Evolutionary Behavior Tree Approaches for Navigating Platform Games

Miguel Nicolau, Diego Perez-Liebana, Michael O'Neill and Anthony Brabazon

***Abstract*—Computer games are highly dynamic environments, where players are faced with a multitude of potentially unseen scenarios. In this article, AI controllers are applied to the Mario AI Benchmark platform, by using the Grammatical Evolution system to evolve Behavior Tree structures. These controllers are either evolved to both deal with navigation and reactiveness to elements of the game, or used in conjunction with a dynamic A* approach. The results obtained highlight the applicability of Behavior Trees as representations for evolutionary computation, and their flexibility for incorporation of diverse algorithms to deal with specific aspects of bot control in game environments.**

***Index Terms*—Platform Games, Videogames, Benchmarking, Grammatical Evolution, Behavior Trees, Autonomous Agents.**

I. Introduction

Creating AI controllers for real-time platform games presents a challenge at multiple levels. An agent must decide the next move to make in a limited time budget, usually a few milliseconds. Within this time limitation, the algorithm that controls the agent must be able to both react to imminent hazards, and also devise an action plan that allows it to accomplish the goals that lead to winning the game.

The primary objective of this paper is to investigate the use of Grammatical Evolution (GE) [1] to evolve Behavior Trees (BT) [2], to deal with both aspects of real-time agent control. In the environment employed, the Mario AI Benchmark [3], the final goal is to reach the end of levels, avoiding enemies or other hazards that may kill the player. Thus, the avatar (Mario) must both react to events that happen in its proximity, and devise a path through the level to make progress.

These two components are strongly related: both make use of the same set of movement actions (plus, in some cases, the shooting action). In this study, it is shown how these can be separated (while keeping their interdependency), by using a combination of a BT representation and a grammar-based evolutionary approach. Two different approaches are compared, one using the same movement actions to face both issues, and another providing specific re-usable behaviors to deal with reactiveness and navigation separately. The results obtained highlight the advantages of the latter approach.

The work described in this paper extends previous studies on the evolution of BTs for Mario AI [4], [5]; it provides a unified set of experiments that allows a direct comparison of those two approaches, and provides their in-depth analysis and comparison in terms of evolvability, generalization, and complexity of resulting controllers, leading to conclusions and recommendations in terms of their applicability, both to Mario AI and other games.

Miguel Nicolau, Michael O'Neill and Anthony Brabazon are with the Natural Computing Research and Applications Group, University College Dublin, Ireland, email `Miguel.Nicolau@ucd.ie`, `M.ONeill@ucd.ie`, `Anthony.Brabazon@ucd.ie`; Diego Perez-Liebana is with the School of Computer Science and Electronic Engineering, University of Essex, Colchester, UK, email: `dperez@essex.ac.uk`

This document first analyzes the relevant literature, in Section II. The various components of this study are then introduced: the Mario AI Benchmark in Section III, the controller approaches in Section IV, BT structures in Section V, and GE and its application in Section VI. Finally, the experimental design and results are presented in Sections VII and VIII. Section IX draws conclusions and future work directions.

II. Relevant Literature

Platform games are one of the most successful game genres of all times [6], and the Mario AI environment [7], [3] has provided an excellent platform for AI research in this genre, over the past few years. In terms of the creation of AI controllers that aim to maximize their final game score, relevant literature includes: the use of rule-based agents with higher-level, hand-designed conditions and actions [8]; the use of cuckoo search and its comparison with a standard genetic algorithm approach, approaching the Mario AI game as an instance of the Traveling Salesman Problem [9]; the use of Q-Learning with full game information, and also with minimized enemy information for reduced search space [10]; the use of Neural-Networks with Manifold Learning as a dimensionality-reducing technique [11]; the evolution of finite-state machines created with genetic algorithms [12], [13]; and the combination of Monte Carlo Tree Search with appropriate heuristics [14].

The literature is also broad in terms of using path planning for navigation, both in robotics [15] and in games, such as *Unreal Tournament*, *Quake III* or *Half Life* [16]. One of the most commonly used algorithms for path finding is A*, because of its great performance, accuracy and efficiency [17]. In fact, the literature shows its recurring usage in the Mario AI environment, such as the use of A* to manage local navigation on the top three submissions to the 2009 Mario AI Championship [18]; its use to determine keystrokes for high level actions [8]; its use for navigation in combination with Q-learning [19]; and to characterize player behavior and their deviations from rational actions [20].

Game environments are dynamic environments, and typically provide noisy fitness. Thus is particularly the case with Mario AI: maps generated with the same difficulty level still have a huge range of difficulties, due to the generation of maps which are physically complex (or even impossible) to navigate.

There is a large body of research in the area of noisy fitness environments: Jin and Branke [21] and Qian et al. [22] provide extensive reviews in this area. Authors such as Di Pietro et al. [23], Mora et al. [24], and Merelo et al. [25], amongst others, have looked at this issue specifically from the point of view of computer games as noisy fitness environments.

Grammatical Evolution has been previously applied to gaming environments. Galván-López et al. [26] evolved Ms. Pac-Man controllers, specifying high-level functions to analyze the game environment and decide on the best course of action; Harper [27] co-evolved controllers for Robocode Tanks.

BTs were introduced as a means to encode formal system specifications [2], but have gained popularity as a way to encode game AI [28]. They were used in high-revenue commercial games, such as "Halo" [29] and "Spore" [30], smaller indie games, such as "Façade" [31], and other unpublished uses [28], illustrating their importance in the game AI world.

The work of Lim et al. [32] specifically dealt with evolving BTs. It used Genetic Programming (GP) [33] to successfully evolve AI controllers for the *DEFCON* game. Some hurdles were encountered in this work, such as how to deal with the exchange of typed tree structures between individuals; these, amongst others, are addressed in the current study.

III. The Mario AI Benchmark

Super Mario Bros is a 2D platform game where the player controls an avatar that must reach the far right end of the level by avoiding enemies, hazards, and collecting bonus items. Markus Persson implemented an open source version of this game (*Infinite Mario Bros*), which was later adapted by Togelius et al. [7], [3] as a benchmark for game AI, and a framework for the different Mario AI Competitions [18], [34].

This benchmark was employed for the experiments described in this paper (concretely the Gameplay track functionality of those competitions). It tests agents in multiple levels, customizable by difficulty, type (over or underground), length, time limit, creatures (presence or absence), dead ends, and random seed for the automatic generation of the level.

A. Environment Information

In order to analyze the environment surrounding Mario, the agent controller may access two matrices, one of which provides information regarding the geometry of the level, and the other indicates the presence of enemies (see Fig. 1).

The benchmark also provides information about the state of the agent: its location in the level, its mode (*Small*, *Big* or *Fire*), and boolean flags indicating extra information, such as if Mario is on the ground, if able to shoot, jump, or if carrying a turtle shell. Additional information is also available, including the game status (running, won or lost), the time left before the game is over, and statistics about enemies killed.

B. Mario Effectors

The agent can perform different actions at each game step. These actions include three directions (*Left*, *Right* and *Down* - *Up* has no meaning in this implementation), *Jump*, and

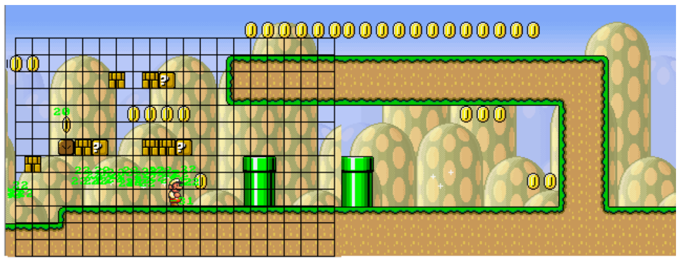


Fig. 1: Mario and environmental information. Both matrices are of size 21×21, centered in Mario.

Run/Fire. If the agent is moving right or left and the action *Run/Fire* is applied, Mario moves faster. Also, when in *Fire* mode, it makes the agent shoot a fireball. If *Jump* and *Run/Fire* are applied simultaneously, Mario jumps farther.

These actions are provided by the controller as a boolean array, allowing an action space of $2^5 = 32$ actions (although some of these are nonsensical, such as left and right pushed at the same time). This array must be returned in the method *getAction()*, which serves as the interface between the game and the agent. As the game is played in real-time, the agent needs to specify an action every $40ms$ or it will be disqualified.

C. Benchmark Version

Several benchmark versions have been used in the different Mario AI competitions, with substantial differences between them. One of the most important distinctions is the presence of *dead ends*, a feature of the level that presents more than one path to move ahead, although at least one of them is a *cul de sac*, forcing the player to go back and take another route.

After the success of A* approaches at the 2009 competition, dead end traps were introduced for the 2010 contest. This was a clear hazard for an A* algorithm with a foresight no longer than the size of the matrices that the environment provides.

This paper uses the 2010 version (0.1.5). The experiments present a way to overcome dead ends, by using a dynamic A* algorithm (Section IV-B). Karakovskiy and Togelius [3] fully describe the dynamics of the benchmark.

IV. Controller Approaches

The focus of this work is on the evolution of BTs as controllers for Mario AI. GE was able to combine the two required aspects of the agent behavior for this game: reactiveness (dealing with close enemies and hazards) and navigation (determining paths to move across the static elements in the level). Both were dealt with using basic game movements (*left*, *right*, *down*, *fire*) or combinations of these (see Table IV).

To analyze the importance of the navigation component of the algorithm's behavior, two different approaches were studied in this research, each using a different set of routines. In the first one, *ReactiveMario (NoAstar)*, a combination of reactiveness and (very basic) navigation routines are employed to evolve the BTs. The second approach, *PlanningMario (Astar)*, uses a specific navigation algorithm (A*), allowing GE to focus primarily on the reactiveness behavior. The overall structure of the evolved BTs, as well as the integration of reactiveness and navigation, are detailed in Section VI-A.

A. ReactiveMario (NoAstar)

This first approach employs no explicit path-finding, with the agent only focused on reacting appropriately to moving elements in the game. Among these elements, Mario considers the position of *goombas*, bullets, flying turtles, bonus mushrooms and fire flowers. For this agent, GE evolves BTs that react to these entities and navigates the agent through the levels. This controller was submitted to the 2010 Mario AI Competition (Gameplay track), ranking 4^{th}out of 8 entries [3].

One of the most difficult navigational hazards are dead ends; an example is shown in Fig. 1. Two sub-trees (*UseRightGap* and *AvoidRightTrap*, see Table IV) were manually designed to solve this specific problem. The latter sub-tree detects a dead end in front of Mario and moves him back until he has left the trap (i.e. there is no obstacle over his head). Then, the former routine is used to find a platform which Mario can jump onto, to overcome the trap by running through the upper part.

B. PlanningMario (Astar)

In order to employ a path finding algorithm such as A*, the level must be represented as a navigable graph, a structure not supplied by the benchmark. Also, the resulting graph needs to be able modifiable, either because changes in the blocks (which can be destroyed by Mario) or changes in the state of the agent itself (from *Big* to *Small*, or vice-versa) can modify the validity of old paths. Thus, it is the responsibility of the agent to generate this graph *dynamically* at each step.

This section briefly summarizes the graph creation process. For the complete description of this procedure, the reader is referred to the previous work of the authors [5].

1) Level structure and nodes: The initial step of the algorithm consists of analysing the environment matrices (described in Section III-A). As Mario advances through the level, the positions of fixed blocks are stored to build the level map. As Mario is able to stand in all these blocks, a node for the graph is added for each one of them. This allows the inclusion of additional meta-data information, such as the type of block (question or brick), enemies, and/or collectible items, which can be later used for queries in the BT.

2) Graph edges: Constructing the graph for this game comes with several challenges. First, the navigation of the level can depend on the state of Mario (*Small* or *Big*). Second, it must deal with the asymmetry on the edges. As the game is played sideways, horizontal and vertical edges must be traversed in different ways (in contrast with a top view, zenithal perspective). The following types of links are created for the graph, which are also shown in the example in Fig. 2.

- **Walk links**: Bidirectional edges that join two horizontally adjacent nodes.
- **Jump links**: Unidirectional upward edges that join nodes vertically separated by no more than 3 cells and horizontally by 1 position.
- **Vertical jump links**: Unidirectional upward edges joining nodes vertically separated by no more than 3 cells.
- **Fall links**: Unidirectional downward edges that join nodes vertically separated by any number of cells and horizontally by 1 position.

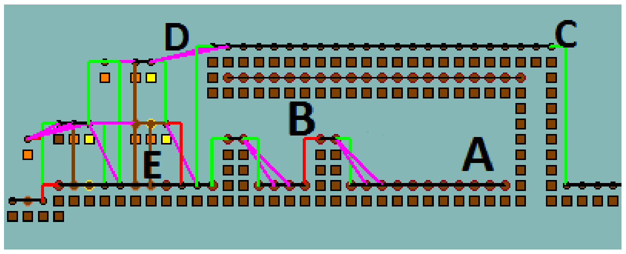


Fig. 2: Navigation graph representation. Different types of edges: A: Walk link. B: Jump link. C: Fall link. D: Faith jump link. E: Break jump link.

- **Faith jump links**: Bidirectional edges that link two nodes horizontally separated by no more than 4 cells.
- **Break jump links**: Special case of *jump* link with a brick block in the trajectory of the jump. Because this block can (potentially) be destroyed, this link is included in the graph as it can become a regular *jump* link.

The cost of each edge is calculated as the product of a basic cost (Manhattan distance, i.e. the sum of absolute differences in cartesian coordinates) and a factor determined by the link type. As traversing some edges involves more risk (i.e. jumping is more prone to fail than walking), higher factors are given to more complex links. A factor of 1.5 is assigned to any link associated with a jump, and 3.0 to *break jump* links due to the extra cost involved in trying to break the brick.

Once A* can be used to generate paths to different positions in the level, one can design actions and routines for GE to use during the evolution of BTs. The next section gives a definition of BTs, and how are they used for this game.

V. Behavior Trees

A BT is a structure that allows the organization of behaviors in a hierarchical manner, decomposing an initially broad task in several sub-trees of reduced complexity. For instance, the behavior of a game NPC could be decomposed in different sub-behaviors such as patrolling or attacking, all the way to low level actions to play sounds or animations.

Fig.3 shows an example BT, such as used in this study. Nodes can return a *success* or *failure* value to their parent node, and are divided into two major categories: *control nodes* and *leaf nodes*. The first control the flow through the tree, using values from children nodes to choose the next node to execute. They include **Sequence** nodes, which execute children nodes from left to right until one returns *failure*; and **Selector** nodes, which execute children nodes until one succeeds. Finally, **Filter** nodes can modify the execution flow in different ways (like loops, running a node until failure, etc).

Leaf nodes are **Conditions** and **Actions**. Conditions query situations and features of the current game state, while actions apply moves in the game. Actions usually return *success*, as executing an action is normally always possible, while the returned value of conditions depends on the query performed.

A. Behaviour Trees for Mario

For this study, the leaf nodes used in BTs are:

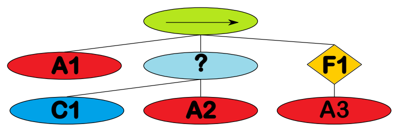


Fig. 3: Example BT with sequence (→), selector (?), condition (C), action (A) and filter (F) nodes.

- **Conditions**. Provide information about enemies (distance to Mario and their type) and obstacles (type and position of the blocks) within the observation range of the agent.
- **Actions**. The most useful action combinations are provided to the BT, based on those described previously (see Section III-B). Examples are *Down*, *Fire*, *RunRight* (*Right* and *Run* both pressed), *NOP* (no buttons pressed) or *WalkLeft*. There are also actions to request paths to specific locations, when using A*.
- **Sub-trees**. These indivisible units perform actions that require a specific sequence of moves to be applied. Examples are jumps, which require a first cycle without pressing the jump button, and consecutive repetitions of the move in order to make longer jumps. Sub-trees are achieved by combining different filter and action nodes. Fig. 4 shows the sub-tree to make long jumps to the right. The `NOP` action ensures no button is pressed, and then a `Loop` filter executes `JumpRight` a given number of times. As the BT execution pauses until the next game cycle when an action is reached, this sub-tree will press the jump button for a certain number of game cycles.

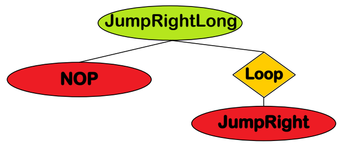


Fig. 4: Sub-tree for executing long jumps to the right.

Table IV includes all conditions, actions, filters and sub-trees designed for the agent, and available to the evolutionary algorithm. Note that some sub-trees are only available for controllers without A*. These are used for navigational purposes, which are taken care of by A* routines in the other controllers.

B. Behavior Tree XML Structure

The BTs used are stored in XML files. This implies that the agent must be able to read these files and, more importantly, there is a concrete file structure that must be generated by GE.

The structure of the XML file is hierarchical, defining the type of each node and the operation that it represents. Listing 1 shows the (simplified) XML code of the sub-tree in Fig. 4.

```
<?xml version="2.0" encoding="UTF−8"?>
<Node Type="Sequence">
        <Node Type="Action" Operation="NOP"/>
        <Node Type="Filter" Filter_Type="Loop" Times="9">
                <Node Type="Action" Operation="JumpRight"/>
        </Node>
</Node>
```

Listing 1: JumpRightLong sub-tree XML

VI. Grammatical Evolution

The syntax of the BT controllers can be quite complex, as seen in Listing 1. Also, the variety of control and leaf nodes requires a system that can ensure the syntactic correctness of the evolved controllers, and also incorporate some domain knowledge if possible (explained below). These are requirements easily achievable with the Grammatical Evolution [1] (GE) system, hence its choice to evolve the BT controllers.

While similar to GP [33], GE evolves linear numerical strings, and maps these to syntactically-correct solutions, through the use of a context-free grammar.

GE has comparable performance with GP for symbolic regression problems [1], but its grammar provides extra control of the syntax of solutions, both in terms of biases [35], [36] and data-structures used. This allows GE to be applied to a variety of problem domains, such as Financial Modeling [37], animation optimization [38], or game controllers [27].

A. Generating Behaviour Trees with GE

1) BT Structure: The BT (XML) syntax was specified in the grammar, with all conditions, actions, sub-trees and filters. In earlier experimentation, GE was free to combine these, but this approach proved to be too flexible: with no structural guidelines, most trees were badly structured (such as sequences of sequences, with `NOP` actions at their leaves), non human-readable, and computationally demanding to execute.

To avoid these issues, the syntax of BTs was limited through the grammar. While still of variable size, BTs are contrived to follow an *and-or* tree structure [39], much like a binary decision diagram [40], which is a recommended [41] way of building BTs for game AI. The following structure was used:

- The root node is a selector, with a variable number of *Behavior Block* (BB) sub-trees, encoding sub-behaviors;
- Each BB consists of a sequence of one or more conditions, followed by a sequence of actions or sub-trees;
- A last (unconditioned) BB, which is either a sequence of actions and sub-trees, or a default navigation behavior (when using A*).

Fig. 5 exemplifies the syntax described. At the beginning of the BT execution, the root selector chooses the leftmost BB. If its associated conditions fail, the execution follows in a left-to-right priority order. As the conditions provided are complex representations of the game state, the grammar limits the number of associated conditions of each BB to one or two, leaving the number of actions and sub-trees unlimited.

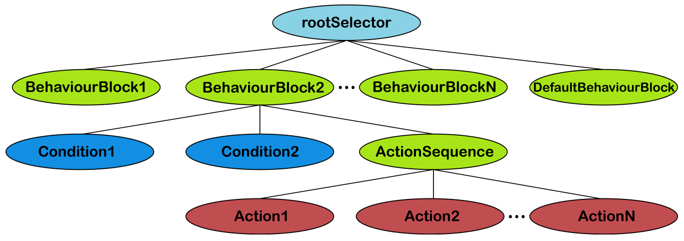


Fig. 5: Structure of evolved BTs.

The default BB is the right-most block. Thus, it is the one with the lowest priority, only being executed if all the previous blocks were not. Its contents depend on the navigation used: without A*, it is simply composed of a sequence of actions, without associated conditions.

If A* is used, however, it is composed of the behavior `DefaultGoRight`, a selector with two sub-trees, shown in Fig. 6. The first (`Default Path Planner`) establishes if a new path is required, and calculates that path (with a default behavior of jumping to the right, if it was not possible to calculate a new path). The second (`Path Follower`) checks if a path was set, and executes actions to follow that path.

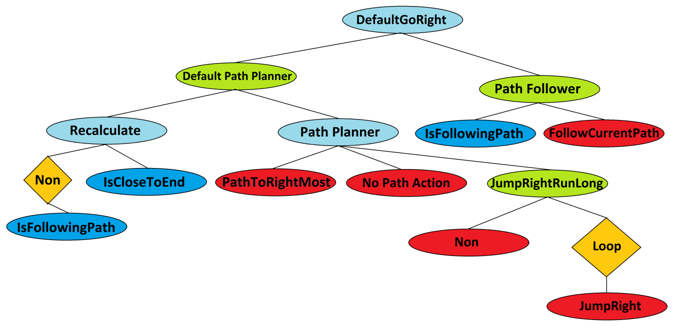


Fig. 6: Sub-tree for path planning. It calculates, if needed, the path to the rightmost position available.

2) Grammar Design: With the syntax described above, each BB becomes a self-contained structure, and it makes sense to allow individuals to exchange these between them. To this end, specific crossover points were encoded in the grammar, defining blocks for exchange. This technique [42] uses a special grammar symbol (`<GEXOMarker>`) to label crossover points; the search algorithm then only slices an individual according to these points. It has been used in previous iterations of this work [4], [5], as well as in the evolution of 3D projections [43].

We used a two-point crossover, creating an operator similar to sub-tree crossover in GP, but allowing the exchange of a variable number of blocks between individuals. Without these markers, the 1-point crossover used in standard GE would provide more exploration but less exploitation; given the cost of the fitness function, this trade-off was necessary.

Finally, an individual is also allowed to crossover with himself, thus creating a sub-tree swap operation; this works as a means to modify the priority of a BB: the further to the left within the root selector, the bigger the likelihood of execution of a block. Fig. 7 contains part of the grammar used, specifying the XML syntax, along with the crossover markers.

VII. Experiments

A series of experiments were run, to ascertain the evolvability of BTs as controllers for Mario AI. BTs were evolved using GE, and their training and test performance were monitored over time, along with other statistical measurements.

These experiments also tested the separation of reactiveness and navigation routines, and whether this leads to improved results. Finally, a series of different approaches were tested, to deal with the highly dynamic environment created by the Mario AI Benchmark. These are discussed in Section VII-B.

A. Fitness Evaluation

To test each evolved controller, a set of Mario AI levels is generated. Each *mapset* is composed of 10 levels (5 difficulty settings, with two types of map each), and is generated with a single random seed. The resulting fitness value, to be maximized, is a weighted sum of distance traveled and other factors, such as enemy kills and collected items (this is the actual Mario AI Benchmark score). As both Mario AI maps and BT controllers are deterministic, applying the same controller to the same map always wields the same fitness.

B. Adaptable Controllers

A difficulty with such a dynamic problem is that of generalization performance. In the Mario AI competition, controllers are tested in a set of random mapsets, where each map ranges from very easy to physically impossible to terminate (for the same difficulty level). To improve the generalization of the evolved controllers, the following approaches were tested (note that both non-A* and A* versions of these were used):

- **Single**: always use a single "typical" mapset for evaluation (same seed for all independent runs), assuming that the agent is faced with enough enemy and obstacle diversity to evolve good reactiveness routines.
- **Five**: test each controller in five mapsets, which never change during the evolutionary cycle (and are the same for all runs); this increases the variety of situations each controller is evaluated on. This means each controller is evaluated in five more maps than in the *Single* approach.
- **Change1**: use only one mapset for evaluation, but change it at every generation (same sequence of mapsets for each run), to increase the variety if situations each controller trains on, while keeping the evaluation effort small. To ensure continuity between generations, the parent population is reevaluated with the new generation's mapset.
- **Change5**: use five mapsets for each evaluation, but change all five at each generation (same sequence for all runs), ditching the previous mapsets. The parent population is reevaluated with the new mapsets at the start of each new generation.
- **Slide**: use five mapsets for each evaluation, replacing one mapset with a new one at every generation, in a `12345`, `23456`, etc. sliding window manner (same sequence for all runs). The parent population is reevaluated with the new five mapsets at the start of each new generation.

C. Experimental Setup

Each of the 10 systems (five approaches, with and without A*) used the setup shown in Table I. As different approaches use a different number of mapsets for evaluation, and a single mapset took anywhere between 0.7s and 6.0s to evaluate (using a single core of a 2.8 GHz Intel Core i7 processor), different numbers of generations were used, so that each approach used the same number of mapsets per run.

```
<BT> ::= '<?xml version="1.0" encoding="utf-8"?>\n' <XMLPart>
<XMLPart> ::= '<Behavior>\n' <RootNode> '</Behavior>\n'

<RootNode> ::= '<Node Name="GE_BT3" Type="Root">\n'<RootSelectorNode>'</Node>\n'

<RootSelectorNode> ::= '<Node Type="Selector" Name="rootSelector" >\n' <2orMoreSeqAndDefaultBehaviour> '</Node>\n'
<2orMoreSeqAndDefaultBehaviour> ::= <ConnectorHeader> <SequenceNodes> <GEXOMarker> <FinalSequence> '</Connector>\n'
<ConnectorHeader> ::= '<Connector Identifier="GenericChildren">\n'

<SequenceNodes> ::= <SequenceNode> | <SequenceNodes> <SequenceNode>
<SequenceNode> ::= <GEXOMarker> '<Node Type="Sequence" Name="BehaviourBlock">\n' <ConnectorHeader>
                   <1to2Conditions> <FilteredSeqOfActionsAndLUTs> '</Connector>\n</Node>\n'

<1to2Conditions> ::= <ConditionNode> | <ConditionNode> <ConditionNode> | <ConditionedLUT>
<ConditionNode> ::= '<Node Name="'<ConditionOp>'" Type="Condition" />\n'

<FilteredSeqOfActionsAndLUTs> ::= <FilterHeader> <ConnectorHeader> <SeqOfActionsAndLUTs>
                   '</Connector>\n</Node>\n' | <SeqOfActionsAndLUTs>

<SeqOfActionsAndLUTs> ::= '<Node Type="Sequence" Name="mySequence" >\n'
                   <ConnectorHeader> <1orMoreActionsOrLUTs> '</Connector>\n</Node>\n'
<FinalSequence> ::= '<Node Type="Sequence" Name="defaultSequence">\n'
                   <ConnectorHeader> <DefaultBehaviourBlock> '</Connector>\n </Node>\n'

<1orMoreActionsOrLUTs> ::= <ActionOrLUT> | <1orMoreActionsOrLUTs> <ActionOrLUT>
<ActionOrLUT> ::= <ActionNode> | <LUTNode>
```

Fig. 7: Extract of the grammar used, showing the incorporation of the XML syntax.

TABLE I: Experimental Setup

GE	Population Size	500
	Evaluations	250000
	Derivation-tree Depth Range (for initialization)	20…30
	Derivation-tree Max Depth	*unset*
	Tail Ratio (for initialization)	50%
	Selection Tournament Size	1%
	Elitism (for generational replacement)	10%
	Marked 2-point Crossover Ratio	50%
	Marked Swap Crossover Ratio	50%
	Average Mutation Events per Individual	1
Mario	Level Difficulties	0…4
	Level Types	0 1
	Level Length	320

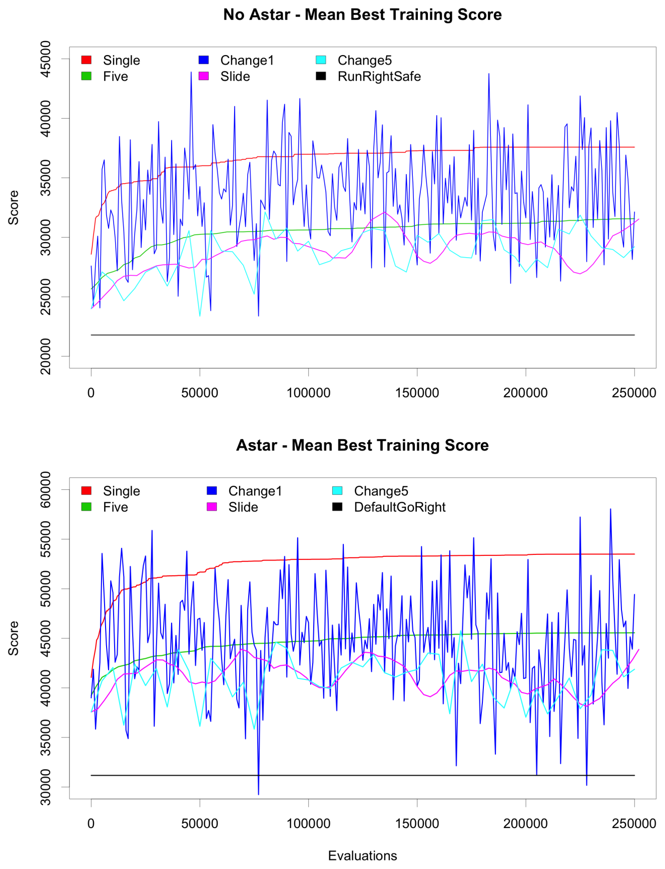


Fig. 8: Mean best training score across time, for all approaches not using (top) or using (bottom) A* (results averaged over 30 independent runs).

VIII. Results and Analysis

A. Training Performance

Fig. 8 plots the mean best controller training score, without (top) or with (bottom) A*, for all approaches. It also shows the average performance of the respective reference behaviors: *RunRightSafe* without A*, and *DefaultGoRight* for A* (these were calculated using a generalization test, described below).

The first evident observation is that all approaches do substantially better than their respective reference behaviors. This shows that the BT approach is effective in adding reactive elements to the controllers, which enhance their performance.

The second observation is the relative performance difference between controllers without or with A* navigation. The *RunRightSafe* controller has an average performance of just below 22000 points, while *DefaultGoRight* averages above 31000. This is very close or superior to the average controller performance for most approaches not using A* (apart from the *Single* approach), and highlights the performance boost of using a dedicated, non-deterministic algorithm for navigation.

As for each of the approaches, their relative performance is similar with or without A*. The *Single* approach has the best training performance; it is quite successful at optimizing the controller behavior for the single mapset it is trained on, regardless of the initial random seed. It achieves the best training score with or without A*, and also exhibits the best evolvability, with a typical optimization performance curve.

The *Five* approach exhibits a similar behavior, with a steady improvement in average performance across the five mapsets

TABLE II: Least-Squares Analysis of Learning Rates

		Approach	Intercept	Slope	Std. E.	Res. E.
Train	No A*	Single	3.49E+4	1.40E-2	4.84E-4	784.5
		Five	2.87E+4	1.36E-2	1.00E-3	739.3
		Change1	3.24E+4	8.64E-3	3.43E-3	3946
		Slide	2.75E+4	1.12E-2	2.03E-3	1383
		Change5	2.70E+4	1.25E-2	3.35E-3	1763
		RunRightSafe	21790	0.0	0.0	0.0
	A*	Single	5.03E+4	1.69E-2	7.63E-4	1236
		Five	4.28E+4	1.41E-2	9.29E-4	680.9
		Change1	4.55E+4	-5.81E-3	4.47E-3	5133
		Slide	4.16E+4	2.94E-3	2.16E-3	1467
		Change5	4.06E+4	4.34E-3	4.50E-3	2369
		DefaultGoRight	3.11E+4	0.0	0.0	0.0
Test	No A*	Single	2.12E+4	2.26E-3	2.50E-4	127.7
		Five	2.24E+4	3.22E-3	6.29E-4	321
		Change1	2.25E+4	1.18E-2	1.25E-3	639.1
		Slide	2.29E+4	1.15E-2	1.34E-3	684.2
		Change5	2.27E+4	1.17E-2	1.30E-3	667.1
		RunRightSafe	21790	0.0	0.0	0.0
	A*	Single	3.50E+4	-3.69E-3	4.53E-4	231.4
		Five	3.63E+4	4.45E-4	3.38E-4	172.6
		Change1	3.69E+4	1.04E-3	9.59E-4	489.6
		Slide	4.06E+4	1.61E-2	1.76E-3	902.7
		Change5	3.80E+4	8.16E-3	7.74E-4	395.2
		DefaultGoRight	3.11E+4	0.0	0.0	0.0

it was trained on, but with a lower total score; this is due to optimizing the performance across five mapsets.

The *Change1* approach is the noisiest in terms of evolution across time, as the mapset used for evaluation changes every generation. This shows the difficulty range of maps generated with different random seeds, even with the same difficulty setting. With or without A*, this approach has both the highest and lowest average score of all approaches, and with A*, sometimes performs worse than the default behavior.

Finally, the *Change5* and *Slide* approaches exhibit similar performance. These approaches also suffer from the extreme range of difficulties of the generated maps, but not to the same degree as *Change1*. The *Slide* approach dampens this effect to a higher degree, due to its moving window of used mapsets.

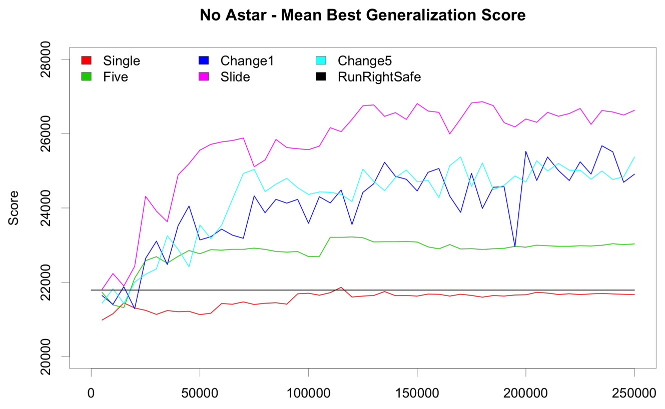
In order to analyze the average learning rate of the different approaches, a linear regression was calculated, with the data from Fig. 8. This is shown in the top half of Table II. Although the learning curves are clearly not linear, a simple linear model allows one to draw some observations: the intercept roughly represents the starting performance of each controller, the slope is an approximation of the learning rate of each approach, and the standard and residual errors are a measure of the noise present in the average learning performance.

The values measured show that the *Single* approach exhibits the best average learning rate across all runs, with *Five* also exhibiting a good learning rate. *Slide* and *Change5* exhibit lower, similar learning rates, with higher noise. Finally, *Change1* exhibits the lowest learning rate, which is actually negative when used in conjunction with A* navigation. It also has the highest residual error, an indication of the range of scores obtained with different maps. This also highlights how hard it is to evolve controllers in such a dynamic environment. Videos of the best controllers of some runs can be checked online[1].

[1] http://tinyurl.com/gebtMarioAI

B. Test Performance

To measure the performance of evolved controllers in unseen scenarios, a generalization test was devised, consisting of 20 unseen mapsets (seeds 666 to 685), with the same parameters as training mapsets. The best individual (according to training performance) was tested every 5000 evaluations; the average results across all runs are shown in Fig. 9.



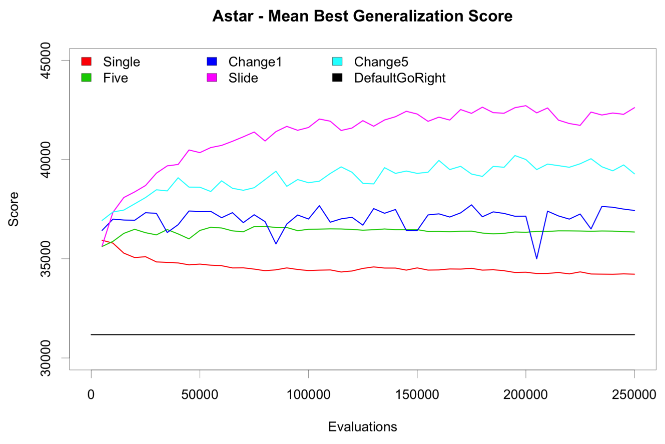


Fig. 9: Mean best test score of the best training individual every 5000 evaluations, for all approaches not using (top) or using (bottom) A* (averaged across 30 runs).

The first observation is the performance range of all approaches. While the training performance without A* ranged from 25000 to 40000 points, in testing it ranges from 21000 to 27000. The same happens with A*, with a training range of 35000-55000, and testing 34000-44000. Given that these are scores using unseen scenarios, this is to be expected.

The relative performance of all approaches is quite different from the training performance. The *Single* approach clearly overfitted its single training mapset, and has the lowest generalization score overall, which without A* is actually worse than the reference *RunRightSafe* behavior, while with A* its average generalization score worsens as evolution progresses.

The *Five* approach does slightly better. It improves its generalization score over time without A*, albeit with a few signs of training overfitting. With A*, it reaches its best test performance early on, and never improves over time. Note that its performance is once again substantially better with A* (over 36000 points) than without (around 23000 points).

Change1 steadily improves its generalization performance,

TABLE III: Average Test Performance and Std. Deviation

	Approach	Avg. Score	Std. Dev.
No A*	Single	21668.1	1531.9
	Five	23033.3	2210.9
	Change1	24910.4	1860.2
	Slide	26629.3	1631.7
	Change5	25374.7	1609.9
	RunRightSafe	21790.2	0.0
A*	Single	34224.1	1016.8
	Five	36350.1	468.6
	Change1	37435.2	596.2
	Slide	42616.7	731.7
	Change5	39282.5	579.9
	DefaultGoRight	31173.8	0.0

when used without A*. This is despite its very noisy average training performance. A reason for this is the large number of generations it is allowed to evolve, given the reduced number of mapsets used per generation. When used with A*, it only slightly improves its generalization performance over time.

Change5 steadily improves its generalization score over time, particularly when used without A*. The same is true about the *Slide* approach, but with a substantially better average generalization score at all evaluation steps.

Table III shows the test performance of the best training controllers (averaged across 30 runs). All A* approaches present significantly better test performance when compared with their No A* counterparts, and the relative performance of the different approaches is observable, with mostly non-overlapping standard deviation intervals.

The bottom half of Table II analyzes the test score rates of all approaches. It shows very low improvement rates for the *Single* and *Five* approaches, with the former having a negative rate, when used with A* navigation. *Slide* and *Change5* exhibit good test performance improvement over time, with the learning rate of *Slide* with A* being one of the highest across all sets (training and testing) and approaches (A* or not). Finally, of interest is also the learning rate of the *Change1* approach, when using A*: albeit very low, it is positive, in contrast to its negative training learning rate.

C. Further Analysis

1) Fitness Breakdown: Fig. 10 plots the test performance of the best evolved controllers, but monitoring specific fitness contributions of their actions: number of cells passed, time left (i.e. time left when Mario dies or finishes a level), and number of kills (all averaged across the 20 test mapsets).

It is again evident the contribution that A* navigation makes to the survivability of Mario (and hence to the overall fitness). The average number of cells passed with the *DefaultGoRight* controller is much higher, leading to a higher number of (random) kills. The time left with A* is also superior, due mainly to it not getting stuck in difficult to navigate areas.

When controllers are evolved without A*, their test performance is actually worse in terms of total cells passed than their reference behavior (*RunRightSafe*), as BT structures need to be evolved to effectively combine navigation and reactiveness actions (such as number of kills). Even the poorly generalizable *Single* and *Five* controllers are able to improve their average number of kills, even though they exhibit little or no evolution in number of cells passed or total time left.

When A* is used, the BT structures are evolved mainly for reactiveness, and this is evident from early on: all approaches produce good reactiveness behavior blocks, which allow the controllers to improve the good navigation base that the *DefaultGoRight* behavior provides, and increase the number of cells passed, as well as the number of kills (with a much higher improvement than controllers without A*).

Across all these measures, it is again clear that the *Single* approach does not generalize well, and exhibits little or no improvement (apart from number of kills, when used without A*). *Five* performs similarly, at a slightly better level; all other approaches evolve better generalizable behaviors across time.

Of interest is also the (non-)evolution of total time left when using A*. Better navigation approaches mean that the maps will be finished in less time. However, these combined with complex reactive behaviors also increase the survivability of the player, and thus the time spent in each map.

Fitness-contributing behaviors such as collecting items were ignored by the controllers (they are hard to evolve): less than one item on average was collected by controllers without A*, and between 1 and 2 items when using A* (not plotted).

2) Genotype Solution Size: Another interesting analysis can be drawn for the average solution size, plotted in Fig. 11. It provides further evidence that the *Single* and *Five* approaches overfit their target maps, with average genotype sizes steadily increasing throughout evolution. This is true to some extent for all controllers, when used without A*. When A* navigation is used, however, the solution size is a lot more stable.

3) Behavior Tree Solution Structure: Although the genotype size (and hence number of nodes in the BT controllers) is comparable with or without A*, the actual structure of these trees is radically different with the two navigation approaches.

Fig. 12 plots the number of Behavior Blocks (BBs) in the best controllers across time. This number is very stable, indicating that evolution chooses the number of BBs early on, and then mostly just optimizes the contents of each BB.

The smaller number of BBs without A* means that each BB is a very complex structure. In fact, this number is so small, it limits the effectiveness of the crossover operator, designed for exchanging BBs. This happens because each BB incorporates a complex mix of both navigation and reactiveness actions, which does not mix quite as well in the context created in BTs evolved by other individuals in the population.

When using A*, each BB is mostly a compact set of conditions and actions evolved for reactiveness, and relying on the default behavior for navigation. As a result, a larger number of BBs is evolved by each controller, which can be exchanged through crossover as independent reactive sequences.

IX. Conclusions

This article presented an extension of previous work [4], [5] on the evolution of Behavior Trees for the Mario AI Benchmark. Two approaches were compared: a typical *black-box* approach, where all possible actions were combined to create one-in-all controllers; and a layered approach, where

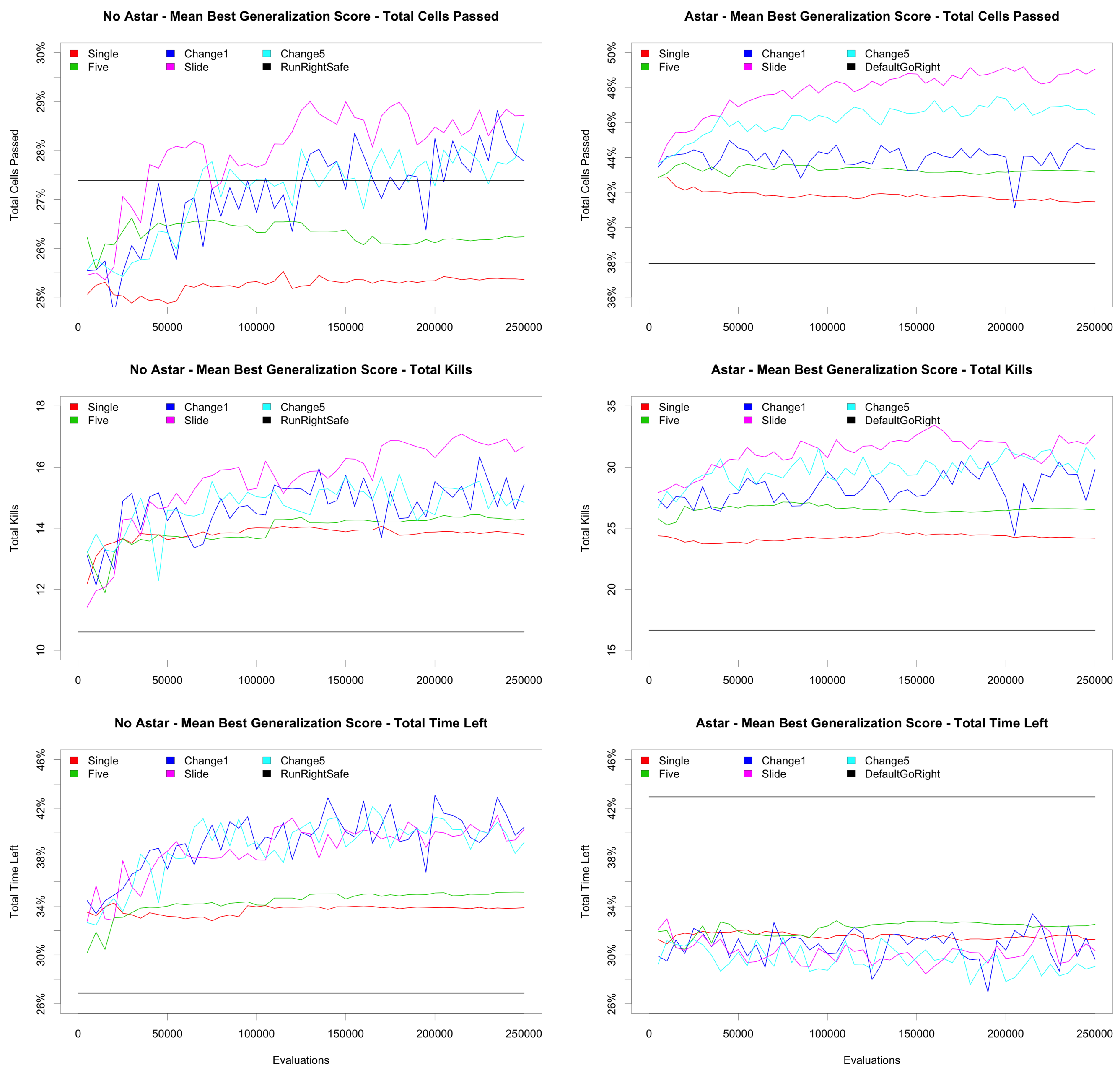


Fig. 10: Breakdown of average test performance of best evolved controllers, without A* (left) and with A* (right). The first row plots average percentage of passed cells; the second the average number of kills; and the last the percentage of time left.

routines dealing mainly with reactiveness were evolved and applied with priority, and a lower-priority A* approach was used for navigation. The latter gave the best results overall.

The combination of GE with BTs provides a flexible approach to Mario AI. The resulting solutions are human readable, and easy to analyze and fine-tune, addressing a concern of the game industry regarding evolutionary approaches. Also, the use of a grammar provides full control over the complexity of the resulting BT controllers; this facilitates the application of the current approach to games where BTs have been used previously (see Section II), as well as games where decomposition of behaviors is desirable [41], [28]. Finally, the combination of a carefully designed syntax with specific crossover locations allows the definition and exchange of Behavior Blocks, accelerating the evolutionary process.

Another challenge in dynamic game environments is the variety of environments and situations to face (and the disparity of fitness scores), leading to a risk of over-fitting specific game scenarios. This is particularly true in the Mario AI Benchmark, where levels created with the same difficulty setting can be drastically different, leading to noisy fitness landscapes. In this study, five approaches were used to deal with this issue.

A. Future Work

There are many avenues of research to improve this study. The fitness evaluation can be broken down into some of its

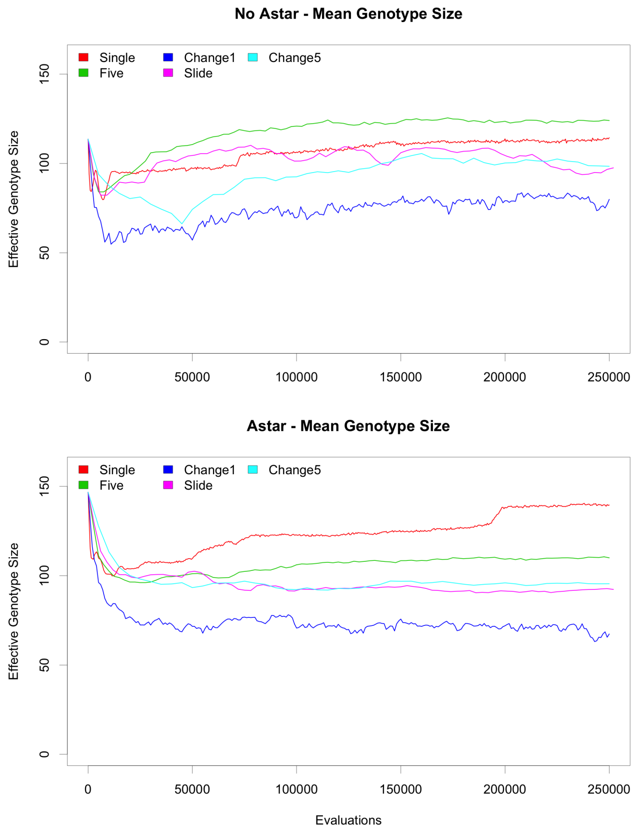


Fig. 11: Genotype solution size without (top) and with (bottom) A*, for the best individuals, averaged across 30 runs.

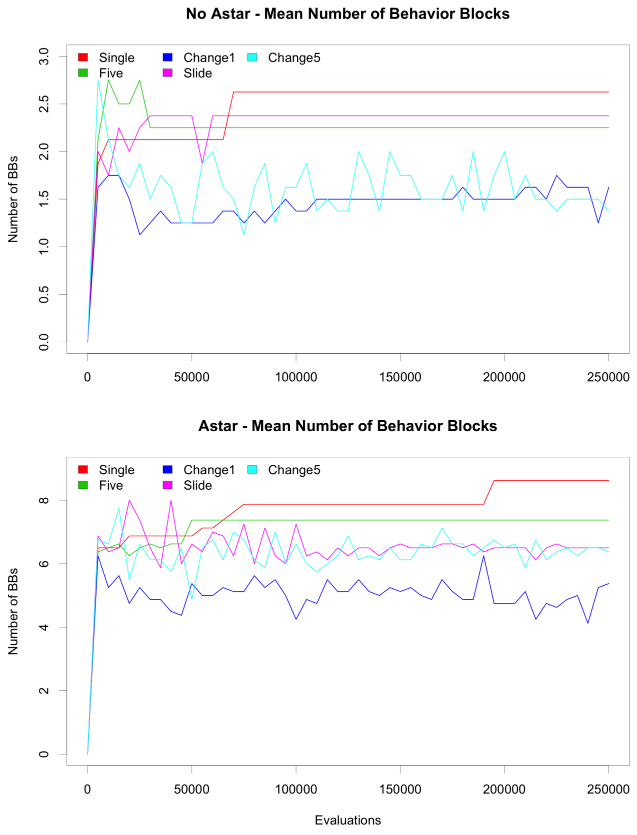


Fig. 12: Mean number of BBs for the best individuals, without (top) and with (bottom) A*, averaged across 30 runs.

constituents, such as distance traveled or number of kills; this information can be exploited with multi-objective approaches.

BB structures also present potential for further optimization. Frequency of use, number of kills, and complexity can be recorded for each BB; this would allow the BB-exchange operator to work more effectively. It would also allow optimization of each controller, by pruning non-executed BBs.

Finally, other approaches exist to deal with noisy fitness environments, such as moving average fitness calculations [23] or memory-based methods [25], amongst others (as pointed in Section II). Their incorporation into the presented system could lead to improved fitness.

Acknowledgments

This research is supported by the Science Foundation Ireland, under Grants No. 08/IN.1/I1868 and No. 13/IA/1850.

References

[1] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, ser. Genetic programming. Kluwer Academic Publishers, 2003, vol. 4.

[2] R. Colvin and I. Hayes, "A Semantics for Behavior Trees," ARC Centre for Complex Systems (ACCS), Technical Report ACCS-TR-07-01, 2007.

[3] S. Karakovskiy and J. Togelius, "The Mario AI Benchmark and Competitions," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4:1, pp. 55–67, 2012.

[4] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution," in *EvoApplications, European Conference on the Applications of Evolutionary Computation*, Cecilia Di Chio et al., Ed. Springer, 2011, pp. 123–132.

[5] ——, "Reactiveness and Navigation in Computer Games: Different Needs, Different Approaches," in *IEEE Conference on Computational Intelligence and Games*, 2011, pp. 273–280.

[6] D. Boutros, "A Detailed Cross-Examination of Yesterday and Today's Best-Selling Platform Games," http://www.gamasutra.com/view/feature/130268/a_detailed_crossexamination_of_.php?print=1, October 2015.

[7] J. Togelius, S. Karakovskiy, J. Koutnik, and J. Schmidhuber, "Super Mario Evolution," in *IEEE Symposium on Computational Intelligence and Games*, 2009, pp. 156–161.

[8] S. Bojarski and C. B. Congdon, "REALM: A Rule-based Evolutionary Computation Agent that Learns to Play Mario," in *IEEE Conference on Computational Intelligence and Games*, 2010, pp. 83–90.

[9] E. R. Speed, "Evolving a Mario Agent Using Cuckoo Search and Softmax Heuristics," in *Games Innovations Conference (ICE-GIC), International IEEE Consumer Electronics Society's*, 2010, pp. 1–7.

[10] J.-J. Tsay, C.-C. Chen, and J.-J. Hsu, "Evolving Intelligent Mario Controller by Reinforcement Learning," in *International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, 2004, pp. 266–272.

[11] H. Handa, "Dimensionality Reduction of Scene and Enemy Information in Mario," in *IEEE Congress on Evolutionary Computation*, 2011, pp. 1515–1520.

[12] N. C. Hou, N. S. Hong, C. K. On, and J. Teo, "Infinite Mario Bross AI using Genetic Algorithm," in *IEEE Conference on Sustainable Utilization and Development in Engineering and Technology (STUDENT)*, October 2011, pp. 85–89.

[13] A. M. Mora, J. J. Merelo, P. García-Sánchez, P. A. Castillo, M. S. Rodríguez-Domingo, and R. M. Hidalgo-Bermúdez, "Creating Autonomous Agents for Playing Super Mario Bros Game by Means of Evolutionary Finite State Machines," *Evolutionary Intelligence*, vol. 6, no. 4, pp. 205–218, 2014.

[14] E. J. Jacobsen, R. Greve, and J. Togelius, "Monte Mario: Platforming with MCTS," in *GECCO, Genetic and Evolutionary Computation Conference*, D. V. Arnold, Ed. ACM, 2014, pp. 293–300.
[15] J.-A. Meyer and D. Filliat, "Map-based Navigation in Mobile Robots: : II. A Review of Map-Learning and Path-Planning Strategies," *Cognitive Systems Research*, vol. 4, pp. 283–317, 2003.
[16] K. Birdwell, "The CABAL: Valve's Design Process for Creating Half Life," *Game Developer*, vol. 6 (12), pp. 40–50, 1999.
[17] S. Bandi and D. Thalmann, "Space Discretization for Efficient Human Navigation," *Computer Graphics Forum*, vol. 17, no. 3, pp. 195–206, 1998.
[18] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 Mario AI Competition," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 1–8.
[19] S. Shinohara, T. Takano, H. Takase, H. Kawanaka, and S. Tsuruoka, "Search Algorithm with Learning Ability for Mario AI – Combination A* Algorithm and Q-Learning," in *ACIS, International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing*, 2012, pp. 341–344.
[20] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis, "Generative Agents for Player Decision Modeling in Games," in *Poster Proceedings of the 9th Conference on the Foundations of Digital Games*, 2014.
[21] Y. Jin and J. Branke, "Evolutionary Optimization in Uncertain Environments a Survey," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 3, pp. 303–317, June 2005.
[22] C. Qian, Y. Yu, and Z.-H. Zhou, "Analyzing Evolutionary Optimization in Noisy Environments," *CoRR*, vol. abs/1311.4987, no. 4, 2013.
[23] A. D. Pietro, L. While, and L. Barone, "Learning in RoboCup Keepaway Using Evolutionary Algorithms," in *GECCO, Genetic and Evolutionary Computation Conference*, W. B. L. et al., Ed. Morgan Kaufmann, 2002, pp. 1065–1072.
[24] A. M. Mora, A. Fernández-Ares, J. J. Merelo, P. García-Sánchez, and C. M. Fernandes, "Effect of Noisy Fitness in Real-Time Strategy Games Player Behaviour Optimisation Using Evolutionary Algorithms," *Journal of Computer Science and Technology*, vol. 27, no. 5, pp. 1007–1023, 2012.
[25] J. J. Merelo, P. A. Castillo, A. Mora, A. Fernández-Ares, A. I. Esparcia-Alcázar, C. Cotta, and N. Rico, "Studying and Tackling Noisy Fitness in Evolutionary Design of Game Characters," in *International Conference on Evolutionary Computation Theory and Applications*, A. R. et al., Ed. SCITEPRESS, 2014, pp. 76–85.
[26] E. Galván-López, D. Fagan, E. Murphy, J. M. Swafford, A. Agapitos, M. O'Neill, and A. Brabazon, "Comparing the Performance of the Evolvable PiGrammatical Evolution Genotype-Phenotype Map to Grammatical Evolution in the Dynamic Ms. Pac-Man Environment," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 1587–1594.
[27] R. Harper, "Evolving Robocode tanks for Evo Robocode," *Genetic Programming and Evolvable Machines*, vol. 15, no. 4, pp. 403–431, 2014.
[28] A. Champandard, M. Dawe, and D. H. Cerpa, "Behavior Trees: Three Ways of Cultivating Strong AI," Game Developers Conference, Audio Lecture, 2010.
[29] D. Isla, "Managing Complexity in the Halo 2 AI System," in *Game Developers Conference*, 2005.
[30] L. McHugh, "Three Approaches to Behavior Tree AI," in *Game Developers Conference*, 2007.
[31] M. Mateas and A. Stern, "Managing Intermixing Behavior Hierarchies," in *Game Developers Conference*, 2004.
[32] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving Behaviour Trees for the Commercial Game DEFCON," in *EvoApplications, European Conference on the Applications of Evolutionary Computation*, ser. LNCS, Cecilia Di Chio et al., Ed., vol. 6024. Springer, 2010, pp. 100–110.
[33] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*, 1st ed. A Bradford Book, 1992.
[34] J. Togelius, N. Shaker, S. Karakovski, and G. N. Yannakakis, "The Mario AI Championship 2009-2012," *AI Magazine*, vol. 34, no. 3, pp. 89–92, 2013.
[35] M. Nicolau, "Automatic Grammar Complexity Reduction in Grammatical Evolution," in *GECCO, Genetic and Evolutionary Computation Conference Workshops*, R. P. et al., Ed., 2004.
[36] R. Harper, "GE, Explosive Grammars and the Lasting Legacy of Bad Initialisation," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 2602–2609.
[37] A. Brabazon and M. O'Neill, *Biologically Inspired Algorithms for Financial Modelling*. Springer, 2006.
[38] J. E. Murphy, M. O'Neill, and H. Carr, "Exploring Grammatical Evolution for Horse Gait Optimisation," in *EuroGP, European Conference on Genetic Programming*, ser. LNCS, Leonardo Vanneschi and Steven Gustafson, Ed., vol. 5481. Springer, 2009, pp. 183–194.
[39] N. J. Nilsson, *Artificial Intelligence, A New Synthesis*. Morgan Kaufmann Publishers, 1998.
[40] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 509–516, 1978.
[41] A. Champandard, "Behavior Trees for Next-Gen Game AI," Game Developers Conference, Audio Lecture, 2007.
[42] M. Nicolau and I. Dempsey, "Introducing Grammar Based Extensions for Grammatical Evolution," in *IEEE Congress on Evolutionary Computation*, 2006, pp. 2663–2670.
[43] M. Nicolau and D. Costelloe, "Using Grammatical Evolution to Parameterise Interactive 3D Image Generation," in *EvoApplications, European Conference on the Applications of Evolutionary Computation*, ser. LNCS, Cecilia Di Chio et al., Ed., vol. 6625. Springer, 2011, pp. 374–383.

Miguel Nicolau is a Lecturer of Business Analytics in the UCD School of Business, Ireland. He received his PhD degree from the University of Limerick in 2006, and then worked as an expert engineer at the INRIA lab in France, and as a Research Fellow with the Natural Computing Research & Applications Group In UCD. His main research interests are Genetic Programming, Grammatical representations, Data and Business Analytics, and advanced biological models.

Diego Perez-Liebana is a Lecturer in Computer Games and Artificial Intelligence at the University of Essex (UK), where he achieved a PhD in Computer Science (2015). He has published in the domain of Game AI, with interests on Reinforcement Learning and Evolutionary Computation. He organized several Game AI competitions, such as the Physical Travelling Salesman Problem and the General Video Game AI competitions, held in IEEE conferences. He has programming experience in the videogames industry with titles published for game consoles and PC.

Michael O'Neill is the ICON Chair of Business Analytics in the UCD School of Business, and is a founding Director of the UCD Natural Computing Research & Applications Group. He has published in excess of 250 peer-reviewed publications and has co-authored a number of successful funding applications with a total value over Euro 9 Million.

Anthony Brabazon is currently Associate Dean of the School of Business at UCD. His research interests concern the development of natural computing algorithms and their application to real-world problems. He is co-founder and co-director of the Natural Computing Research & Applications Group at UCD (see http://ncra.ucd.ie). Anthony has published in excess of 200 peer-reviewed studies and has authored / edited thirteen books.

Name	¬A*	A*	**Description**
			Conditions
CanIFire	✓	✓	Checks if Mario is able to shoot fireballs.
CanIJump	✓	✓	Indicates if Mario is able to jump (if he is on the ground).
IsFollowingPath	✓	✓	Indicates if Mario is following a path given by A*.
IsStuck	✓	✓	Checks if Mario has been idle for many cycles.
UnderBrick	✓	✓	Verifies if there is a brick block directly over Mario.
UnderQuestion	✓	✓	Indicates if there is a question brick block directly over Mario.
EnemyAhead †	✓	✓	Checks if there is an enemy ahead of Mario.
EnemyAheadUp †	✓	✓	Indicates if there is an enemy ahead and over Mario.
EnemyAheadDown †	✓	✓	Verifies if there is an enemy ahead and below Mario.
JumpableEnemyAhead †	✓	✓	Checks if there is an enemy that can be killed by stomping on it ahead of Mario.
NoJumpableEnemyAhead †	✓	✓	Indicates if there is an enemy that cannot be killed by stomping on it ahead of Mario.
IsBulletToHead	✓	✓	Checks if there is a bullet coming towards Mario at his head's height.
IsBulletToFeet	✓	✓	Indicates if there is a bullet coming towards Mario at his feet's height.
AvailableJumpAhead †	✓		Verifies if there are no obstacles over and ahead of Mario.
HoleAhead †	✓		Indicates if there is a hole ahead of Mario.
IsGapAhead †	✓		Indicates if there is a free space under an obstacle at Mario head's height, just ahead of him.
IsBreakableUp	✓		Checks if there is a breakable block directly above Mario.
IsBreakableUpAhead †	✓		Indicates if there is a brick block ahead and over Mario.
IsClimbableUp	✓		Verifies if there is a platform that can be reached jumping from below, directly over Mario.
IsClimbableUpAhead †	✓		Checks if there is a climbable platform ahead and above Mario.
IsJumpPlatformUpAhead †	✓		Verifies if there is a platform ahead and over Mario.
IsPushableUp	✓		Indicates if there is a question mark block directly over Mario.
IsPushableUpAhead †	✓		Checks if there is a question brick ahead and over Mario.
ObstacleAhead †	✓		Verifies if there is an obstacle ahead of Mario.
ObstacleHead †	✓		Indicates if there is an obstacle ahead of Mario, but only at his head's height.
			Actions
NOP	✓	✓	No action.
Down	✓	✓	Atomic action *Down*.
Fire	✓	✓	Atomic action *Fire*.
WalkRight †	✓	✓	Atomic action *Right*.
RunRight †	✓	✓	Combination of the atomic actions *Right* and *Fire*.
GetPathToClosestBrick		✓	Uses A* to get a path to the closest brick block to Mario.
GetPathToClosestQuestion		✓	Uses A* to retrieve a path to the closest question mark block to Mario.
GetPathToClosestItem		✓	Uses A* to get a path to the closest item to Mario.
GetPathToGround		✓	Uses A* to obtain a path to lowest position (or node) found in the grah of the level.
GetPathToTop		✓	Uses A* to get a path to highest position (or node) found in the grah of the level.
GetPathToClosestRightMost		✓	Uses A* to retrieve a path to rightmost position (or node) found in the grah of the level.
GetPathToClosestLeftMost		✓	Uses A* to obtain a path to leftmost position (or node) found in the grah of the level.
			Filters
Loop	✓	✓	Repeats the execution of its child sub-tree N times.
Non	✓	✓	Negates the result given by its sub-tree.
UntilFails	✓	✓	Repeats the execution of its sub-tree until it receives the result *failure*.
UntilFailsLimited	✓	✓	Repeats the execution of its child sub-tree N times or until it receives the result *failure*.
			Sub-Trees
UseRightGap †	✓		This sub-tree moves Mario to the right until there is a vertical over him with no blocks. Then, it jumps to try to reach a higher platform and continue from there.
AvoidRightTrap †	✓		This sub-tree attempts to overcome a dead end. It first takes Mario back to the point where there was a bifurcation in the path. Then, it uses *UseLeftGap* to take a secondary path in order to avoid the dead end.
GoUnderRight †	✓		This sub-tree detects situations where it is not possible to go further, but there is a gap ahead of Mario, which could be traversed if Mario would be small, or by running towards the gap, crouching down and sliding. This sub-tree executes the latter sequence of actions.
DefaultPathPlanner		✓	Calculates the path to the rightmost position on the screen.
PathFollower		✓	Follows the last path calculated.
JumpRightLong †	✓	✓	Makes a long jump to the right (see Figure 4). The filter executes the *JumpRight* action (that enables the buttons *jump* and *right*) for 9 game cycles.
JumpRightShort †	✓	✓	As *JumpRightLong*, but the filter executes the *JumpRight* action during 3 steps.
JumpRightRunLong †	✓	✓	As *JumpRightLong*, but the loop action is JumpRightRun (that enables the buttons *jump*, *right* and *run*).
JumpRightRunShort †	✓	✓	As *JumpRightShort*, but the loop action is JumpRightRun (that enables the buttons *jump*, *right* and *run*).
WalkRightSafe †	✓	✓	Moves Mario to the right, checking for obstacles, enemies and holes. If any of these is detected, the agent tries to avoid it with a long jump (or short jump, if it is an enemy that can be killed by stomping on it).
RunRightSafe †	✓	✓	As *WalkRightSafe*, but the input *run* is always on.
VerticalJumpLong	✓	✓	As *JumpRightLong*, but the loop action is Jump (that only activates the input *jump*).
VerticalJumpShort	✓	✓	As *JumpRightShort*, but the loop action is Jump (that only activates the input *jump*).

TABLE IV: Actions, conditions, filters and subtrees that can be used by the evolutionary algorithm. †Denotes sub-trees that have an analogous *left* (or *back*) variant. Note that some actions and conditions can be analogous in both the controllers with and without A* (i.e. *IsBreakableUp* vs. *UnderBrick*); they are, however, different: while the A* version checks the nodes in the graph, the no-A* implementation needs to analyze the contents of each cell. Finally, note that actions use the terms *left* and *right*, which imply movement, while conditions use *ahead* (for right) and *back* (for left).