an A* searching. This algorithm has proved to be quick and flexible enough to find routes through the generated stages even in real-time.

Our aim has been implementing an intelligent agent which could compete in the Gameplay track. In order to do this, we have applied GAs along with behavioural models (FSMs). GAs have been chosen due to its adaptability, as with them we do not need specific knowledge of the problem that we want to solve; they operate with diverse simultaneous solutions to the problem, and

[4] `http://www.marioai.org/`

they allow to optimise these solutions regarding to different goals. Even though its convergence is not as fast as other more specific algorithms for the problem we want to tackle, in most of the cases they can find a very competent solution.

The comparison between a FSM-based approach and a rule-based method such as the commented REALM [3] is the flexibility that the states provide. The rules model a strict and quite limited set of conditions and actions, on the other hand FSMs are based in a set of situations in which the agent can be. Then, in every situation, a set of events could happen, corresponding these events to a group of rules which perform different actions depending on the state where the agent is. This behavioural model is closer to the human rationale and thus it would be more suitable for implementing human-like behaviour.

The difference of the proposed method with a searching technique as A* is that the former is an offline (before the game) optimization approach, while the latter is an online (during the game) method. The reason why the chose offline optimisation is that it can obtain better results for a previously trained set. Thus, agents created by means of evolutionary optimisation can learn to behave in a wide range of different events, if they are trained during enough time and in enough different situations. In addition online techniques could have a bad performance if the difficulty level is increased, for instance, because the amount of possibilities (enemies, gaps, etc) would grow exponentially and the time to react is limited in order to simulate real-time decisions (40ms in Mario AI Competition).

Related to the Gameplay track and regarding the application of EAs, the work by Togelius et al. [26], combined this metaheuristic with Neural Networks in order to evolve Super Mario controllers that can complete low difficulty levels.

The scope of the Mario AI competitions has also be profitable in other areas/tracks, such as the Level Generation, in which Shaker et al. used evolutionary algorithms to improve automatic level designs [22], or in which Pedersen et al. [19] analysed the player experience (satisfaction, frustration, etc) regarding the level design parameters (number and positions of items, gaps, enemies and so on).

4 Evolutionary FSM-based Agents

We have named the objective agent to obtain *evoFSM-Mario*, since it is based in a FSM which models a logical behaviour, such as expert player knowledge. This tool has been combined with EAs (GAs), given their excellent adaption and optimisation capabilities, which are very useful for improving predefined behavioural rules.

The FSMs will model a set of limited states for the agent, which correspond to all the possible (valid) actions an agent can perform in a specific moment, i.e. feasible combinations of Mario's basic movements and actions, i.e. moving left, moving right, crouch (going down), jump and fire/run. Table 1 shows a

proposed codification of the states in boolean values, in order to deal with them in the GA.

Table 1 Codification of the feasible states of the FSM which will model the Mario agent's AI. 1 is $true/active$, 0 is $false/non-active$.

	Right	Left	Fire/Run	Jump	Down
State 0	1	0	0	0	0
State 1	1	0	0	1	0
State 2	1	0	1	0	0
State 3	1	0	1	1	0
State 4	0	1	0	0	0
State 5	0	1	0	1	0
State 6	0	1	1	0	0
State 7	0	1	1	1	0
State 8	0	0	0	0	0
State 9	0	0	0	0	1
State 10	0	0	0	1	0
State 11	0	0	0	1	1
State 12	0	0	1	0	0
State 13	0	0	1	1	0

Depending on an input, the state changes (or remains), so the transition is decided. Inputs are all the possible situations an agent (Mario) can observe around him in a game tick (environment). They are included in the observation matrixes that the simulator yields. Considering the dimensions of these matrixes (19x19 cells) and the different values (objects) they can contain (enemies, gaps, blocks, items, platforms, etc).

In order to deal with this enormous set of situations, these have been simplified, considering a vector/string in which some situations have been converted into 'fuzzy' values; for instance enemies' positions with respect to Mario are stored as *right*, *right-top*, *right-down*, and so on, in a shape of cardinal points. The distances are also considered as 'fuzzy' being *close*, *near*, *medium*, *far*, etc. Thus, these values are taken just as simpler references, instead of having absolute floating point positions.

The agents will manage input strings, which are sets of values being *true (1)* if a specific situation has happened.

Even though there could be a wide number of different inputs, which grows with the difficulty of the stage to be passed by the agent to obtain; i.e. there are more enemies and gaps, for instance, in difficult stages than in easier ones.

These feasible states along with the possible inputs and transitions will be evolved by means of the GA. It should be noticed that the output state for a new entry would be randomly set, however it will be done according to a set of probabilities which models the preference of the states. These probabilities are based in the experience of an expert player (one of the authors has been playing Super Mario games for more than 20 years), and model the chance of happening each one of the situations. This is made in order to improve the convergence of the algorithm, because in a preliminary set of experiments a

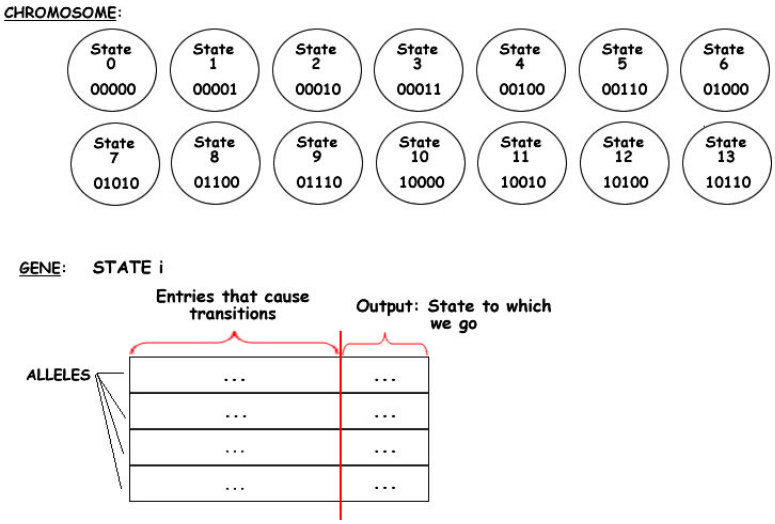

Fig. 3 Chromosome codification to evolve FSMs by means of a GA.

purely random output assignment meant a very low convergence rate and thus, a very time-consuming algorithm. Thus, the 14 feasible states are grouped in three possible categories with a probability of being assigned:

- States where Mario does not advance to right or left. This category has associated a 10% probability.
- States where Mario advances to left, with a 20% probability.
- States where Mario advances to right, with a 70% probability.

Due to the huge search space (hundreds of possible inputs), a parameter indicating the percentage of new individuals that will be added in each generation is included. This parameter tries to control the diversity rate.

Every individual in the population is represented by a set of tables, one per state, and every table contains an output state for every possible input (i.e. the transition). Figure 3 shows a chromosome/individual. The chromosome is the set of tables, a gene corresponds to one state, and the alleles are each one of the possible inputs (or entries) and transitions (next state). See Figure an example in 4.

The *fitness function*, as usually in EAs, is the most important factor, since the algorithm performance strongly depends on it. To calculate this function for one individual, the FSM represented in a chromosome is set as the AI of one agent which is placed in a game stage, and then, it plays for getting its fitness value. Two different schemes have been implemented:

- *mono-seed*: Every individual is tested in the same stage (with the same difficulty) one time, until it passes the stage, dies or gets stacked (do not get the end of the level on time). The length of the stage and thus the time to complete it grow with the generations, being easier in the first

generations and harder when the algorithm run advances. The aim is to improve the convergence. If the individuals in a generation do not obtain a good enough fitness on average, the stage length and available time remain the same in the next generation.
- *multi-seed*: Every individual is tested in 30 stages (with the same difficulty) generated randomly using different seeds, which are the same for all the individuals. It plays, as previously, until complete the stage, die or get stacked. The fitness is computed considering the results of all the plays for the individual. The aim of this scheme is: 1) to avoid the usual noise [17] present in this type of problems (videogames), i.e. it tries to get a fair valuation for an individual, since the same configuration could represent an agent which is very good sometimes and quite bad some others, due to the stochasticity of every play (regarding the agent's behaviour); and 2) to obtain individuals prepared to face a wide set of situations in every stage and difficulty, since 30 stages should present a high amount of different scenarios and configurations.

Thus, there is a *generic fitness* which has as restriction completely finish the stage to be set to positive. On the contrary, individuals that have not finished the stage start from the lowest fitness possible and their negativity is reduced according to the behaviour during the stage run. This generic fitness is a weighted aggregation, similar to the one implemented inside the simulator to assign the score, and which is based in the next values:

- *marioWinner*(win): is set to 1 if the agent finish the stage.
- *marioStatus*(stat): 0 if the agent is small, 1 if it is Big, and 2 if it is Fire.
- *numKilledEnemies(enems)*: as the name suggest, is the number of killed enemies by the agent.
- *numTotalEnemies(numEnems)*: number of total enemies during the stage.
- *passedCells* (cells): number of cells passed by the agent.
- *totalCells* (totCells): total number of cells in the stage.
- *remainingTime* (remTime): time to finish the stage.
- *spentTime (speTime)*: time spent by the agent.
- *totalTime (totTime)*: total time to complete the stage.
- *coinsGathered* (coins): number of gathered coins.
- *totalCoins (numCoins)*: total coins appeared in the stage.

STATE i

Input	Next State
ENEMY (Winged Goomba) - left_top - far	5
ENEMY (Red Koopa) - right - near	7
ITEM (Coin) - top - close	2
GAP - far	10
...	...

Fig. 4 Example gene corresponding to State i. The different inputs are the alleles, which have associated the outputs or next states, modelling the transitions in the FSM.

- *numCollisions (numCollis)*: number of times the agent has bumped with an enemy.
- *numGatheredPowerUps (powers)*: number of mushrooms or flower the agent has picked up.
- *causeOfDeath (cDead)*: in case the agent is dead, this value is also taken into account in the fitness computation.

The fitness function in the Mario AI simulator is defined as:

$$Fitness = win{\cdot}1024{+}stat{\cdot}32{+}enems{\cdot}42{+}coins{\cdot}16{+}remTime{\cdot}8{+}cells{\cdot}1 \quad (1)$$

The proposed fitness function combines all the previous values, in a weighted sum, where the weights have been set taken those in Equation 1 as a reference in order to assign the relative of every term. The *generic fitness function* is:

```
IF (win == 1)
```

$$\begin{aligned} Fitness = win \cdot 200 + stat \cdot 300 + (powers - numCollis) \cdot 200 \\ +(enems/numEnems) \cdot 300 + (coins/numCoins) \cdot 50 \\ +(remTime/totTime) \cdot 50 \end{aligned} \quad (2)$$

```
Else
```

$$\begin{aligned} Fitness = MIN\_FITNESS + (cells/totCells) \cdot 600 \\ +((cells/totCells) - (speTime/totTime)) \cdot 200 \\ +(enems/numEnems) \cdot 150 + (coins/numCoins) \end{aligned} \quad (3)$$

Where MIN_FITNESS is the negative bound of the fitness values. As it can be seen in Equation 3 if the agent (individual to evaluate) has not success (falls down or dies), the fitness is set as negative and increased with the positive objectives reached by the agent (gathered items, killed enemies, passed cells).

Considering this generic fitness function, the evolved agents will be improved to maximize this score. However the aim in this track of the competition is firstly finish the stages and thus, the most important factor in the function is that Mario has completed the scenario. In addition, since the obtained agents have been evolved considering a big amount of situations (inputs), they will be very competent in solving almost any stage of the same difficulty level to the one for which they were optimised, above all the multi-seed solutions, which were evolved using a set of different stages every generation. So, taking into account these facts, the evolved individuals could be considered as versatile agents: able to complete every stage (in the same difficulty), and also maximize the most common score functions in platforming games, and specifically in Super Mario Bros. videogames: remaining time and (sometimes) number of collected coins.

The applied GA operators are described in the following paragraphs. The *selection mechanism* takes into account a population ordering regarding the

fitness. Thus it is considered as the result of the individuals' evaluation in the mono-seed approach, meanwhile multi-seed considers a *hierarchical fitness* ordering, where the population is ordered according to the next criteria: First, taking into account the percentage of stages where the individuals have been stacked or fallen from a cliff. Then, they are ordered considering the average percentage of stages completed. Finally, the individuals are ordered by the average generic fitness obtained in the stages.

The reason for this selection is that, according to several previous experimentation with other approaches (such as the most successful before), it is quite common that a very good individual in one stage (with a very high generic fitness value), falls down or gets stacked in the following stage. Thus, the less worse individuals are chosen.

The selection then considers as parents, on one side the best individual, and on the other side a percentage of the best ones. They are selected by tournament according to their fitness, generic or hierarchical, depending on the approach. Thus, when comparing two individuals in the multi-seed scheme, three values are considered in cascade: percentage of stages where the agent has been stacked or fallen in a gap, percentage of stages completed, and generic fitness.

Crossover is performed considering the best individual of the present generation as one of the parents, and one of the individuals with positive fitness as the other parent. They generate a number of descendents which depends on the percentage of population to complete with the crossover. If there are not enough parents, we select the 'less worse' individuals of those with negative fitness. A uniform crossover has been used, i.e. each gene is randomly selected from one of the two parents.

The *Mutation operator* is slightly different from the usual one: after selecting a percentage of individuals to be mutated (mutation rate), various genes in each of these individuals are randomly selected to be mutated. Mutation is performed by randomly changing the output state for an input in the table.

Regarding the *replacement* to form the new population, there is a $1 - elitism$ mechanism, so the best individual survives. The rest of the population is composed by the offspring generated in the previous generation, which was a percentage of the global population, and the rest of individuals being random ones, in order to increase the diversity, and thus, the exploration. The percentage defined, in fact, is related to the number of random individuals, which is the complementary to the percentage of the generated offspring. This value is reduced with the generations in the mono-seed approach. The aim is getting a high diversification factor in the first generations, i.e. an exploration phase, and reduce this factor when the best solutions arise (in some generations), changing to an exploitation phase. In multi-seed approach the percentage remains constant (quite small), because the fitness is much more representative of the individuals' quality, so the exploration has lower significance.

5 Experiments and Results

The aim of the proposed methods is to find good enough agents for completing any stage in any difficulty level, so in order to test these approaches, several experiments have been conducted.

As an initial step it was performed a hard fine-tuning stage (through systematic experimentation), in which several different values for the algorithms' parameters were tested, looking for the best configuration.

In addition, in the experiments a wide analysis on the influence of these parameters was done, due to the difficulties that the algorithms found from difficulty level 5 onwards, as will be described in the next paragraphs.

It is important to remark that a single play of an agent could spend around 40-50 seconds on average (it depends on the stage length and difficulty level), because it must be played in real-time, not simulated. This fact means that a single evaluation for one individual in the multi-seed approach (30 plays) could take around 25 minutes in some stages.

Moreover, when the experiments were conducted, it could be noticed that the most difficult stages strongly limited the algorithmic behaviour of the approaches, making it hard to converge and even ending the execution abruptly. In addition to the high computational cost (one run may take several days), the huge size (in memory terms) of the structure that stores the individuals' FSM, can make the program finish due to a lack of memory. This structure grows exponentially during the run (new inputs and outputs are added to the tables in crossovers), so in difficulty levels higher than 4, the program frequently crashes.

5.1 Mono-seed approach

The first set of experiments were performed on a Phenom II X4 925 with 16GB RAM. The main problem arose initially in difficulty levels higher than 5, where the memory limitations make the application crash.

Thus, the global FSM structure implementation was redesigned (at the implementation level) to an optimal one:

- Using more appropriate data types for all the values (`Short` instead of `Integer`, for instance).
- Using `TreeMap` instead of `HashMap` to model the tables. Because the latter requires four times more resources than the former.
- Garbage recollection improvement, setting the unused references to `NULL` and explicitly calling to the collector `System.gc()`.

This gave us the chance to evolve agents in all possible difficulty levels, but unfortunately, still with a shorter number of generations than would be recommended.

The parameter analysis yielded the best configuration for the GA, whose values are presented in Table 2.

Table 2 Optimal parameter values for mono-seed approach.

	Optimal value
Population size	1000 (difficulty 0)
Number of generations	30 (difficulty 0)
Crossover percentage	95%
Mutation percentage	2% (individuals)
Mutation rate	1% (genes)
Percentage of random individuals	5% (decreased with the generations)
Fitness function	generic (aggregation)

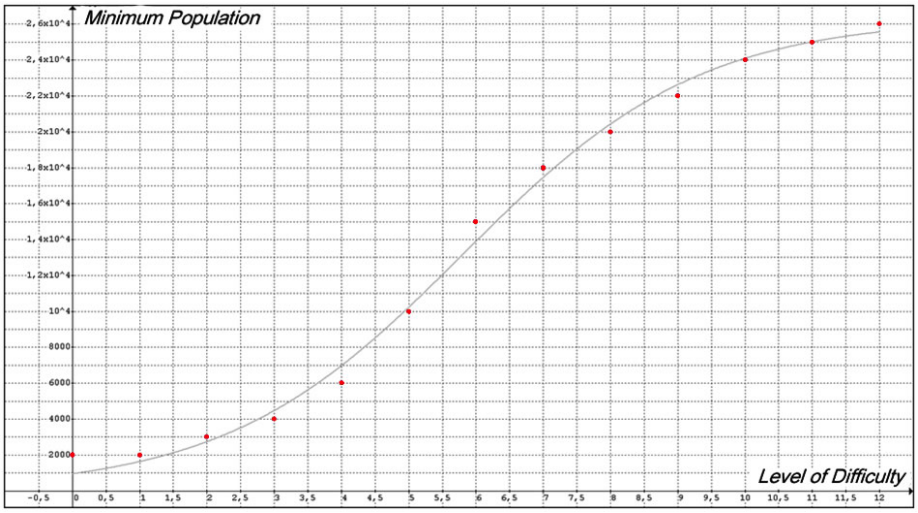

Fig. 5 Population size recommended for each difficulty level in mono-seed approach.

As it can be seen a small number of generations is required to get competent agents in difficulty level 0. The population size should be quite high to ensure the evolution (yield individuals able to complete the stage), since every individual is just tested once in this approach. Thus, the population size was deeply studied, performing several runs for every configuration, pointing out if a successful agent was obtained for each difficulty level. The obtained results are plotted in Figure 5. In this figure it can be seen the huge population sizes required to obtain competent agents in the most difficult levels, due to the evaluation criteria based in just one play per individual.

Moreover, Figure 6 presents the requirements in computation time and memory that a complete evolution (run of the algorithm) needed in every difficulty level. As it is shown, the algorithm is very exigent regarding machine resources.

Figure 7 shows the fitness evolution (in difficulty 0), where the maximum fitness value (of the best individual to the moment) and the number of individuals with positive fitness (required to perform the crossover) are plotted. It can be seen that the fitness grows with the generations (top graph), as ex-

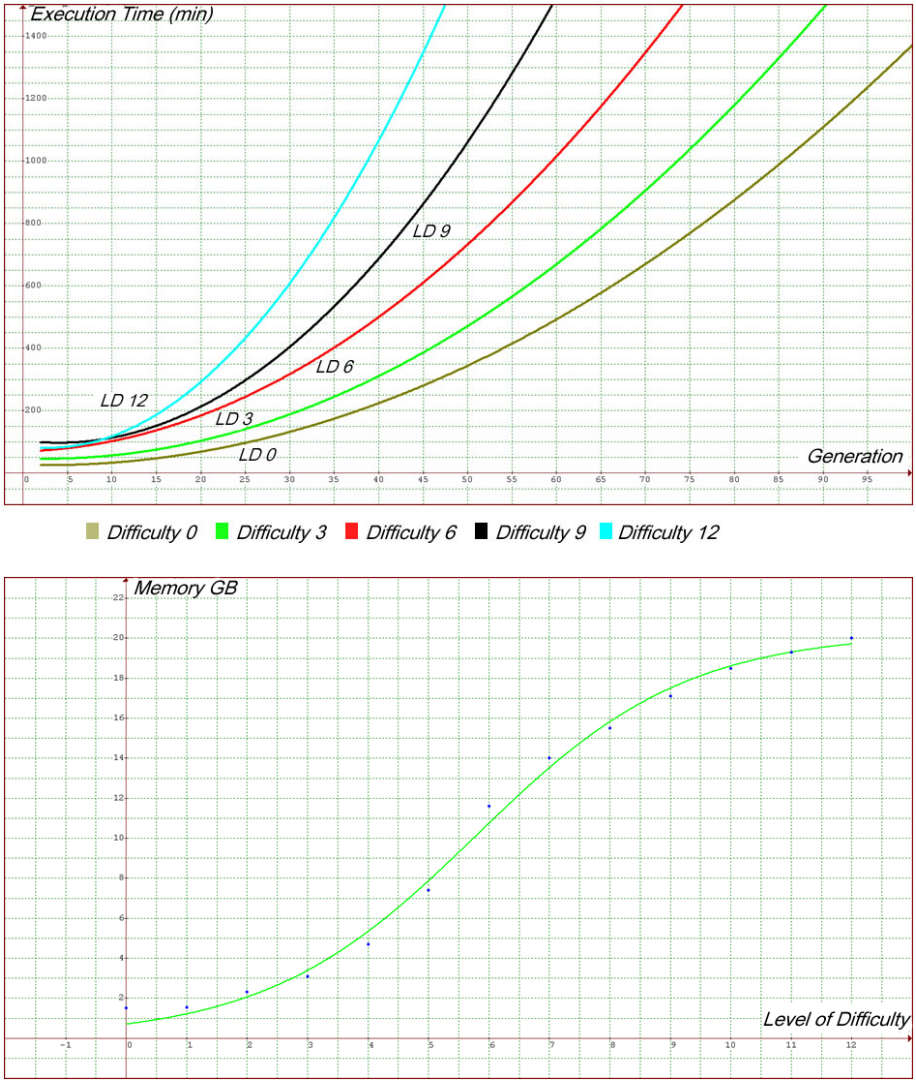

Fig. 6 Run machine requirements: computation time (in minutes) and occupied memory (GB) for mono-seed approach.

pected in a GA, and there are always enough individuals with positive fitness to ensure the offspring generation (bottom graph).

The agents obtained with mono-seed approach are very good for the specific stage (it depends on the generation seed) and difficulty level for which they were 'trained', i.e. those which were set during the optimisation process. They also behave very well in the same stage with different lengths, since the length grows with the generations, as commented in Section 4. However their

Fig. 7 Maximum fitness in every generation, along with the number of individuals with positive fitness in mono-seed approach. Difficulty level 0.

behaviour is quite bad when the stage (generation seed) or the difficulty level are changed.

5.2 Multi-seed approach

Due to the problems of previous experiments, multi-seed study was conducted on a better computer, an Intel Core i7-920 processor with 32GB RAM. However the memory problem still remained. Thus, as a first improvement the number of states where reduced to 12, by deleting those considered as non-useful (in Table 1): State 8 (no action is done) and State 11 (Mario jumps and crouch, since this action could be considered as the less useful). With this change, it was possible to evolve competent agents from difficulty levels 0 to 4, which are, in turn, enough for the Gameplay competition.

As previously stated, in this approach, every individual is evaluated in 30 different stages, with different random seeds and stage type (exterior, subterranean or castle), but with the same length and associated difficulty.

Moreover, in order to get a better computation time, we profited the architecture of the machine (4 processors), so the fitness evaluation was divided into four parallel threads. Thus a different play (a different stage in a different instance of the simulator) was launched, parallelizing the evaluation process.

The optimal values for this approach are shown in Table 3.

In this case, the number of generations is smaller than in mono-seed case for difficulty level 4 (which was set to 6000 in that case). Since in this approach the individuals are evaluated in 30 different stages, so the adaptation higher and thus, a high amount of individuals is not needed.

Table 3 Optimal parameter values for multi-seed approach.

	Optimal value
Population size	2000 (difficulty 4)
Number of generations	500 (difficulty 4)
Crossover percentage	95%
Mutation percentage	2% (individuals)
Mutation rate	1% (genes)
Percentage of random individuals	5% (constant)
Fitness function	hierarchical

However, the time spent and memory required are even higher than before, as it can be seen in Figure 8, where the analysis of memory and computation time consumed in the evolution (until difficulty level 4) is shown. The time consumption was logical, due to the number of evaluations to perform per individual. The memory requirements is explained since every table in each chromosome grows when new inputs (situations) arise in the game, so they must be considered in the FSM. In 30 plays and in medium-difficulty stages, this number grows exponentially.

However, the agents obtained with multi-seed approach are able to complete (almost) any stage in every difficulty level for which they have been optimised.

Some examples of the obtained evolved FSM-based agents can be seen in action (in a video) from the next urls:

- *Mono-Seed - Difficulty level 1 (completed)*:
  `http://www.youtube.com/watch?v=6Pj6dZCE070`
- *Multi-Seed - Difficulty level 2 (completed)*:
  `http://www.youtube.com/watch?v=gtfuY-L0WDA`
- *Multi-Seed - Difficulty level 3 (completed)*:
  `http://www.youtube.com/watch?v=qQVQ43sWwYY`
- *Multi-Seed - Difficulty level 12 (stacked)*:
  `http://www.youtube.com/watch?v=zNGfBApX7sk`

The last one was evolved for some generations (not all the desired) in that difficulty level, due to the commented problems, so it cannot complete this hard stage in the simulator. However, as it can be seen, it is quite good in the first part of the play. Thus, if the complete evolutionary process could be finished in this difficulty level, and following the tendency, the obtained agent would pass almost any scenario. The reason that leads us to think this way is that difficulty level 12 presents a huge amount of situations enough to cover a wide part of the input space.

6 Conclusions and Future Work

This work presents two different approaches of evolutionary-based autonomous agents for playing the videogame Super Mario Bros. They have been implemented using Finite State Machine (FSM) models evolved by means of

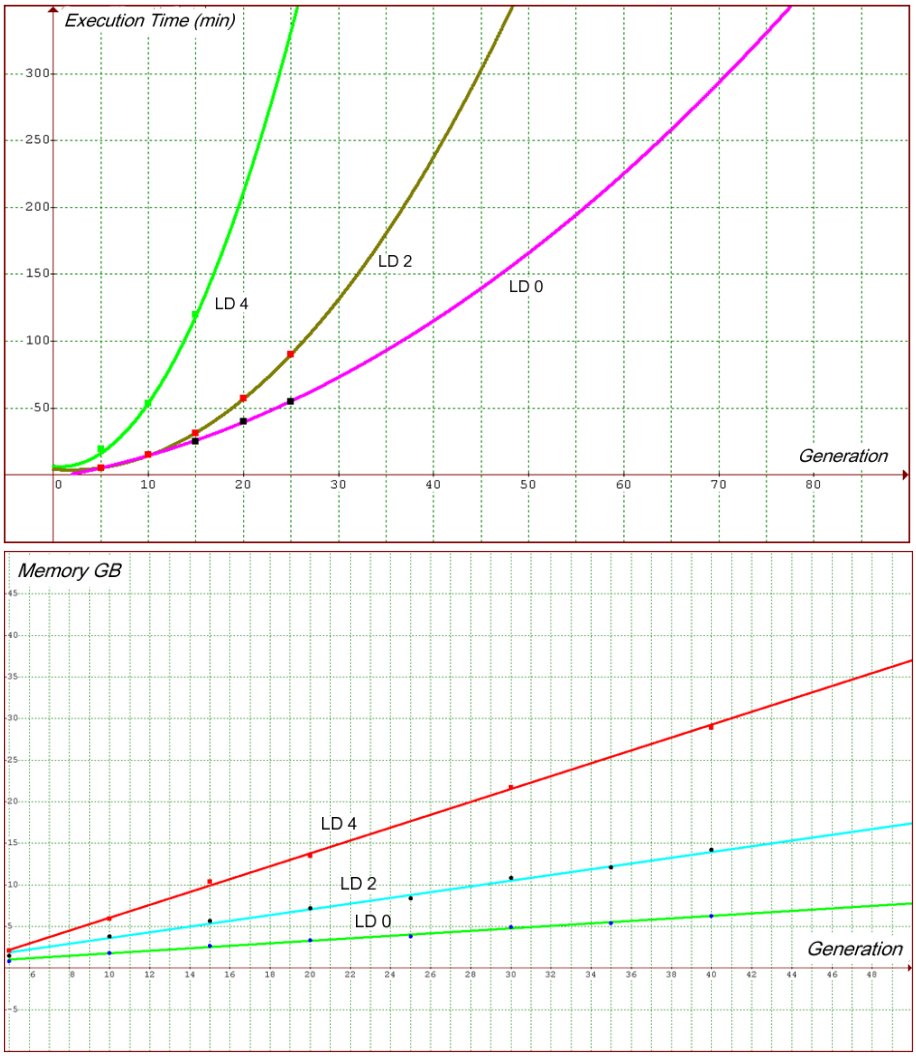

Fig. 8 Run machine requirements: computation time (in minutes) and occupied memory (GB) for multi-seed approach.

a Genetic Algorithm (GA), using different evaluation schemes: mono-seed and multi-seed approaches, and two different fitness functions. Both approaches have been implemented inside a simulator named Mario AI, based in the so-called Infinite Mario Bros, which was implemented for the International Mario AI Competition. The proposed agents were focused, in principle to participate in the Gameplay Track of that competition.

Several experiments have been conducted to test the algorithms and a deep analysis has been performed in each case, in order to set the best configuration

parameters for the GA. However some problems have arisen such as the high memory requirements, which have made it difficult to complete the optimisation process in some cases.

Even so, very competent agents have been obtained for the difficulty levels 0 to 4 in both approaches, which are, in turn, enough for the Gameplay competition requirements.

In the comparison between the approaches, mono-seed can yield excellent agents for the stage where they were 'trained' (evolved), having a quite bad behaviour in a different stage. Multi-seed takes much more computation time and has higher memory requirements, but the agents it yields are very good playing in any stage of the difficulty level considered during the evolution. All these agents play much better than an expert human player and can complete the stages in a time impossible to get for the human.

Since this is a novel research scope for the authors, several future lines of work are opened, starting with the resolution of the commented memory problem. A redesign of the GA's chromosome structure could be useful in this sense. Moreover, some other algorithmic proposals should be tested, such as a reduction in the number of individuals in the population, along with a change in the selection, crossover and mutation mechanisms to control the required balance between exploration and exploitation in GAs.

Once the machine resources problem is solved, the next step could be the comparison between the obtained solutions with any of the existent methods, both online methods like A* approaches, and offline, such as rule-based systems.

Another line of research could be the consideration of a high-level expert knowledge in the FSM. For instance each state should be based in principles proposed by [3]. That means the agents should be not designed according a fixed set of states, but the specific necessities that it should solve in every situation. This solution could reduce the search space and increase the possibility to solve harder stages.

Acknowledgements This work has been supported in part by the P08-TIC-03903 project awarded by the Andalusian Regional Government, the FPU Grant 2009-2942 and the TIN2011-28627-C04-02 project, awarded by the Spanish Ministry of Science and Innovation.

References

1. Bäck, T., *Evolutionary algorithms in theory and practice*, Oxford University Press, 1996.
2. R. Baumgarten, *Mario AI A* agent*, http://www.doc.ic.ac.uk/ rb1006/projects:marioai.
3. Bojarski, S., Bates-Congdon, C., *REALM: A Rule-Based Evolutionary Computation Agent that Learns to Play Mario*. In Proc. of the 2011 IEEE Symposium on Computational Intelligence and Games (CIG 2011), IEEE Press, pp. 83–90, 2011.
4. Booth, T.L., *Sequential Machines and Automata Theory*, John Wiley and Sons, Inc., New York, 1st edition, 1967.
5. Esparcia-Alcázar A.I., Martínez-García A.I., Mora A.M., Merelo J.J., García-Sánchez, P., *Controlling bots in a first person shooter game using genetic algorithms*. In Proc. 2010 IEEE Congress on Evolutionary Computation (CEC 2010), pp. 1–8, 2010.

6. Esparcia-Alcázar A.I., Martínez-García A.I., Mora A.M., Merelo J.J., García-Sánchez, P., *Genetic evolution of fuzzy finite state machines to control bots in a first-person shooter game*. In Proc. GECCO 2010, pp. 829–830, 2010.
7. Fernández-Ares, A., Mora, A.M., Merelo, J.J., García-Sánchez, P., Fernandes, C., *Optimizing player behavior in a real-time strategy game using evolutionary algorithms*. In Proc. 2011 IEEE Congress on Evolutionary Computation, 2011. CEC '11, pp. 2017–2024, 2011.
8. Fernández, A.J., González-Jiménez, J.: *Action Games: Evolutive Experiences*. In Computational Intelligence, Theory and Applications Advances in Soft Computing, Volume 33, pp. 487–501, 2005
9. Fu, D., Houlette, R., *The Ultimate Guide to FSMs in Games*. AI Game Programming Wisdom 2, Charles River Media, pp. 283–302, 2004.
10. Goldberg D.E., Korb B., Deb K., *Messy genetic algorithms: motivation, analysis, and first results*, Complex Systems, 3(5), pp. 493–530, 1989.
11. Hagelbäck, J., *Potential-Field Based navigation in StarCraft*. In Proc. of the 2012 IEEE Symposium on Computational Intelligence and Games (CIG 2012), IEEE Press, pp. 388–393, 2012.
12. Jang, S.H., Yoon, J.W., Cho, S.B., *Optimal strategy selection of non-player character on real time strategy game using a speciated evolutionary algorithm*. In Proc. of the 5th IEEE Symposium on Computational Intelligence and Games (CIG'09), IEEE Press, pp. 75–79, 2009.
13. Laird, J.E, *Using a computer game to develop advanced AI*. Computer, 34(7), pp. 70–75, 2001.
14. Martín, E., Martínez, M., Recio, G., Saez, Y., *Pac-mAnt: Optimization based on ant colonies applied to developing an agent for Ms. Pac-Man*. In Proc. 2010 IEEE Conference on Computational Intelligence and Games (CIG 2010), IEEE Press, pp. 458–464, 2010.
15. Mora, A.M., Montoya, R., Merelo, J.J., García-Sánchez, P., Castillo, P.A., Laredo, J.L.J., Martínez, A., Esparcia, A.I., *Evolving bot AI in Unreal*. In C.D.C et al., ed.: Applications of Evolutionary Computing, Part I. Volume 6024 of LNCS., Springer, pp. 170–179, 2010.
16. Mora, A.M., Moreno, M.A., Merelo, J.J., Castillo, P.A., García-Arenas, M.I., Laredo, J.L.J., *Evolving the cooperative behaviour in Unreal^{TM} bots*. In Proc. 2010 IEEE Conference on Computational Intelligence and Games (CIG 2010), IEEE Press, pp. 241–248, 2010.
17. Mora, A.M., Fernández-Ares, A., Merelo, J.J., García-Sánchez, P., *Dealing with noisy fitness in the design of a RTS game bot*. In Proc. Applications of Evolutionary Computing: EvoApplications 2012, LNCS, vol. 7248, Springer, pp. 234–244, 2012.
18. Nareyek, A., *AI in Computer Games*. Queue Journal, 1(10), ACM, pp. 58–65, 2004.
19. Pedersen, C., Togelius, J., Yannakakis, G., *Modeling Player Experience in Super Mario Bros*. In Proc. 2009 IEEE Symposium on Computational Intelligence and Games (CIG'09), IEEE Press, pp 132–139, 2009.
20. Pepels, T., Windans, H.M., *Enhancements for Monte-Carlo Tree Search in Ms Pac-Man*. In Proc. 2012 IEEE Conference on Computational Intelligence and Games (CIG 2012), IEEE Press, pp. 265–272, 2012.
21. Ponsen, M., Munoz-Avila, H., Spronck, P., Aha, D.W., *Automatically generating game tactics through evolutionary learning*. AI Magazine, 27(3), pp. 75–84, 2006.
22. Shaker, N., Nicolau M., Yannakakis, G.N., Togelius, J., O'neill, M., *Evolving Levels for Super Mario Bros Using Grammatical Evolution*. In Proc. 2012 IEEE Symposium on Computational Intelligence and Games (CIG 2012), IEEE Press, pp 304–311, 2012.
23. Small, R., Bates-Congdon, C. *Agent Smith: Towards an evolutionary rule-based agent for interactive dynamic games*. In Proc. 2009 IEEE Congress on Evolutionary Computation (CEC'09), pp. 660–666, 2009.
24. Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E., *Improving opponent intelligence through offline evolutionary learning*. International Journal of Intelligent Games and Simulation, 2(1), pp. 20–27, 2003.
25. Synnaeve, G., Bessière, P., *A Bayesian Model for RTS Units Control applied to StarCraft*. In Proc. 2011 IEEE Symposium on Computational Intelligence and Games (CIG 2011), IEEE Press, pp 190–196, 2011.

26. Togelius, J., Karakovskiy, S., Koutnik, J., Schmidhuber, J., *Super Mario evolution*. In Proc. 2009 IEEE Symposium on Computational Intelligence and Games (CIG'09), IEEE Press, pp 156–161, 2009.