Evolving Behaviour Tree structures using Grammatical Evolution

Diego Perez-Liebana and Miguel Nicolau

Abstract Behaviour Trees are control structures with many applications in computer science, including robotics, control systems, and computer games. They allow the specification of controllers from very broad behaviour definitions (close to the root of the tree) down to very specific technical implementations (near the leaves); this allows them to be understood and extended by both behaviour designers and technical programmers. This chapter describes the process of applying Grammatical Evolution (GE) to evolve Behaviour Trees for a real-time video-game: the Mario AI Benchmark. The results obtained show that these structures are quite amenable to artificial evolution using GE, and can provide a good balance between long-term (pathfinding) and short-term (reactiveness to hazards and power-ups) planning within the same structure.

1 Evolving Behaviour Trees for Game Playing Control

The objective of this chapter is to demonstrate how Grammatical Evolution (GE) [1] can be used to evolve control structures for agents that interact in highly dynamic environments. An example of this application is the control of Non-Player Characters (NPCs) in computer games. These are highly dynamic environments, where NPCs must be able to react efficiently and effectively in previously unseen scenarios. In the concrete case of real-time games, these agents must be able to perform actions in a limited time budget, typically in just a few milliseconds. These actions must tackle both dynamic (enemies, power ups, moving structures) and static (level

Diego Perez-Liebana
School of Computer Science and Electronic Engineering, University of Essex, Colchester, UK, e-mail: dperez@essex.ac.uk

Miguel Nicolau
Natural Computing Research and Applications Group, University College Dublin, Ireland, e-mail: Miguel.Nicolau@ucd.ie

structure, path-finding) elements in the game. This requires the agent to be able to react to imminent hazards, as well as devising action plans to accomplish the goals that lead to winning the game.

An example of a platform game with these characteristics is the Mario AI Benchmark [2, 3]. In this environment, the goal is to reach the end of the level, avoiding (or killing) enemies or other hazards that may harm the player. The agent (Mario) must react to dynamic events that happen at close distance, and also plan ahead to make progress in the level. Section 3 describes this benchmark, the environmental information and the avatar effectors.

These two components, reactiveness and navigation, are closely related in this game environment: both make use of the same set of actuators (or actions) that Mario can employ. Section 4 describes two different approaches for tackling them, one using the same actuators for both needs, and another providing different sub-behaviours that deal with them separately.

Previous studies [4] have successfully applied GE to evolve Behaviour Trees (BTs) [5] for the Mario AI Benchmark. A BT is a tree structure composed of different types of nodes, including control nodes that permit control over the flow of execution, condition nodes that query the game state, and action nodes that execute actions in the game. Section 5 describes how BTs work and how they are implemented for the framework used in this study, and Section 6 details how to implement these structures using GE.

A set of approaches is compared in terms of evolvability, generalisation, and complexity of resulting controllers. Sections 7 and 8 describe the experimental study and results obtained, with conclusions and recommendations for their applicability, both to Mario AI and other dynamic environments, given in Section 9.

2 Related Work

This section reviews applications of GE to control environments including games, other approaches to create controllers for the Mario AI benchmark, and applications of BTs as game controllers.

From the onset, GE practitioners have used its grammar-based syntax specification to solve a multitude of problems, including controllers for a diverse range of environments. A typical example is the use of GE to solve the Santa Fe Ant Trail problem [6], which rapidly became a typical benchmark over many years. Other examples include its application to the Lawn-Mower problem [7], and its combination with a gene regulatory network, to solve the pole-balancing problem [8]. Regarding gaming environments, examples include the work of Galván-López et al. [9], who evolved controllers for Ms. PacMan, and Harper [10, 11], who used GE to co-evolve controllers for Robocode Tanks.

The application of GE to game environments is not limited to control agents, however. Other interesting applications include the design and optimisation of horse

gate animations [12], and the design of levels [13] and personalised content [14] for the Mario AI environment.

Mario AI provides a suitable platform for research in the application of controllers to dynamic environments. Multiple approaches that aim at maximising the game score can be found in the literature. These include rule-based agents with higher-level, hand-designed conditions and actions [15], cuckoo search and its comparison with a standard genetic algorithm approach [16]), the evolution of finite-state machines created with genetic algorithms [17, 18], Q-Learning with full game information [19], Neural-Networks with Manifold Learning as a dimensionality-reducing technique [20] and, finally, the combination of Monte Carlo Tree Search with appropriate heuristics [21], creating agents that outperform the leading state of the art controllers in this game.

Most game environments are highly dynamic environments, often resulting in noisy fitness evaluation. This is also true for Mario AI, where a random seed can vary the events and levels generated with the same difficulty level, presence and absence of enemies, etc. There exists a large body of research in the area of noisy fitness environments. A recent example of such studies are the works of J. Liu et al. [22] and K. Kunanusont [23], who evolved game parameters for AI-assisted game design in a search space with noisy and expensive evaluations.

BTs, initially introduced as a means to encode formal system specifications [5], have gained popularity as a way to encode game controllers in a modular, scalable and reusable manner [24]. They have now been used in high-revenue commercial games, such as "Halo" [25], "Spore" [26], and other smaller indie games (such as "Façade" [27]) illustrating their importance in the game AI community.

The evolution of BT structures has been explored in the work of Lim et al. [28], where the authors used Genetic Programming [29] to evolve AI controllers for the *DEFCON* game. In this study, the resulting agent played against the standard *DEFCON* AI controller, achieving a success rate superior to 50%.

One of the main hurdles encountered in the work of Lim et al. was how to exchange typed BT structures between individuals. This issue is easily dealt with GE, which was also used to evolve BTs, as controllers for Mario AI [4]. The current chapter details those experiments, giving insight on the actual process of specifying the syntax of BTs through a grammar, and maximising the exchange of coherent genetic material between solutions, increasing the effectiveness of the search process.

3 Mario AI as a Dynamic Game Benchmark

Super Mario Bros is a popular two-dimensional platform game where the player controls Mario, who must reach the right end of the level by avoiding enemies, other hazards, and collecting bonus items and power-ups. Therefore, the Mario AI benchmark exemplifies a highly dynamic environment with a final goal that requires long term planning. This benchmark, an adaptation of an open source version of *Super Mario Bros* (by Markus Persson), is used for the experimental work developed

for this chapter. The framework allows testing agents in multiple levels, customising them 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.

3.1 Game State Information

The playing agent is able to analyse the environment surrounding Mario by means of two matrices, one providing information about the geometry of the level, and the other indicating the presence of enemies.

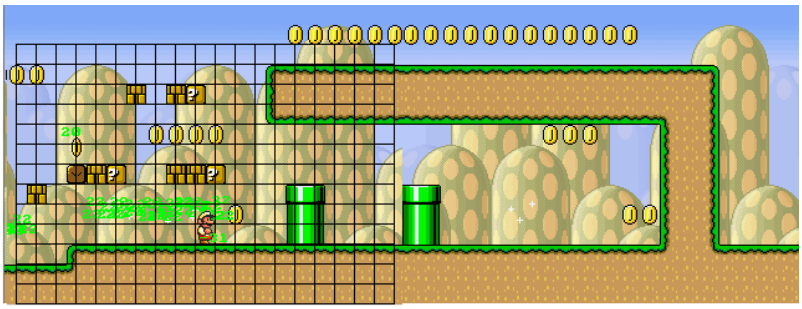


Fig. 1: Mario and environmental information. Both matrices are of size 21×21, centered in Mario.

The level of detail to be retrieved from both matrices can be set to three values: **Zoom 0** represents the world with a different integer for each entity in the game, whereas **Zoom 2** gives the simplest possible representation (1 meaning enemy or obstacle presence, and 0 the absence of it). As a mid point, **Zoom 1** provides the information categorised in useful groups, such as enemies that can be killed by stomping or shooting, different types of blocks, etc.

Mario itself can be in three different states during the game: *Small*, where an enemy or hazard hit causes the player to lose; *Big*, reachable by eating a mushroom from the *Small* state; and *Fire*, which permits the player to shoot fireballs. Being hit by an enemy changes Mario's state to the previous one. The benchmark also provides information about the state of the agent, plus its location in the level and 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 (game running, won or lost), the time left to complete the level, the current score and a set of *kill statistics*, including the number of enemies killed and how they were eliminated (by stomp, by fire or by shell).

3.2 Game Actions

Mario can choose among several actions to be performed at each game step. These actions include three directions (*Left*, *Right* and *Down* - *Up* has no meaning in this implementation), *Jump*, and *Run/Fire*. If the agent is already moving right or left, applying the action *Run/Fire* makes Mario move faster. If *Jump* and *Run/Fire* are applied simultaneously, Mario jumps farther. Also, when in *Fire* mode, it makes the agent shoot a fireball. Therefore, the set of possible actions defines an action space of $2^5 = 32$ actions (although some of these are nonsensical, such as left and right pushed at the same time). As the agent is played in real-time, the action supplied at each frame must be provided every $40ms$, or the agent will be disqualified.

4 Mario Agents

The focus of this work is on the evolution of BT data structures that allow a hierarchical decomposition of tasks, by means of GE. The evolved structures need to be able to respond adequately to scenarios that propose dynamic hazards and an overall goal. In the case of Mario, GE evolves a BT that combines the two required aspects of the agent behaviour for this game, also identified in the introduction: reactiveness (dealing with close enemies and hazards) and navigation across the static elements in the level. Both were dealt with using basic game movements or combinations of these. Tables 4 and 5, at the end of the chapter, show the routines employed by GE, along with a brief explanation of each.

Two different approaches are analysed in this study, in order to assert the importance of the navigation component of the algorithm's behaviour. Each approach use a different set of routines. *ReactiveMario (NoAstar)* combines reactive and very basic navigation commands, while *PlanningMario (Astar)* uses A* for navigation in order to let the GE focus on the reactive part of the behaviour. Section 6 details the general structure of the evolved BTs and how reactiveness and navigation are integrated into a single approach.

4.1 A Reactive Mario

In this approach (NoAstar), the agent is exclusively focused on reacting to moving elements in the game, without employing any explicit path-finding. Therefore, the elements considered are the position of enemies and hazards, such as *goombas*, bullets, flying turtles, bonus mushrooms and fire flowers. GE is used here to evolve a BT that avoids these entities and navigates the agent through the levels. This controller, submitted to the Gameplay track of the 2010 Mario AI Competition, ranked 4^{th} out of 8 entries [3] in this contest.

In this edition, one of the most difficult navigational hazards were dead ends. In those, the level presents more than one way to move ahead, but at least one of them is a *cul de sac*. An example of one of these is shown in Fig. 1, where it can be seen that the *cul de sac* is longer than the size of the environmental matrix. Two sub-trees (*UseRightGap* and *AvoidRightTrap*, see Table 5) have been specifically designed by hand to address this problem. The latter routine detects a dead end in front of Mario and moves him back until there is no obstacle over his head. The former sub-tree finds a platform which Mario can jump onto (or a gap to fall through), to overcome the trap by running through the open part.

4.2 A Planning Mario

This second approach (Astar) employs A* to guide navigation through the level, while handing GE the task of dealing with reactiveness to the hazards of the game. However, in order to use A* for navigation, the game level must be represented as a navigable graph, a structure not supplied by the benchmark. Furthermore, due to the nature of the game, this graph needs to be modifiable. Changes in the blocks (which can be destroyed by Mario) or changes in the state of the avatar (i.e., from *Big* to *Small*) can make some old paths invalid. Therefore, this graph must be generated *dynamically*, at each step, by the agent. It is important to note that the map building process is independent from the use of GE to evolve the playing agents and it is not a functionality provided in the framework. The present section briefly summarises this process.

4.2.1 Mapping a level

The first problem when dealing with path planning is the world representation. As the Mario AI Benchmark does not provide access to a complete map of the level, this has to be built as Mario moves through it, using the environment arrays described in Section 3.1. As this map is created for navigation, blocks that do not affect movement (items, enemies or coins) are not taken into account. However, it is possible to add 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.

4.2.2 Nodes for Path Planning

Once a representation of the geometry of the level is available, a graph for the A* algorithm can be built. Given the format of the data, the best solution is to build a tile-based graph approximation.

The first decision is where to place the nodes (or vertices) of the graph within the map; these nodes represent navigable spaces, i.e. positions where Mario can

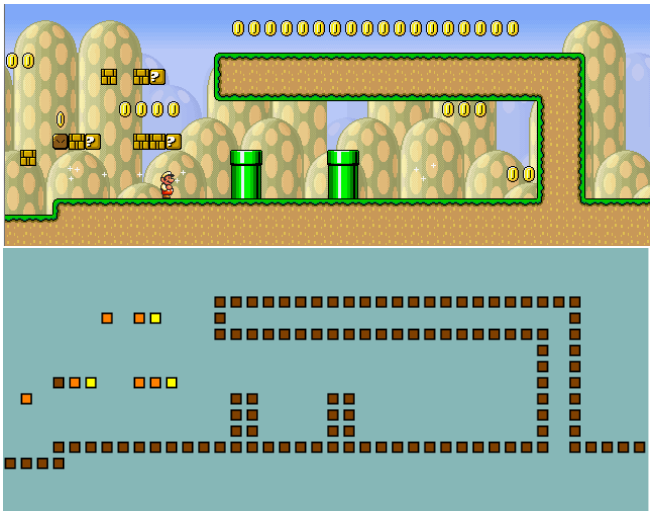


Fig. 2: Top: the original section of the level as seen in game. Bottom: the respective map generated from the environmental matrix received by the agent.

stand. Fig. 3 (left) shows the level structure (as squares) and the graph nodes (as dots). It is worthwhile mentioning that this process identifies some nodes that are not accessible (such as the nodes inside the ceiling of the dead end), but the next step will filter these out of the graph in which Mario is located and can move.

4.2.3 Edges for Path Planning

Although most grid-based path finding networks consider the map as seen from a zenithal perspective (i.e. from the top), this game requires the graph built as seen from the player's perspective, sideways to the level. This incorporates an additional challenge, where horizontal and vertical edges cannot be used in the same way.

The edge creation process analyses the nodes to finish the graph construction, using different types of links. The following links are available for the graph, which are also shown in the example Fig. 3 (right).

- **Walk links**: These are the simplest ones, which can be used just by applying the right (or left) actions. These edges are bidirectional.
- **Jump links**: Unidirectional upward edges that join nodes vertically separated by no more than 3 cells and horizontally by 1 position. Therefore, they can be used to jump to a node that is over the starting node (with a maximum jump height) and one unit to the left or right.
- **Vertical jump links**: Some level formations can be jumped onto from below, keeping the same vertical. For these structures, unidirectional upward edges are created, which join nodes vertically separated by no more than 3 cells.

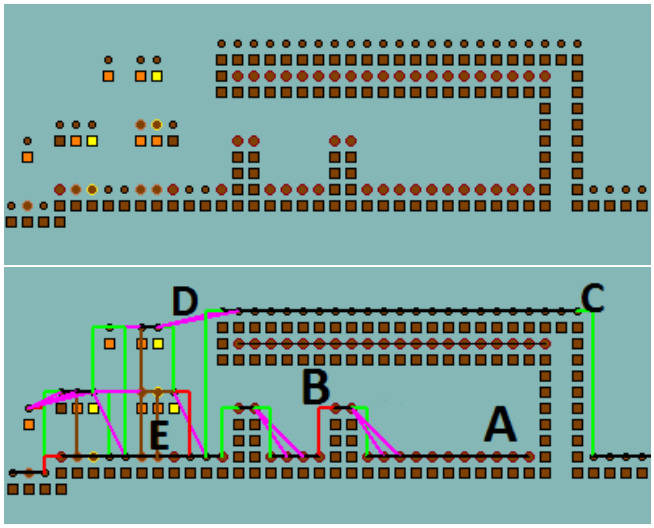


Fig. 3: Top: level structure (squares) and graph nodes (dots). Colours indicate different types of block - question mark, brick, or solid. Bottom: navigation graph representation. Different types of edges are Walk link (A), Jump link (B), Fall link (C), Faith jump link (D) and Break jump link (E).

- **Fall links**: Unidirectional downward edges that join nodes vertically separated by any number of cells and horizontally by 1 position. Some of these links have as a counterpart a *jump* edge (some jump links can also be used to fall in the opposite direction). This distinction is important, because while the former have to be managed by jumping, the latter must be gone through moving in one direction and managing the fall in order to land in the proper place.
- **Faith jump links**: Bidirectional edges that link two nodes horizontally separated by no more than 4 cells. These edges are used to link nodes that are separated horizontally by more than one unit, and with a maximum vertical distance. These can be, in some cases, hard jumps to make, because of the long distance between starting and ending node.
- **Break jump links**: These are very similar to normal jump, but in this case there is a brick block in the trajectory of the jump, in the vertical of the node where the edge starts. Because this block can (potentially) be destroyed, this link is included in the graph as it can become a regular *jump* link. It is also possible that the brick block does not break (becoming a solid block instead) and the link cannot be used. In that case, as the map and graph are repeatedly generated, this link will not be created again.

An important aspect of this game is that Mario can have different states (*Big* or *Small*), and some edges can only be traversed if Mario is *Small* (only one cell is required to pass through). This information is also included in the edges, and it is used not only to traverse the graph, but also to compute the cost of the edges for A*.

The *basic cost* of an edge is calculated as the Manhattan Distance[1] between its nodes. The *final cost* is computed as the basic cost multiplied by a factor determined by the link type. The reasoning is as follows: the factor of traveling an edge walking must be higher than jumping, because calculating the jump and managing landing takes longer and it has a higher associated risk (it is more likely to miss a jump than a simple walk movement). The *basic cost* of each link is therefore multiplied by a factor of 1.5 if the link involves a jump, with the exception of a *break jump* link, which factor is 3.0 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, it is possible to design path finding 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.

5 Behaviour Trees for Decision Making

Behaviour Trees (BT) are data structures that allow to decompose a complex behaviour hierarchically in several sub-trees as tasks of reduced complexity. Their applicability is broad in the fields of AI and technology, such as management of control systems [30], robotics [31] and decision making behaviours in video-games [24, 25, 26], an area in which BTs have achieved great popularity.

In fact, along with Finite State Machines (FSM), BTs are one of the most prolific structures to implement complex Non Player Character (NPC) behaviours in games. In contrast to FSMs, BTs are more flexible, scalable and intuitive, easier to develop (even for non-technical developers and designers) and are able to incorporate multiple concerns such as path planning and path following [32].

For example, a soldier NPC can have different behaviours, such as patrolling, investigating and attacking. Each one of these tasks can be broken down in different sub-tasks (movement tactics, weapon management or aiming algorithms), which at the same time can be composed of lower level actions (playing sounds or animations). In the robotics domain, high level goals can be broken down in sub-goals, like recharging batteries or entering rooms, which in turn can be decomposed in opening, closing and forcing doors [31].

BTs establish a descending order of complexity from the root to the leaves, employing different node types. In the simplest implementation, all nodes can return *success* or *failure* to their parent node (although other versions could return real values, enumerators, etc.). Nodes are divided into two major categories: the first type is control nodes, which manage the flow of execution through the tree:

- **Sequence** nodes execute its children from left to right until one returns *failure*. If all children return *success*, the sequence node itself also returns *success*. Otherwise, it will return *failure* to its parent. These nodes are represented with a right pointing arrow.

[1] The sum of absolute differences in Cartesian coordinates.

- **Selector** nodes execute its children from left to right until one returns *success*. If all children return *failure*, this node will return the same to its parent, and *success* otherwise. These nodes are represented with a question mark.
- **Parallel** nodes execute all its children in parallel. Termination conditions and return values can be diverse (i.e. breaking and returning the value from the first child to finish, returning a majority vote, etc). These are typically represented with parallel right pointing arrows.
- **Decorator** or **Filter** nodes modify the normal execution flow in different ways (negating the value of its child node, loops, running a node until *failure*, etc). These nodes typically have only one child. Decorators are normally represented with a diamond shape.

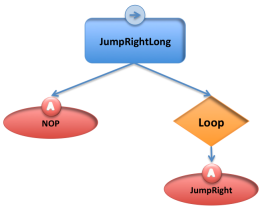
Leaf nodes are **Conditions** and **Actions**. The former query situations and features of the current environment, while the latter apply moves in the scenario the agent is in. Actions usually return *success* unless the action was not possible to be executed for some reason, while the returned value of a condition node depends on the query performed.

BTs can also incorporate handlers for data sharing and sub-behaviour re-use. For the former, *Blackboards* allow passing information between nodes and trees, and it is possible to introduce management mechanisms to coordinate access and usage of resources. For the latter, it is possible to use look-up tables to build a BT library that allows the designer to re-use sub-trees in multiple locations of the overall BT.

5.1 Behaviour Trees and Mario AI

Tables 4 and 5 (at the end of this chapter) include all conditions, actions, filters and sub-trees designed for the agent, and available to the evolutionary algorithm. 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. The leaf nodes described in these tables are also summarised here:

- **Conditions**: Used to provide information about the enemies (distance to the avatar and their type) and obstacles in the map (type and position of the blocks).
- **Actions**: The most useful action combinations are provided to the BT, based on those described previously (see Section 3.2). 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 units are indivisible and require a concrete sequence of moves to be performed. Jumps are more effectively managed in sub-trees, as they require a frame in which the jump button is not pressed before the jump action is executed, and farther jumps can be made with consecutive repetitions of the key being pressed. Sub-trees are achieved by combining different filter and action nodes. Fig. 4 shows the sub-tree to make long jumps to the right.




```
<?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>
```

Fig. 4: Sub-tree for executing long jumps to the right. `NOP` ensures no button is pressed before the `JumpRight` command (Jump plus Right actions) is executed. The `Loop` filter makes sure `JumpRight` is executed during a given number of frames. On the left, graphical representation. On the right, BT XML Structure of this sub-tree.

In this work, the BTs used are stored in XML files, with a hierarchical structure that defines the type of each node and the operation that it represents. Fig. 4 includes the (simplified) XML code of the sample sub-tree from this chapter. Note that this implies that the GE grammar must be able to generate the behaviours in this format.

6 Building Behaviour Trees with Grammatical Evolution

Fig. 4 shows the complexity of the syntax of the BT Controllers. GE's use of grammars can ensure that the evolved controllers maintain syntactic correctness, both keeping compatibility with the variety of BT control and leaf nodes, and also incorporating domain knowledge (as explained below). This is possible through the use of the grammar as an instrument to control of the syntax of solutions both in terms of data-structures and biases [33, 34].

6.1 Structure of a Behaviour Tree

The grammar employed to evolve BTs specifies the (XML) syntax, containing all conditions, actions, sub-trees and filters designed. Earlier experimentation, where GE was free to combine these nodes without a rigid structure, showed that the evolved trees were badly structured (such as sequences of sequences, with `NOP` actions at their leaves), not human-readable, and computationally demanding.

In order to avoid these issues, a constrained structure for the syntax of BTs was imposed via the grammar. Although still variable in size, the BT structures are contrived to follow an *and-or* tree structure [35], the recommended way of building BTs for game AI [36]. Therefore, all evolved BTs have a selector node at the root with a variable number of *Behaviour Block* (BB) sub-trees, each one of them encoding a particular sub-behaviour. Each one of these BBs consists of a sequence one or more conditions, followed by a sequence of sub-trees or atomic actions.

The last child of the root node is an unconditioned BB, which is either a sequence of actions and sub-trees, or a default navigation behaviour (when using the Astar agent). Fig. 5 exemplifies this structure.

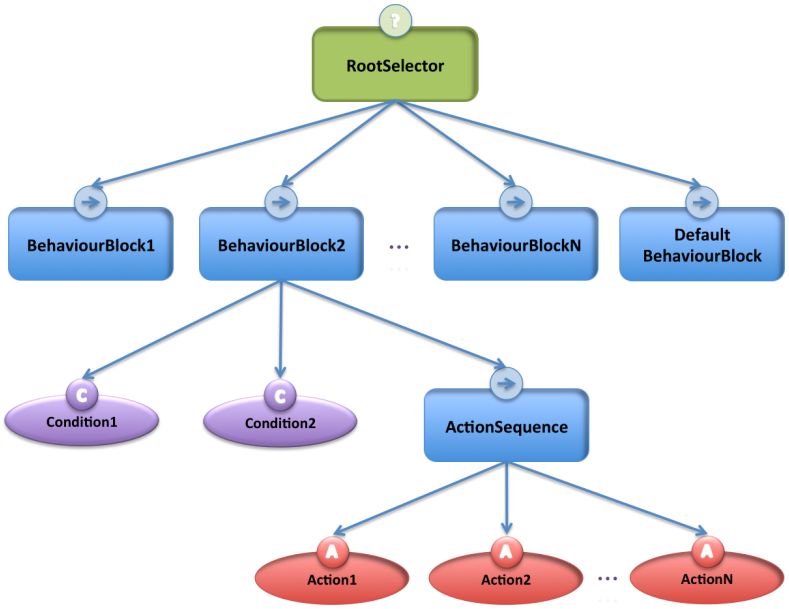


Fig. 5: Structure of evolved BTs.

The root selector starts with the leftmost BB at the beginning of the BT execution. The BT will follow a left-to-right priority order in which each sub-tree will be executed if its stated condition(s) are fulfilled. Hence, the right-most block is the default behaviour, which will be executed in case all the previous conditioned BBs fail to trigger. The conditions provided are complex queries to the game state, which allows limiting the number of conditions associated to each BB to just one or two. The number of actions and sub-trees on each BB is left unlimited.

The default behaviour (right-most block) depends on the navigation used: without A*, it is simply composed of a sequence of actions, without associated conditions. However, if A* us used, a particular sub-tree is provided. Fig. 6 shows this structure (sub-tree `Default Go Right`), which is composed of a selector node, with two sub-trees, `Default Path Planner` and `Path Follower`.

`Default Path Planner` is composed of two sub-trees. `Recalculate` decides if the default path has to be calculated; that can happen when no item is being targeted, or when a path was set but is almost finished. `Path Planner` (executed if the one before is successful) calculates the path to the rightmost position in the map (the direction to follow to the end of the level). In the rare event where

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

a path is not found, Mario enters an emergency state: to keep moving, a default forward jump is executed.

6.2 Designing the Grammar

Given that each BB is a self-contained structure, due to the syntax described above, it is reasonable to permit individuals to exchange these between them. Specific crossover points were thus encoded into the grammar to allow this, using a special grammar symbol (`<GEXOMarker>`) to label crossover points [37, 38]. These constrain GE to only slice individuals according to these points, when applying crossover; a two-point crossover was employed to guarantee this. Without these markers, standard 1-point crossover would provide more exploration with less exploitation; given the high cost of the fitness function, this trade-off was necessary. This technique creates an operator similar to sub-tree crossover in GP, but allowing the exchange of a variable number of behaviour blocks between individuals. Finally, it is also possible for an individual to crossover with himself, which is equivalent to a sub-tree swap operator. In the presented grammar, this is very useful to modify the priority of a BB.

An extract of the grammar used (Astar variant) is shown in Fig. 7. The `<BT>` symbol merely defines the XML prelude, while the `<XMLPart>` symbol provides

specific implementation BT tags. The following two symbols (<RootNode> and <RootSelNode>) define the and-or tree selector root node.

Its contents are defined in <SeqsAndDefBB>: a set of sequences, followed by the default behaviour. A first crossover marker is defined here, as the right-most place at which crossover can occur (just after all the defined filtered sequences, but before the default behaviour, which is equal for all individuals).

The set of filtered sequences (i.e. BBs) is defined by the symbol <SeqNodes>, which is just a recursively defined list of one or more BBs. Each of them, however, defined by <SequenceNode>, places a crossover marker at the start (i.e. left) of the definition of a BB, allowing for the exchange of BBs between individuals.

The definition of the remainder of the XML syntax proceeds in a similar fashion, and most of it is not shown for lack of space (the grammar is composed of 48 non-terminal symbols, and 541 lines). The definition of the Default Go Right sub-tree is worth mentioning (symbol <DefaultGoRight>): GE grammars can define a large sequence of text as a single terminal symbol.

7 Experimental Work

A series of experiments were ran to test several aspects of this approach. To ascertain if BTs are a good structure to evolve 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 measured if the separation of reactive and navigation routines led to improved results. Section 7.2 discusses in detail the different evolutionary approaches tested, aiming to deal with the noisy fitness evaluation that highly dynamic environments such as games can present.

7.1 Evaluating Behaviour Trees

A set of game levels is generated to test each evolved controller. Each *mapset* is composed of 10 levels (5 difficulty settings, with and without enemies), and is generated with a single random seed. The resulting fitness value (to be maximised) is a weighted sum of distance traveled and the actual Mario AI Benchmark score (which includes enemy kills and collected items). Both game levels and BT controllers are deterministic: the same controller in a map always yields the same fitness.

7.2 Generality of Controllers

The Mario AI benchmark is able to randomly create multiple levels, ranging from very easy to physically impossible to terminate. Such a dynamic environment poses

```
<BT> ::= '<?xml version="2.0" encoding="utf-8"?>\n' <XMLPart>
<XMLPart> ::= '<Behavior>\n' <RootNode> '</Behavior>\n'

<RootNode> ::= '<Node Type="Root">\n'<RootSelNode>'</Node>\n'
<RootSelNode> ::= '<Node Type="Selector">\n' <SeqsAndDefBB> '</Node>\n'
<SeqsAndDefBB> ::= <SeqNodes> <GEXOMarker> <DefBB> '\n'

<SeqNodes> ::= <SequenceNode> | <SeqNodes> <SequenceNode>
<SequenceNode> ::= <GEXOMarker> '<Node Type="Sequence">\n'
                   <1to2Conditions> <FilterSeqActNLUTs> '</Node>\n'

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

<FilterSeqActNLUTs> ::= <FilterHeader> <SeqActNLUTs> '</Node>\n'
                      | <SeqActNLUTs>
<FilterHeader> ::= <Loop> | <NON> | <UFL>
<Loop> ::= '<Node Filter_Type="Loop" Times="'<I>'" Type="Filter">\n'
<NON> ::= '<Node Filter_Type="NON" Type="Filter">\n'
<UFL> ::= '<Node Filter_Type="Until_Fail_Lim" Times="'<I>'" Type="Filter">\n'
<I> ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<SeqActNLUTs> ::= '<Node Type="Sequence">\n' <1PlusActOrLUTs> '</Node>\n'
<1PlusActOrLUTs> ::= <ActionOrLUT> | <1PlusActOrLUTs> <ActionOrLUT>
<ActionOrLUT> ::= <ActionNode> | <LUTNode>

<DefBB> ::= '<Node Type="Sequence">\n' <DefaultGoRight> '</Node>\n'
<DefaultGoRight> ::= '<Node Name = "DefGoRight" Type="Selector">
  <Node Name = "DefPathPlanner" Type="Sequence">
    <Node Name="Recalculate" Type="Selector">
      <Node Filter_Type="NON" Type="Filter">
        <Node Operation="IsFollowingPath" Type="Condition"/>
      </Node>
        <Node Operation="IsRightMostCloseToEnd" Type="Condition"/>
      </Node>
    <Node Name="PathPlanner" Type="Selector">
      <Node Operation="GetPathToRightMost" Type="Action"/>
      <Node Operation="NoPathAction" Type="Action"/>
      <Node Name="JumpRightRunLong" Type="Sequence">
        <Node Operation="NOP" Type="Action"/>
        <Node Filter_Type="Loop" Times="25" Type="Filter">
          <Node Operation="JumpRightRun" Type="Action"/>
        </Node>
      </Node>
    </Node>
  </Node>
  <Node Name="Path Follower" Type="Sequence">
    <Node Operation="IsFollowingPath" Type="Condition"/>
    <Node Operation="FollowCurrentPath" Type="Action"/>
  </Node>
</Node>'
```

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

a generalisation problem, and the following approaches were tested aiming at tackling this issue (note that both non-A* and A* versions of these were used):

- **Single**: this approach always uses the same mapset for evaluation, with the same seed for all independent runs.
- **Five**: in this case, the same five mapsets are used to test each controller (kept for all runs). This increases the variety of situations each controller is evaluated on.

Table 1: Experimental Setup

GE	Population Size	500
	Evaluations	250,000
	Derivation-tree Depth Range (for initialisation)	20...30
	Derivation-tree Max Depth	*unset*
	Tail Ratio (for initialisation)	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

- **Change1**: this approach uses only one mapset for evaluation, but changing it at each generation. The same sequence of mapsets is used in all runs. This approach increases the variety of situations seen for each controller, while keeping the evaluation effort small. To ensure continuity between generations, the parent population is reevaluated with the new generation's mapset.
- **Change5**: this case uses five mapsets for each evaluation, but all five are replaced at each generation (same sequence for all runs). The parent population is reevaluated with the new mapsets at the start of each new generation.
- **Slide**: this approach also uses five mapsets for each evaluation, replacing one mapset with a new one at every generation, in a sliding window manner (`12345`, `23456`, etc.). The same sequence is kept for all runs. The parent population is reevaluated with the new five mapsets at the start of each new generation.

Each of the 10 systems evaluated (five approaches described in Section 7.2, with and without A*) used the setup shown in Table 1.

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.

8 Results

8.1 Performance on Training

The mean best controller training score for all approaches is shown in Fig. 8. These plots also include the average performance of the respective reference behaviour (*RunRightSafe* without A*; *DefaultGoRight* for A*)[2].

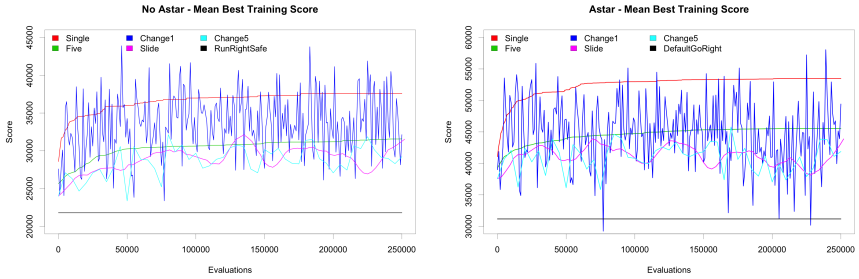


Fig. 8: Mean best training score across time, for all approaches not using (left) or using (right) A*. Note the difference in the scales on the Y axis.

The results confidently show that all approaches substantially outperform their respective reference behaviours. It seems evident that the BT approach can successfully add reactive elements to the controllers, enhancing its performance. A second observation that can be made is that there is a performance difference between the controllers that use A* navigation and those that do not. The *RunRightSafe* controller has an average performance close to 22,000 points, while *DefaultGoRight* averages above 31,000. With the exception of the *Single* approach, this is at a par (or superior) to the average controller performance for the other approaches not using A*. This showcases the performance improvement of using a dedicated, deterministic algorithm for navigation.

The relative performance of each of the approaches is similar with or without A*. It is worthwhile highlighting that the *Single* approach has the best training performance. It is therefore quite successful at optimising the controller behaviour for the single mapset it is trained on, independently from the initial random seed. It shows the best evolvability, with a typical optimisation performance curve, achieving the best training score with or without A*. A similar result can be observed with the *Five* approach, providing a steady improvement in average performance across the five training mapsets. The final lower total score is due to a more diverse performance across the different mapsets.

[2] This and all results reported in this chapter are averaged over 30 independent runs. Videos of the best controllers of some runs are available online (`http://tinyurl.com/gebtMarioAI`).

Table 2: Least-Squares Analysis of Learning Rates

	Approach	Train				Test			
		Intercept	Slope	Std. E.	Res. E.	Intercept	Slope	Std. E.	Res. E.
No A*	Single	3.49E+4	**1.40E-2**	4.84E-4	784.5	2.12E+4	2.26E-3	2.50E-4	127.7
	Five	2.87E+4	1.36E-2	1.00E-3	739.3	2.24E+4	3.22E-3	6.29E-4	321
	Change1	3.24E+4	8.64E-3	3.43E-3	3946	2.25E+4	**1.18E-2**	1.25E-3	639.1
	Slide	2.75E+4	1.12E-2	2.03E-3	1383	2.29E+4	1.15E-2	1.34E-3	684.2
	Change5	2.70E+4	1.25E-2	3.35E-3	1763	2.27E+4	1.17E-2	1.30E-3	667.1
	RunRightSafe	2.18E+4	0.0	0.0	0.0	2.18E+4	0.0	0.0	0.0
A*	Single	5.03E+4	**1.69E-2**	7.63E-4	1236	3.50E+4	-3.69E-3	4.53E-4	231.4
	Five	4.28E+4	1.41E-2	9.29E-4	680.9	3.63E+4	4.45E-4	3.38E-4	172.6
	Change1	4.55E+4	-5.81E-3	4.47E-3	5133	3.69E+4	1.04E-3	9.59E-4	489.6
	Slide	4.16E+4	2.94E-3	2.16E-3	1467	4.06E+4	**1.61E-2**	1.76E-3	902.7
	Change5	4.06E+4	4.34E-3	4.50E-3	2369	3.80E+4	8.16E-3	7.74E-4	395.2
	DefaultGoRight	3.11E+4	0.0	0.0	0.0	3.11E+4	0.0	0.0	0.0

Changing the mapset used for evaluation at each generation makes *Change1* approach the noisiest in terms of evolution across time. This noisy result shows the difficulty range of the levels generated in this framework with different random seeds, even when using 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 behaviour. *Change5* and *Slide* exhibit a similar performance. The evolution curves show the extreme range of difficulties due to the generated maps, but to a lesser extent than *Change1*, especially in the case of *Slide*, thanks to using several mapsets and modifying them between generations.

Table 2 shows the results of performing a linear regression to analyse the average learning rate of the different approaches. Although they are not linear, a simple linear model allows to make 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.

As can be seen, the *Single* approach exhibits the best average learning rate across all runs. *Five* also shows a good learning rate, while *Slide* and *Change5* exhibit similar learning rates, but lower and with a higher noise. Finally, *Change1* has the lowest learning rate, which is actually negative when used in conjunction with A* navigation, and the highest residual error (an indication of the range of different maps explored and how hard it is to evolve controllers in such a dynamic environment).

8.2 Performance on Test

A generalisation test was carried out in order to measure the performance of the evolved controllers in unseen scenarios. This test consisted of 20 unseen maps (seeds 666 to 685), with the same parameters as the training mapsets. The indi-

vidual that obtained the highest training fitness was tested every 5,000 evaluations and the average results across all runs are shown in Fig. 9.

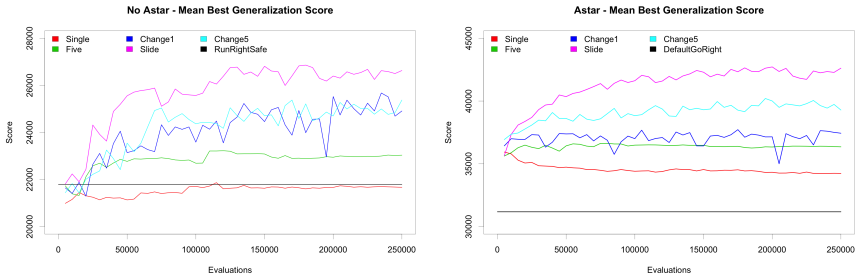


Fig. 9: Mean best test score of the best training individual every 5,000 evaluations, for all approaches not using (left) or using (right) A*.

The first result to notice in these tests is that the performance of all approaches decreases with respects to that of training. Given that these levels are unseen and not used for computing the fitness during training, this is to be expected. A* approaches fall from a $35,000 - 55,000$ training performance range to $34,000 - 44,000$ in test, while no A* behaviours drop from $25,000 - 40,000$ to $21,000 - 27,000$.

As can be observed, the *Single* approach clearly overfitted its single training mapset, and has the lowest generalisation score overall. It is interesting to see that, with no A*, it is even worse than the reference *RunRightSafe* behaviour, while with A* its average generalisation score worsens as evolution progresses. Despite of a few signs of training overfitting, the *Five* approach slightly improves its generalisation score over time without A*. With A* it quickly reaches its best performance without further improvement overtime. Its performance is again substantially better with A* (over $36,000$ points) than without (around $23,000$ points).

Change1 and *Change5* steadily improve their generalisation performance, when used without A*. In spite of the very noisy average training performance, the large number of generations it is allowed to evolve can be a reason for this improvement. In the case if A*, both approaches show a better performance than their no A* counterparts, with *Change5* showing a better improvement than *Change1*. However, the approach that shows the best performance in the generalisation test is *Slide*, with a substantially better average score at all evaluation steps.

Table 3 shows the test performance of the best training controllers (again averaged across 30 runs). All A* approaches present significantly better test performance than their respective no A* behaviours, with mostly non-overlapping standard deviation intervals.

The right half of Table 2 presents the test score improvement rates of all approaches; these are very low for *Single* and *Five*. *Single* has a negative learning rate, when used with A* navigation. *Slide* and *Change5* show good test performance im-

Table 3: Average Test Performance and Std. Deviation

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

provement over time, *Slide* with A* exhibiting the highest learning rate across all sets (in training and testing) and approaches (A* or not). Finally, it is worth highlighting that the learning rate of *Change1* (with A*), albeit very low, is positive, in contrast to its negative training learning rate.

8.3 Analysis

8.3.1 Fitness Analysis

Fig. 10 provides an analysis of the specific fitness contributors (number of cells passed, number of kills and time left when Mario dies or finishes a level) of the best evolved controllers, averaged across the 20 test mapsets.

This analysis reinforces the idea that A* navigation makes an evident contribution to the survivability of Mario. 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 areas that are difficult to navigate.

Controllers evolved with no A* show a worse test performance in terms of total cells passed than their reference behaviour (*RunRightSafe*). This is because BT structures need to be evolved to effectively combine navigation and reactiveness actions (such as killing enemies). *Single* and *Five*, despite of being poor at generalisation, are able to improve their average number of kills. They fail, however, at significantly evolving controllers that improve the number of cells passed or the total time left.

In the A* case, the BT structures are evolved mainly for reactiveness, with all approaches producing good reactiveness behaviour blocks. This allows them to improve the good navigation base given by the *DefaultGoRight* increasing the number of cells passed, as well as the number of kills (with a much higher improvement than controllers without A*).

This analysis provides another indication that the *Single* approach does not generalise well. It exhibits little or no improvement (apart from number of kills, when used without A*). *Five* performs slightly better, and all the other approaches evolve better generalisable behaviours across time.

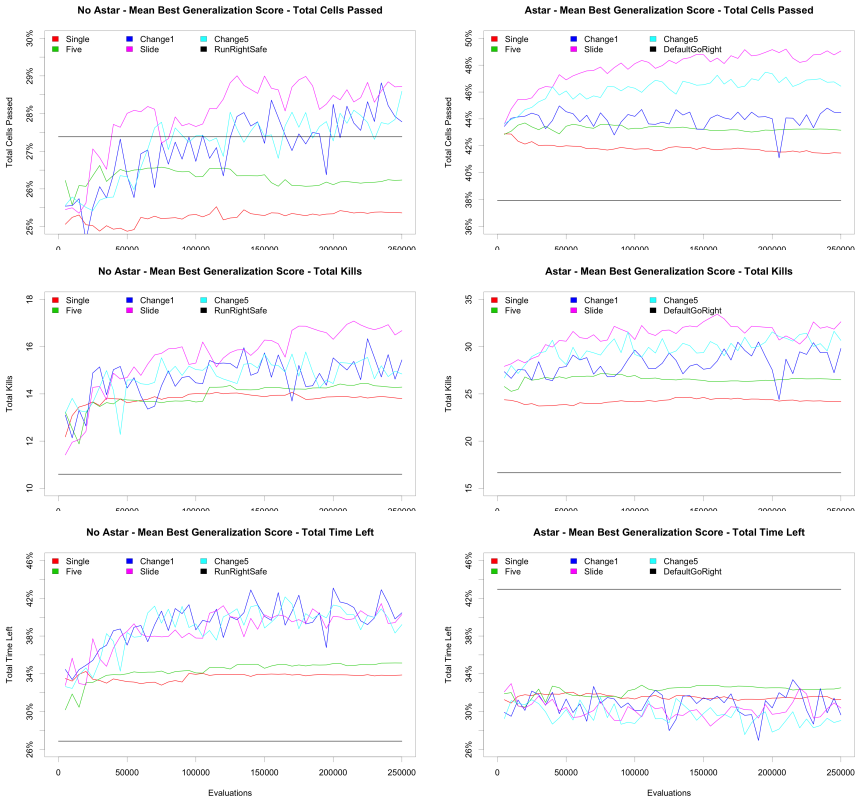


Fig. 10: Breakdown of average test performance of best evolved controllers, without A* (left) and with A* (right). The first row plots average percentage of passed cells; the second the average number of kills; and the last the percentage of time left.

It is also of interest the (non-)evolution of how much time was left after the game was completed when using A*. Even though better navigation means maps will be finished in less time, complex reactive behaviours and overall better survivability increase the amount of time spent in each map. Fitness-contributing behaviours, such as collection of items, were shown hard to evolve. Controllers with A* collected, on average, between 1 and 2 items, while no A* controllers achieved less than 1.

8.3.2 Structure and Size of the Solutions

Fig. 11 (top) shows the average (genotypic) solution size, and it provides further evidence that the *Single* and *Five* approaches overfit their target maps. The size of the genotype steadily increases through evolution, which is more pronounced when

A* is not used (hence solution size seems more stable with a pathfinding algorithm in the behaviour).

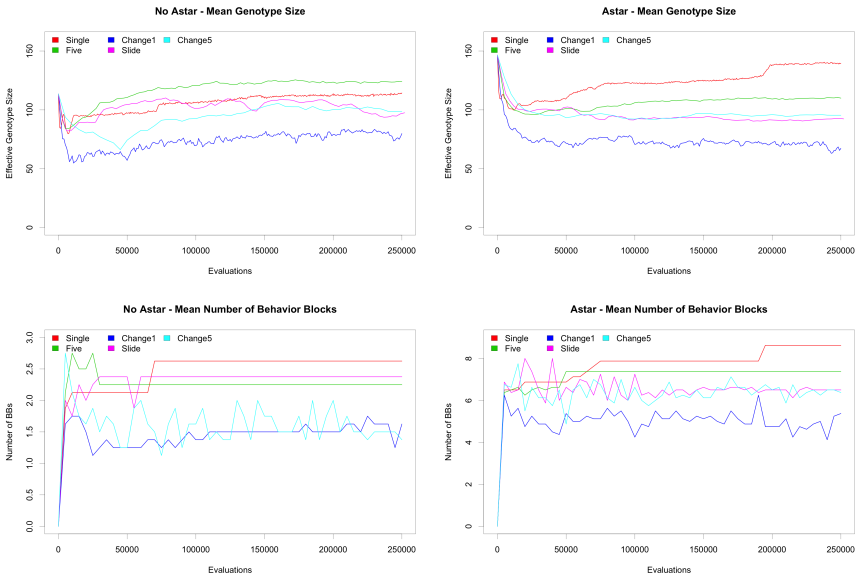


Fig. 11: Top row: Genotype solution size without (left) and with (right) A*, for the best individuals. Bottom row: Mean number of BBs for the best individuals, without (left) and with (right) A*.

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. 11 (bottom) plots the number of Behaviour Blocks (BBs) in the best controllers as seen across time. As can be seen, this number is very stable, indicating that evolution is mainly focused on opimising the contents of each BB keeping a similar number in the BT.

Having a small number of BBs without A* corresponds with a very complex structure on each one of them. It is worth highlighting that this very small number limits the effectiveness of the crossover operator: each BB incorporates a complex mix of both navigation and reactiveness actions, which is difficulty to combine and inter-exchange properly. When using A* as a default behaviour for navigation, each BB is mostly a compact set of conditions and actions evolved for reactiveness, resulting in a larger number of BBs being evolved by each controller. These BBs are more easily exchanged through crossover as independent reactive sequences.

9 Conclusions and Future Work

This chapter analysed the application of GE to evolve Behaviour Trees, a structure that is able to incorporate reactive (short term) and planning (long term) concerns in highly dynamic environments. In particular, this chapter showed the application of this technique to evolve controllers for the Mario AI video-game.

The experiments showed that GE is suitable for this task, allowing the incorporation of domain knowledge and specific syntax restrictions into the generated solutions, through careful grammar design. Also, the specification of crossover points allowed for the definition and exchange of Behaviour Blocks, accelerating the evolutionary process. The resulting solutions are human readable, and easy to analyse and fine-tune. Not only this shows the applicability of this technique to produce usable and maintainable behaviours, but it also addresses one of the main concerns of the games industry regarding speed, applicability and understandability of solutions proposed by evolutionary approaches.

Possible extensions to this work include adopting a multi-objective approach, dividing the objective fitness score into some of its constituents (such as distance traveled or number of kills). Specifically related to the use of BTs, one could also record statistics such as the frequency of use, the number of kills and the complexity of each BB, and use this information to prune or inform the crossover operators. Some mechanisms would have to be in place to avoid early convergence to a very reduced set of BBs, but this could also be achieved through individual BB analysis.

Appendix: Actions, Conditions, Filters and Sub-Trees for Mario

This section contains tables with the actions, conditions, filters and sub-trees used by the approaches described in this chapter to evolve Behaviour Trees. 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 analyse the contents of each cell. Also, note that actions use the terms *left* and *right*, which imply movement, while conditions use *ahead* (for right) and *back* (for left). Entries marked with a † denote sub-trees that have an analogous *left* (or *back*) variant.

Filters			
Name	¬A*	A*	**Description**
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 a *failure*.
UntilFailsLimited	✓	✓	Repeats the execution of its sub-tree *N* times or until *failure*.

Table 4: Filters available to the GE to evolve a BT.

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/Question	✓	✓	Verifies if there is a brick/question block over Mario.
EnemyAhead †	✓	✓	Checks if there is an enemy ahead of Mario.
EnemyAheadUp/Down †	✓	✓	Queries for an enemy ahead and over/below Mario.
JumpableEnemyAhead †	✓	✓	Checks ahead for an enemy that can be stomped.
NoJumpableEnemyAhead †	✓	✓	Checks ahead for an enemy that cannot be stomped.
IsBulletToHead/Feet	✓	✓	Checks for a bullet coming towards Mario's head/feet.
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 gap in front of Mario.
IsBreakableUp/Ahead	✓		Checks for a breakable block above/ahead of Mario.
IsClimbableUp/Ahead	✓		Checks for a climbable platform over/ahead of Mario.
IsJumpPlatformUpAhead †	✓		Verifies if there is a platform ahead and over Mario.
IsPushableUp/Ahead	✓		Checks for a question mark block over/ahead of Mario.
ObstacleAhead †	✓		Verifies if there is an obstacle ahead of Mario.
ObstacleHead †	✓		Indicates if there is an obstacle ahead, at Mario's head.
Actions			
NOP, Down, Fire	✓	✓	No action and atomic actions for *Down* and *Fire*, resp.
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 get a path to the closest question mark block.
GetPathToClosestItem		✓	Uses A* to get a path to the closest item to Mario.
GetPathToGround		✓	Gets a path (with A*) to lowest position seen in the level.
GetPathToTop		✓	Gets a path to highest position seen in the level.
GetPathToClosestRightMost		✓	Gets a path to rightmost position seen in the level.
GetPathToClosestLeftMost		✓	Gets a path to leftmost position seen in the level.
Sub-Trees			
UseRightGap †	✓		Moves Mario to the right until no blocks are over him to then jump to a higher platform and continue from there.
AvoidRightTrap †	✓		Attempts to overcome a dead end. Takes Mario back to the previous bifurcation point to then use *UseLeftGap*.
GoUnderRight †	✓		Passes structures traversable only if Mario is small; or he runs, crouches and slides under it.
DefaultPathPlanner		✓	Gets the path to the rightmost position on the screen.
PathFollower		✓	Follows the last path calculated.
JumpRightLong †	✓	✓	Shown in Fig. 4. Makes a long jump to the right. The filter executes the *JumpRight* action 9 frames.
JumpRightRunLong †	✓	✓	As above, with JumpRightRun (*jump*, *right*, *run*).
JumpRightShort †	✓	✓	As *JumpRightLong*, with *JumpRight* executed 3 frames.
JumpRightRunShort †	✓	✓	As the one above, with JumpRightRun.
WalkRightSafe †	✓	✓	Moves Mario to the right, checking for hazards, trying to jump (or kill, if enemy can be stomped) over them.
RunRightSafe †	✓	✓	As *WalkRightSafe*, but the input *run* is always on.
VerticalJumpLong	✓	✓	As *JumpRightLong*, but the action is Jump (input *jump*).
VerticalJumpShort	✓	✓	As *JumpRightShort*, but the action is Jump.

Table 5: Conditions, Actions and Sub-Trees available to the GE to evolve a BT.

References

1. C. Ryan, J. J. Collins, and M. O'Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language," in *EuroGP 98*, 1998, pp. 83–96. [Online]. Available: https://doi.org/10.1007/BFb0055930
2. J. Togelius, S. Karakovskiy, J. Koutnik, and J. Schmidhuber, "Super Mario Evolution," in *IEEE Symposium on Computational Intelligence and Games*, 2009, pp. 156–161.
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. M. Nicolau, D. Perez-Liebana, M. O'Neill, and A. Brabazon, "Evolutionary behavior tree approaches for navigating platform games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 3, pp. 227–238, September 2017.
5. R. Colvin and I. Hayes, "A Semantics for Behavior Trees," ARC Centre for Complex Systems (ACCS), Technical Report ACCS-TR-07-01, 2007.
6. M. O'Neill and C. Ryan, "Evolving multi-line compilable c programs," in *Genetic Programming, 2nd European Workshop, EuroGP 99, Göteborg, Sweden, May 26-27, 1999, Proceedings*, ser. Lecture Notes in Computer Science, R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, Eds., vol. 1598. Springer Berlin Heidelberg, 1999, pp. 83–92. [Online]. Available: https://doi.org/10.1007/3-540-48885-5_7
7. J. M. Swafford, M. O'Neill, M. Nicolau, and A. Brabazon, "Exploring grammatical modification with modules in grammatical evolution," in *European Conference on Genetic Programming, EuroGP 2011, Torino, Italy, April 27-29, 2011, Proceedings*, ser. Lecture Notes in Computer Science, S. Silva, J. E. Foster, M. Nicolau, P. Machado, and M. Giacobini, Eds., vol. 6621. Springer, 2011, pp. 310–321.
8. E. Murphy, M. Nicolau, E. Hemberg, M. O'Neill, and A. Brabazon, "Differential gene expression with tree-adjunct grammars," in *Parallel Problem Solving from Nature - PPSN XII, 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings*, ser. Lecture Notes in Computer Science, C. A. C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, and M. Pavone, Eds., vol. 7491. Springer, 2012, pp. 377–386.
9. 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.
10. R. Harper, "Co-Evolving Robocode Tanks," in *GECCO, Genetic and Evolutionary Computation Conference*, N. K. et al., Ed. ACM, 2011, pp. 1443–1450.
11. ——, "Evolving Robocode tanks for Evo Robocode," *Genetic Programming and Evolvable Machines*, vol. 15, no. 4, pp. 403–431, 2014.
12. J. E. Murphy, M. O'Neill, and H. Carr, "Exploring Grammatical Evolution for Horse Gait Optimisation," in *EuroGP 2009*, ser. LNCS, Leonardo Vanneschi and Steven Gustafson, Ed., vol. 5481. Springer, 2009, pp. 183–194.
13. N. Shaker, M. Nicolau, G. Yannakakis, J. Togelius, and M. O'Neill, "Evolving levels for super mario bros using grammatical evolution," in *IEEE Conference on Computation Intelligence and Games, CIG 2012, Granada, Spain, September 11 - 14, 2012, Proceedings*, 2012, pp. 304–311.
14. N. Shaker, G. Y. J. Togelius, , M. Nicolau, and M. O'Neill, "Evolving personalised content for super mario bros using grammatical evolution," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-12, 8th Conference, Stanford, USA, October 8-12, 2012, Proceedings*. AAAI, 2012, pp. 75–80.
15. 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.
16. 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.

17. 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.
18. 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.
19. 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.
20. H. Handa, "Dimensionality Reduction of Scene and Enemy Information in Mario," in *IEEE Congress on Evolutionary Computation*, 2011, pp. 1515–1520.
21. E. J. Jacobsen, R. Greve, and J. Togelius, "Monte Mario: Platforming with MCTS," in *GECCO, Genetic and Evolutionary Computation Conference*, 2014, pp. 293–300.
22. J. Liu, J. Togelius, D. Perez-Liebana, and S. M. Lucas, "Evolving game skill-depth using general video game ai agents," in *IEEE Conference on Evolutionary Computation*, 2017.
23. K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas, "The N-tuple Bandit Evolutionary Algorithm for Automatic Game Improvement," in *IEEE Conference on Evolutionary Computation*, 2017.
24. A. Champandard, M. Dawe, and D. H. Cerpa, "Behavior Trees: Three Ways of Cultivating Strong AI," Game Developers Conference, Audio Lecture, 2010.
25. D. Isla, "Managing Complexity in the Halo 2 AI System," in *Game Developers Conference*, 2005.
26. L. McHugh, "Three Approaches to Behavior Tree AI," in *Game Developers Conference*, 2007.
27. M. Mateas and A. Stern, "Managing Intermixing Behavior Hierarchies," in *Game Developers Conference*, 2004.
28. C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving Behaviour Trees for the Commercial Game DEFCON," in *EvoApplications 2010*, vol. 6024. Springer, 2010, pp. 100–110.
29. J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*, 1st ed. A Bradford Book, 1992.
30. A. Klöckner, "Behavior Trees for UAV Mission Management," *INFORMATIK 2013: Informatik angepasst an Mensch, Organisation und Umwelt*, pp. 57–68, 2013.
31. M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees," *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, 2017.
32. I. Millington and J. Funge, *Artificial Intelligence for Games*. CRC Press, 2016.
33. M. Nicolau, "Automatic Grammar Complexity Reduction in Grammatical Evolution," in *GECCO, Genetic and Evolutionary Computation Conference Workshops*, 2004.
34. R. Harper, "GE, Explosive Grammars and the Lasting Legacy of Bad Initialisation," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 2602–2609.
35. N. J. Nilsson, *Artificial Intelligence, A New Synthesis*. Morgan Kaufmann Publishers, 1998.
36. A. Champandard, "Behavior Trees for Next-Gen Game AI," Game Developers Conference, Audio Lecture, 2007.
37. M. Nicolau and I. Dempsey, "Introducing Grammar Based Extensions for Grammatical Evolution," in *IEEE Congress on Evolutionary Computation*, 2006, pp. 2663–2670.
38. M. Nicolau and D. Costelloe, "Using Grammatical Evolution to Parameterise Interactive 3D Image Generation," in *EvoApplications 2011*, ser. LNCS, vol. 6625. Springer, 2011, pp. 374–383.