Reactiveness and Navigation in Computer Games: Different Needs, Different Approaches

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

***Abstract*—This paper presents an approach to the Mario AI Benchmark problem, using the A* algorithm for navigation, and an evolutionary process combining routines for the reactiveness of the resulting bot. The Grammatical Evolution system was used to evolve Behaviour Trees, combining both types of routines, while the highly dynamic nature of the environment required specific approaches to deal with over-fitting issues. The results obtained highlight the need for specific algorithms for the different aspects of controlling a bot in a game environment, while Behaviour Trees provided the perfect representation to combine all those algorithms.**

I. Introduction

Computer games can be an extremely challenging benchmark for Evolutionary Algorithms, and for Artificial Intelligence in general. The challenges presented go from static or dynamic path planning and single move optimisation, to adaptation in dynamic environments, learning and cooperative behaviours. Extra challenges include the need for human-like behaviours, avoidance of repetitiveness, and conformity to the ability of human-opponents.

Evolutionary algorithms can help to solve some of these problems, making them particularly suitable for certain game environments. Their stochastic nature, along with tunable high- or low-level representations, contribute to the discovery of non-obvious solutions, while their population-based nature can contribute to adaptability, especially in dynamic environments. There are also drawbacks, however: traditionally, the games industry tends to adopt traditional, *classic AI* algorithms, such as *A**, *min-max* and others, due to their field-tested reliability, versus the stochastic, variable nature of evolutionary algorithms.

The main objective of this paper is to investigate the applicability of Genetic Programming [13] (GP) systems to evolve Behaviour Trees [9] (BTs), in order to improve the reactiveness of agents in dynamic game environments.

Behaviour Trees (BTs) were introduced as a way to encode formal system specifications [9]. Recently, they have also been used to encode game AI in a modular, scalable and reusable manner [7]. They have been used in high-revenue commercial games, such as *Halo* [11] and *Spore* [17], smaller indie games, such as *Façade* [19], and many other unpublished uses [7], which illustrate their flexibility and growing importance in the commercial game AI world.

Diego Perez is an independent author, email: Diego.Perez.Liebana@gmail.com; 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, Michael.ONeill@ucd.ie, Anthony.Brabazon@ucd.ie

The Mario AI Benchmark [31] was used, as it provides this kind of environment, with a series of obstacles to bypass, all the while avoiding (or eliminating) enemies and collecting bonuses as coins or items. The reactive nature of BTs can be used as a powerful representation for dynamic environments, and the flexibility provided by Grammatical Evolution [23] (GE) facilitates the evolution of behaviour tree structures, and their evaluation in a live play scenario.

One of the major challenges faced in this environment is the combination of navigation and reactiveness, in order to achieve the ultimate goal (reach the end of each level). On one hand, Mario has to deal with enemies and other dynamic objects that can hurt him or power him up, so there is the need of instant, reactive behaviour. On the other hand, Mario has to advance through the level in order to reach its end, dealing with structural hazards (jumps, traps, etc.), that require some kind of path planning. The approach described here uses an A* implementation to dynamically devise a path through each level, while BTs, driven by evolution, are responsible for the reactive behaviour of the agent.

This paper starts by giving some literature background. The Mario AI Benchmark environment is then described, followed by an overview of the path finding algorithm used for navigation. An introduction to Behaviour Trees follows, detailing their specific application to the problem; this is followed by a section detailing the application of the Grammatical Evolution system. Finally, the experimental setup and results obtained are discussed.

II. Relevant Literature

The literature provides us with some examples of using evolutionary computation algorithms to control AI agents in game environments. In terms of anticipating and reactive behaviour, examples include the work of Nason and Laird [22], who proposed an approach to add anticipation to bots in the *Quake* game, using reinforcement learning; and that of Thurau et al. [29], who produced agents that try to learn desirable behaviour based on imitation of already existing players, by training a neural network on data gained from human players. Mora et al. [18] used a Genetic Algorithm to fine-tune parameters of an existing AI bot code, and Genetic Programming to change the default set of rules or states that define a behaviour; and finally, Priesterjahn[26] used Evolution Strategies to evolve bot players in the *Quake III* game, by using an evolutionary process to create and select input/output rules, with inputs being a grid representation of the world around the bot, along with an associated action.

978-1-4577-0011-8/11/$26.00 ©2011 IEEE

The work of Lim et al. [14] specifically dealt with evolving behaviour tree structures. It used GP to evolve AI controllers for the *DEFCON* game; the final evolved tree was pitted against the standard DEFCON AI controller, and achieved a success rate superior to 50%. 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 easily dealt with by using grammar-based GP systems. An earlier version of the work described in the current paper highlighted that [24]: the Grammatical Evolution system was applied to the Mario AI Benchmark, evolving Behaviour Tree controllers.

The literature is also broad in terms of using path planning for navigation, both in robotics [16] and in games, such as *Unreal Tournament*, *Quake III* or *Half Life* [4]. The most common algorithm used for path finding is A*, because of its great performance, accuracy and efficiency [3].

In the Computational Intelligence and Games (CIG) 2009 conference, the three top entries for the Mario AI Championship, by Robin Baumgarten, Peter Lawford and Andy Sloane [30], used an A* algorithm to manage local navigation. Additionally, the CIG 2010 Mario AI Championship was won by Slawomir Bojarski [5], who also implemented an A* to determine keystrokes for high level actions.

Furthermore, path planning algorithms are usually combined with other reactive behaviours or systems, such as general planning, obstacle avoidance, speech or 3D animation [6].

III. The Mario AI Benchmark

The Mario AI Benchmark was used for the experiments described in this paper. This benchmark is an open source software, developed by Togelius et al. [31], and it was used in the 2010 Mario AI competitions. It allows the creation of agents that play the game, by providing two methods: one to retrieve and process environment information, and the other to specify the actions of the bot.

A. Environment information

All the information that can be used to analyse the world around Mario is given in two matrices (21x21). Each of these provides data about the geometry of the level, and the enemies that populate it. These arrays are centred around Mario, so 10 grid cells in each direction from the position of Mario can be processed every cycle (see Fig. 1).

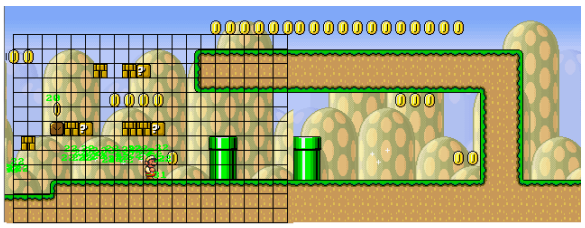

Fig. 1. Matrix centred in Mario.

Additionally, three different levels of detail can be specified to retrieve data in both arrays, depending on the information we are looking for. **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 absence of it. As a mid term, **Zoom 1** categorizes the information in useful groups, such as enemies that can be killed by stomping, those that can be killed by shooting, different types of blocks, etc.

Apart from this information, more useful input represents the current state of the game. The Mario **position** is a pair of values that indicates the coordinates in pixels of Mario in the level. The Mario **status** details the state of the game: running, win or dead. During the game, Mario can be small or big, with or without being able to fire, which is represented by Mario's **mode**. There are also some Mario **state indicators**, that provide data such as the ability of Mario to shoot and jump, the time left for the level and whether Mario is on the ground or not. Finally, some Mario **kill statistics** are also available, such as the number of enemies killed and how they died (by stomp, by fire or by shell bashing).

B. Mario effectors

The actions that can be performed by Mario are all the different inputs that a human player could use with a control pad. They are represented as a boolean array, where each control has a concrete index assigned.

The controls that may be used are the directions (*Left*, *Right*, *Up* and *Down*), a button for jumping and a command to both fire and speed up. Going faster only works if Mario is moving to the right or left. Jumps while pressing this button also make Mario reach farther platforms.

IV. Deliberative vs. Reactive: A* for navigation

The implementation proposed in this paper controls how Mario moves through the level in two different points of view: reactive and deliberative (or path planning).

A. Reactive behaviours

Reactive behaviours are concerned with all the elements in the game that can change their position. In this case, entities that move are enemies, such as evil mushrooms, bullets or flying turtles, which can follow different types of trajectories. Also included in this category are items that can appear in the level after pushing brick blocks, like bonus mushrooms (which make Mario big) and fire flowers (which give Mario the ability to shoot).

In this agent, the Grammatical Evolution algorithm is in charge of managing the reaction to these entities, through the evolution of Behaviour Trees; these systems and their combination are detailed in Sections V and VI.

B. Path planning with A*

In order to use A* for navigation, a graphical representation of the level is required. The Mario Benchmark does not provide any graph, or even a map, of the level that is being played. Furthermore, the level geometry can change during

the game: Mario can break blocks and new paths are created. For this reason, the creation process of the map (and, hence, the graph) is executed frequently. In this section, this map and graph creation process is explained in detail.

a) Building the map of the level: The first problem that must be faced when dealing with path planning is the world representation. In the case of the Mario Benchmark, the agent has no access to the map of the level, so it has to build one as Mario is moving through it. The only way to examine the environment is to read the information available in the level matrices.

However, this information is relative to Mario's position (from now on, *local coordinates*), while the map of the level should be built in absolute values (or *world coordinates*). Furthermore, Mario's position is provided in pixels with the origin at the upper left corner of the beginning of the level, whereas the local information matrix gives the information in grid cells units. For instance, Mario could be at (225, 53) while a brick block has been detected in (+1, -2).

Knowing that every cell block is equivalent to a 16x16 pixels square, we can easily translate the coordinates from local to world system. In the previous example, Mario would be in the coordinates $(225/16, 53/16) = (14, 3)$, and the brick block in $(14 + 1, 3 - 2) = (15, 1)$.

Using this conversion system, we can determine the position of every unit in the level in a discrete grid. Fig. 2 shows a portion of a level, and Fig. 3 shows the translation of this portion to a discrete coordinate system.

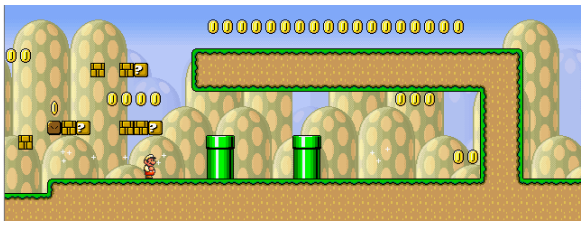

Fig. 2. Segment of a level in the Mario Benchmark's view.

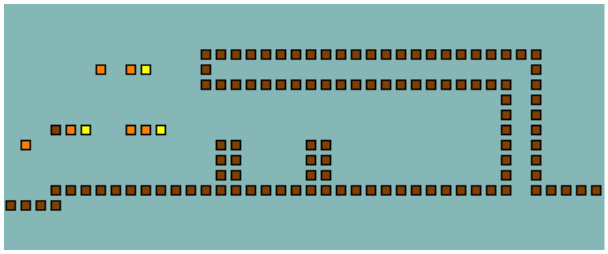

Fig. 3. Representation of the level map.

It is important to note that this map is being created for navigation, so blocks that do not affect movement (items, enemies or coins) are not taken into account.

b) Identifying nodes for the graph: 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 to make is where to place the nodes (or vertices) of the graph within the map; those will be those points in the map where Mario can stand. In Fig. 4, the nodes of the graph are depicted as dots.

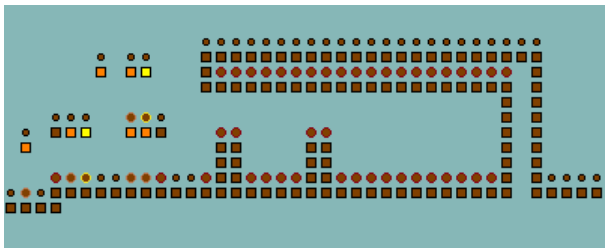

Fig. 4. Representation of the level map and graph nodes.

Two important observations can be made. First, some nodes are created for the graph that are not accessible; that is the case of the nodes inside the ceiling of the dead end. While Mario cannot actually reach those locations, the data representation given by the benchmark allows him to stand there. However, this does not represent a problem, because the creation process of the edges will ignore the nodes that cannot be reached by the bot.

Secondly, useful information about what is over each node is stored in these vertices as meta-data: some of them can be seen in the map (depicted as different dots), such as question mark blocks, brick blocks or rough obstacles. But more information is stored, such as enemies, items and coins, as that information will be used to execute other (non-navigational) actions.

c) Creating edges for the graph: Once the nodes have been created, it is time to add the edges that link them to the graph. Most of the grid graph solutions in games are used considering the map as seen from a zenithal perspective, whereas in this case the map is built considering the view of the player. This fact introduces an important change: horizontal and vertical edges of the graph cannot be used in the same way. The edge creation process analyses the nodes (and their meta-data) in order to finish the graph construction, generating different kinds of links:

- **Walk links**: These edges join two nodes that are horizontally adjacent. For instance, a node in (X, Y) could be linked with the nodes $(X - 1, Y)$ and $(X + 1, Y)$ using this type of edge. These are the simplest ones, they can be used just by using the right (or left) actions, and they are bidirectional.
- **Jump links**: These are unidirectional edges that can be used to jump to a node that is over the starting node (with a maximum jump height, H) and at one unit to the left or right. An edge like this, starting at (X, Y), could link nodes from $(X-1, Y+M)$ to $(X+1, Y+M)$, where M goes from 1 to H.
- **Special jump link**: There are some level formations that can be jumped from the bottom up, keeping the same

vertical. For those edges, a special case of jump link is created where X_{origin} is the same as $X_{destination}$.

- **Fall link**: These edges are used to join those nodes that are next to each other in X, but at a lower Y. A subset of these links are the opposite ones to each *jump link*. In other words, those links that can be used to jump in one direction, can also be used to fall in the other way. It is important to make this distinction, 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. In coordinates, we can say that these edges start at (X, Y_a) and link nodes from $(X-1, Y_b)$ to $(X+1, Y_b)$, where $Y_b < Y_a$. If $M = Y_b - Y_a$, and M is lower than the maximum height jump, there will be an equivalent jump link edge in the opposite direction.
- **Faith jump link**: These edges are used to link nodes that are separated horizontally by more than one unit, and with a maximum vertical distance. Again in coordinates, a node from (X, Y) can be linked using this edge type with any other in the range from $(X-N, Y-M)$ to $(X+N, Y+M)$, where N goes from 2 to a maximum horizontal distance D and M from 1 to a maximum vertical distance H.
- **Break jump link**: These edges are very similar to the normal jump links, but in this case there is a brick block in the trajectory of the jump, concretely in the vertical of the node where the edge starts. If Mario jumps to hit this block, and it breaks, the path to the destination will be freed so he can jump again to arrive at the destination. However, there is a possibility that the block does not break (it can become a solid non-breakable block instead of disappearing) and the link cannot be used. Nevertheless, as stated before, as the map and graph are constantly generated, this link will not be created again.

Most of these link nodes are represented in Fig. 5, where the whole graph of this section of the map can be seen.

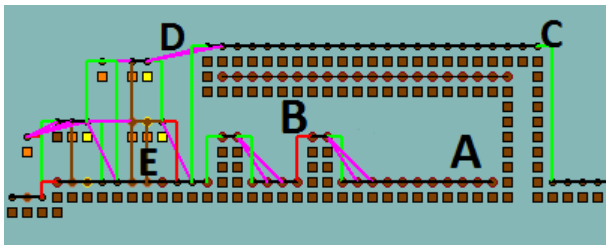

Fig. 5. Navigation graph representation. Different types of edges: A: Walk link. B: Jump link. C: Fall link. D: Faith jump link. E: Break jump link.

Apart from all these edge types, another modifier needs to be taken into account. As Mario can be small or big, that affects navigation. There are some edges that are only usable if Mario is small, as in this case he needs only one cell for his body to pass through. Making these distinctions for all the types of edges is crucial for the navigation, and not only for accessibility or otherwise of graph nodes: this information is also used to calculate the cost of each edge of the graph used by the A*.

The *basic cost* of an edge is calculated using the Manhattan distance between its nodes. However, the associated cost must be different depending on the link type: the cost of travelling an edge walking must be lower than jumping, because navigation takes more time to calculate the jump, it needs landing management and a higher risk (it is more likely to miss a jump than a simple walk movement). For this reason, the *basic cost* of each link that involves a jump is multiplied by 1.5, with the exception of the `break jump link`, that is multiplied by 3 because of its complexity.

C. Reactive and Path Planning together

An important question stills needs to be answered: how to combine both behaviours, reactive and deliberative, to enable Mario to follow paths and react to enemies at the same time. This is handled by the behaviour tree, by having a default behaviour that follows the last path established and a set of sub-trees with higher priority actions that manage the reactiveness of the bot. This organisation will be seen in detail in the section VI-B, where the default behaviour tree structure is described.

V. Behaviour Trees

A. Introduction

Behaviour Trees are an excellent data structure to organise behaviours in a hierarchical way, in order to establish a descending order of complexity; broad behavioural tasks are at the top of the tree, and are broken down into several sub-tasks. For example, a soldier in a first-person shooter game might have a behaviour AI that breaks down into patrol, investigate and attack tasks. Each of these can then be further broken down: attacking for example will require movement tactics, weapon management, and aiming algorithms. These can be further detailed, up to the level of playing sounds or animation sprites.

BT nodes can be divided into two major categories: *Control Nodes* and *Leaf Nodes*. Control nodes drive the execution flow through the tree, deciding which node is the next to be executed, and can be classified into **Sequence** nodes, **Selector** nodes, and **Filter** nodes (see Section V-B). Leaf nodes can be **Actions** or **Conditions**; the first make decisions and carry out specific tasks, while the second usually make queries about the game state.

B. Behaviour Trees for Mario

It is important to understand the engine mechanics when designing BTs for a specific game, as well as its data flow. In this case, at every step, a set of pressed buttons is required to move Mario. This affects how to run a given BT: control nodes and conditions will be executed until an action node is reached. For instance, an action to walk right safely will run a certain number of checks, until reaching a *Right* action; when the BT reaches this action, it finishes its execution

for this cycle, resuming from that point in the tree in the following step.

Another important decision regarded which nodes to provide for the BT. Regarding control nodes, the following were programmed: **sequence nodes** (which execute all their children from left to right until one fails, behaving like a logic AND): **selector nodes** (which execute their children until one succeeds, the equivalent of an OR); and **filters**, including `Loops`, which execute a node a specified amount of times, `Non`, which negates the result of a node, and `UntilFailsLimited`, which executes a node until failure, or an execution limit is reached.

The leaf nodes encoded can be grouped in three categories:

- **Conditions**. Using the environment information available (see III-A), these check the level for enemies and obstacles. For enemies, they consider if there are any close by, their location, and their type; for obstacles, they query the position of blocks near Mario. Examples include *EnemyAhead*, and *UnderQuestion*.
- **Actions**. These are the possible movements of Mario (see III-B). The actions programmed for the BT are the most interesting button combinations: actions like *Down*, *Fire*, *RunRight* (where *Right* and *Run* are both pressed), *NOP* (no buttons pressed) or *WalkLeft*. There is also a family of actions devoted to the task of getting a path to a concrete location: the actions *GetPathToRightMostPosition*, *GetPathToClosestQuestion* or *GetPathToClosestItem* are examples of members of this family. Some actions, however, require a button to be pressed more than once: for instance, to make long jumps, the longer the *Jump* button is pressed, the farther the jump will be. This problem can be solved with the elements of the next category.
- **Sub-trees**. These are manually designed BTs, to solve specific problems. For example, to make jumps, the jump button is required to start unset, followed by several cycles with the button pressed. This can be achieved by using different types of nodes, such as `Loop` filters, that in this case are used to repeat the press of the jump button for several cycles. Different subtrees were programmed for jumps, like *JumpRightLong*, *VerticalJumpLong* or *JumpRightRunLong*.

VI. Grammatical Evolution

As seen before, the control of Mario's behaviour throughout the game is controlled by a BT, which alternates between enemy management sections (reactive) and navigation sections (path planning). The actual structure of the BT was evolved with Grammatical Evolution [23] (GE), presented next.

GE is a grammar-based form of GP [15] that specifies the syntax of possible solutions through a context-free grammar, which is then used to map integer strings to syntactically correct solutions. Those integer strings can therefore be created by any search algorithm.

One of the key characteristics of GE is that the syntax of the resulting solutions is specified through a grammar. This facilitates its application to a variety of problems with relative ease, and explains its usage for the current application.

```
<BT>        ::= <BT> <Node> | <Node>
<Node>      ::= <Condition> | <Action>
<Condition> ::= if(obstacleAhead) <Action>;
              | if(enemyAhead) <Action>;
<Action>    ::= moveLeft; | moveRight;
              | jump; | shoot; | crouch;
```

Fig. 6. Illustrative grammar for an approach to a generic shooting game.

GE employs a genotype-to-phenotype mapping process: variable-length integer strings are evolved, typically with a Genetic Algorithm [10], and are then used to choose production rules from a grammar, which create a phenotypic program, syntactically correct for the problem domain. Finally, this program is evaluated, and its fitness returned to the evolutionary algorithm.

A. Example Mapping Process

To illustrate the mapping process, consider the grammar in Fig. 6. Using the integer string `(4, 5, 3, 6, 8, 5, 8, 1)`, the first value is used to choose one of the two productions of the start symbol `<BT>`, through the formula $4\%2 = 0$, i.e. the first production is chosen, so the mapping string becomes `<BT><Node>`.

The following integer is then used with the first unmapped symbol in the mapping string, so through the formula $5\%2 = 1$ the symbol `<BT>` is replaced by `<Node>`, and thus the mapping string becomes `<Node><Node>`.

Proceeding in this fashion, the mapping string then becomes `<Action><Node>` through the formula $3\%2 = 1$, and through $6\%5 = 1$ it becomes `moveRight; <Node>`. After all symbols are mapped, the final program becomes `moveRight; if(enemyAhead) then shoot;`, which could be executed in an endless loop.

Sometimes the integer string may not have enough values to fully map a syntactic valid program; several options are available, such as reusing the same integers (in a process called wrapping[23]), assigning the individual the worst possible fitness, or replacing it with a legal individual. In this study, an unmapped (hence invalid) offspring is replaced by his (valid) originating parent.

B. Using GE to evolve BTs

The BT (XML) syntax was specified in the grammar, and all conditions (18), actions (15), sub-trees (14) and filters (4) were available. In the first version of this work [24], it was shown that restricting the structure of BTs was necessary, in order to avoid the evolution of syntactically correct but semantically non-sensical BTs; this was achieved with a carefully designed grammar. The trees that can be evolved, although still of variable size, are contrived to follow an *and-or* tree structure [21], much like a binary decision diagram [1], which is a recommended [8] way of building behaviour trees for game AI. The following structure was decided upon:

- The root node consists of a selector (`rootSelector`), with a variable number of sub-trees (`BehaviourBlocks`);
- Each `BehaviourBlock` consists of a sequence of one or more conditions, followed by a sequence of actions (filtered or not);
- A last (unconditioned) `BehaviourBlock`, called `DefaultPathPlanner`, carries out the A* path planning behaviour.

Fig. 7 illustrates the syntax described; it works as follows. When the BT is executed, the `rootSelector` will choose one `BehaviourBlock` to execute, based on the conditions associated with each one, on a left-to-right priority order; if none of those conditions fires, then the `DefaultPathPlanner` is executed. As the high-level conditions available are quite complex, it made sense to limit the number of these associated with each `BehaviourBlock`; this is easily done through the grammar, and in our experiments, there were only one or two conditions associated with each block. The number of actions and sub-trees in the associated sequence was unlimited.

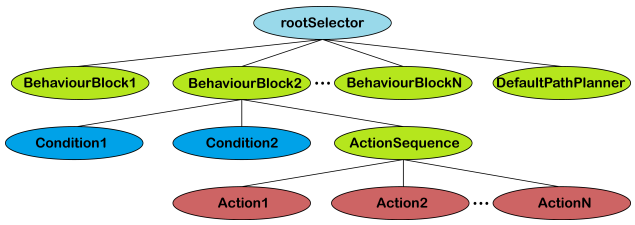

Fig. 7. Structure of evolved behaviour trees.

The `DefaultPathPlanner` is the last block in this sequence, and hence it has the lowest execution priority: it will only be executed if none of the previous blocks were. It is composed of two sub-trees, with a selector node as a parent of both. The first sub-tree is depicted in Fig. 8; it is in charge of calculating the default path to be followed.

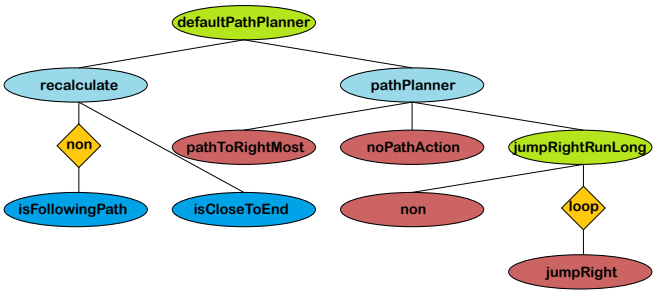

Fig. 8. First sub-tree for path planning. It calculates, if needed, the path to the rightmost position available.

Its root is a sequence node with two children. The first one (*Recalculate*) decides if the default path has to be calculated. That can happen when no item is being targeted, or when a path has been set but is about to be finished. Note that the former reason allows potential paths requested by the reactive part not to be overwritten by the default planner.

The second child (*Path Planner*), executed if the one before is successful, calculates the path to the right most position in the map (which is the direction to follow to the level end). In the unusual case of not finding a path, Mario enters an emergency situation: in order to keep moving, a default forward jump is executed.

Finally, the second sub-tree checks if a path has been set and, if that is the case, executes the action in charge of following the path.

C. Extensions to GE

With the syntax described above, each `BehaviourBlock` 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, bounding these blocks for exchange. This is a recent technique [20] in which a special grammar symbol is used to label crossover points; the search algorithm then only slices an individual according to these points. We extended this by using a two-point crossover, effectively creating an operator much like sub-tree crossover in GP [13], but allowing the exchange of different numbers of blocks between individuals. Without these markers, the standard 1-point crossover as used in standard GE would provide more exploration but less exploitation, and given the expensiveness of the fitness function, a trade-off seemed to make sense.

Finally, an individual is allowed to crossover with himself, thus creating a sub-tree swap operation; this makes sense, as a mean to increase (or decrease) the priority of a `BehaviourBlock`: the further to the left (right) within the `rootSelector`, the bigger (smaller) the likelihood of execution of a block.

D. Generalisation Issues

A difficulty with such a dynamic problem is that of generalisation performance. In the Mario AI competition [31], the off-line generated controllers are tested in a series of randomly-generated maps; the difficulty of these maps can be drastically different, ranging from surprisingly easy to virtually impossible (for the same difficulty level). Taking this into account, three approaches were tested to evolve BTs with GE:

- **Run5**: test each controller in five random sets of maps, which are never changed throughout the evolutionary cycle (and indeed are the same for all runs);
- **Change1**: only test one level, but change the random seed of that level at every generation (same set of seeds for each run), reevaluating the parent population with the new map only, thus allowing for more evolution cycle generations for the same amount of evaluations;
- **Run1**: always test in one "typical" level (same seed for all runs), assuming that the agent is presented with enough enemy and obstacle diversity to evolve good

reactiveness routines, allowing for the biggest number of evolution cycle generations.

VII. Experiments

A. Setup

The experimental parameters used are shown in Table I. All individuals in the initial generation were valid [27], and a variation of tournament selection was used, which ensures that each individual participates at least in one tournament event. Also, the mutation rate was set such that, on average, one mutation event occurs per individual (regardless of their size).

TABLE I
Experimental Setup

GE	Population Size	500
	Evaluations	125000
	Derivation-tree Depth Range (for initialisation)	20...30
	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 Lengths	320

Each evaluation is comprised of 10 levels (5 difficulty settings, with two types of map each). All three approaches (Run5, Change1 and Run1) were tested on 30 different runs, for statistical purposes. The individual fitness value, to be maximized, is a weighted sum of distance travelled and some other factors, like enemy kills and collected items.

B. Results

Fig. 9 shows the mean best individual fitness, for all three approaches, averaged across all runs. The graph shows that the Run1 approach was quite successful at optimising the controller behaviour, for the map that it was tested on. The Run5 approach also shows improvement over time, albeit at a lower rate. Finally, the Change1 approach seems quite noisy, which is a clear indication of the extreme range of difficulty of maps generated with different random seeds, even with the same difficulty setting.

A generalisation test was also devised, which consisted of 360 unseen levels (20 different random seeds, each generating 9 difficulty levels, with two types of map each). The best individual at every 5000 evaluations was tested, and the averaged results across all runs are shown in Fig. 10.

The results show interesting differences between the three approaches. The Run1 approach starts by improving its score, but very rapidly worsens it. This is a clear sign of over-fitting: the evolved controllers were tested in a single map, and a comparison with Fig. 9 shows that, despite an improving training score, the generalisation score keeps worsening.

The Run5 approach shows the most stable results. The generalisation score improves over time, albeit slowly; with a limited number of maps to train on, the evolved controllers

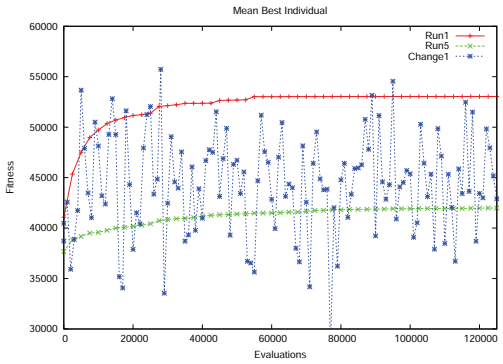

Fig. 9. Mean best individual score for all three approaches, averaged across 30 independent runs.

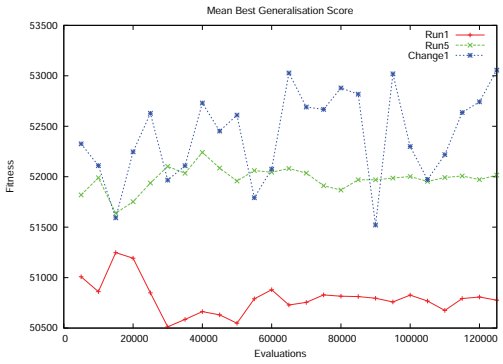

Fig. 10. Generalisation score of the best individual every 50000 evaluations, for all three approaches, averaged across 30 independent runs.

quickly converge to an overall *average* behaviour, as seen in Fig. 9, and with evolution stagnated (roughly after 80000 evaluations), there is little or no generalisation improvement.

Finally, the results of the Change1 approach are the most interesting. These highlight once again the variety of scores that can be achieved with differently seeded maps. Although the generalisation score is very irregular, there is an overall trend to improve it across time, and the scores achieved are the best of all three approaches.

VIII. Conclusions

This paper presented an extension of previous work [24], evolving Behaviour Trees for the Mario AI Benchmark. The A* algorithm was applied in a dynamic manner, constantly providing path planning routines that were combined with reactive routines in BTs; these trees were then evolved using Grammatical Evolution.

The combination of a Genetic Programming type algorithm with BTs provided a flexible approach. The resulting solutions are human readable, and thus easy to analyse and fine-tune, which aims at one of the main concerns of the

game industry regarding evolutionary approaches, as stated early in section I. Also, the use of a grammar allows full syntactic control of the resulting BTs, providing full control of their breadth, depth, and overall complexity. Finally, the combination of a carefully designed syntax with specific crossover locations allows the definition and exchange of meaningful behaviour blocks, accelerating the evolutionary process.

One of the potential drawbacks of using a stochastic algorithm is the need for an evolutionary process, making online learning difficult (and sometimes impossible). By applying the A* algorithm in a dynamic manner, however, the evolved behaviours remain adaptive, particularly in what concerns navigation.

The experiments and results presented are relevant in many aspects. They highlight the dynamic nature of game environments; with such a multitude of possible environments and situations to face (and the disparity of possible fitness scores), there is a real risk of over-fitting to specific game situations, and experimental design is of the utmost importance. Of all the approaches presented, the one with the best results reevaluated all individuals at each generation in a new set of unseen maps; this resulted in a very noisy fitness landscape, but in the end provided the best generalisation results.

Future work will aim to improve these results. There exists a wide body of research work dealing with the problem of over-fitting, and techniques such as using training and generalisation sets for early stopping of the evolutionary process [25], or sliding windows of training cases [32], should provide better generalisation scores. It will also be interesting to compare the performance of the GE algorithm with similar ones, as for instance a strongly typed tree-based Genetic Programming approach.

Finally, a controller combining all the techniques described in this paper will be submitted to the Mario AI Benchmark competition [12], allowing its comparison with other existing approaches.

ACKNOWLEDGMENTS

This research is based upon works supported by the Science Foundation Ireland under Grant No. 08/IN.1/I1868.

REFERENCES

[1] Binary Decision Diagrams. IEEE Transactions on Computers, C-27(6), pp. 509-516. (1987)
[2] P. Angeline: Subtree Crossover: Building Block Engine or Macromutation?. In: Genetic Programming 1997, Proceedings. Morgan Kaufmann (1997) pp. 9–17
[3] S. Bandi, D. Thalmann: Space discretization for efficient human navigation. In: Computer Graphics Forum, Vol. 17, Issue 3, September 1998 pp. 195–206
[4] K. Birdwell: The CABAL: Valve's Design Processing for Creating Half Life. In: Game Developer 6 (12) (1999) pp. 40–50
[5] Bojarski, S.; Congdon, C.B.; REALM: A rule-based evolutionary computation agent that learns to play Mario. In: 2010 IEEE Symposium on Computational Intelligence and Games (CIG) (2010). pp. 83–90
[6] Cavazza, M.; Bandi, S.; and Palmer, I.: Situated AI in Video Games: Integrating NLP, Path Planning, and 3D Animation. In: Papers from the AAAI 1999 Spring Symposium on Artificial Intelligence and Computer Games (1999) Technical Report SS-99-02. pp. 6–12
[7] A. Champandard, M. Dawe and D. H. Cerpa: Behavior Trees: Three Ways of Cultivating Strong AI. In: Game Developers Conference, Audio Lecture. (2010)
[8] A. Champandard: Behavior Trees for Next-Gen Game AI. In: Game Developers Conference, Audio Lecture. (2007)
[9] R. Colvin and I. J. Hayes: A Semantics for Behavior Trees. ARC Centre for Complex Systems, tech. report ACCS-TR-07-01. (2007)
[10] D. E. Goldberg: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley (1989)
[11] D. Isla: Managing Complexity in the Halo 2 AI System. In: Game Developers Conference, Proceedings. (2005)
[12] J. Togelius, S. Karakovskiy and R. Baumgarten: The 2009 Mario AI Competition. In: IEEE Congress on Evolutionary Computation, Proceedings. IEEE Press (2010) pp. 1–8
[13] J. R. Koza: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
[14] C. Lim, R. Baumgarten and S. Colton: Evolving Behaviour Trees for the Commercial Game DEFCON. In: Applications of Evolutionary Computation, EvoApplications 2010, Proceedings. (2010)
[15] R. I. McKay, X. H. Nguyen, P. A. Whigham, Y. Shan and M. O'Neill: Grammar-Based Genetic Programming - A Survey. Genetic Programming and Evolvable Machines, 11(3-4). (2010) pp. 365–396
[16] J. Meyer, D. Filliat: Map-based navigation in mobile robots: II. A review of map-learning and path-planning strategies. Cognitive Systems Research, 4(4). (2003) pp. 283–317
[17] L. McHugh: Three Approaches to Behavior Tree AI. In: Game Developers Conference, Proceedings. (2007)
[18] A. M. Mora, R. Montoya, J. J. Merelo, P. G. Sánchez, P. A. Castillo, J. L. J. Laredo, A. I. Martínez and A. Espacia: Evolving Bot AI in Unreal. In: Applications of Evolutionary Computation, EvoApplications 2010, Proceedings. Springer Verlag. (2010) pp. 171–180
[19] M. Mateas and A. Stern: Managing Intermixing Behavior Hierarchies. In: Game Developers Conference, Proceedings. (2004)
[20] M. Nicolau and I. Dempsey: Introducing Grammar Based Extensions for Grammatical Evolution. In: IEEE Congress on Evolutionary Computation, Proceedings. IEEE Press (2006) pp. 2663–2670
[21] N. J. Nilsson: Artificial Intelligence, A New Synthesis. Morgan Kaufmann Publishers. (1998)
[22] S. Nason and J. Laird: Soar-RL: Integrating Reinforcement Learning with Soar. In: International Conference on Cognitive Modelling, Proceedings. (2004)
[23] M. O'Neill and C. Ryan: Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language. Kluwer Academic Publishers (2003)
[24] D. Perez and M. Nicolau and M. O'Neill and A. Brabazon. Evolving Behavior Trees for the Mario AI Competition Using Grammatical Evolution. In Applications of Evolutionary Computation, EvoApplications 2011, Proceedings. Springer Verlag. (2011) pp. 121–130
[25] L. Prechelt. Early Stopping - But When?. In Neural Networks: Tricks of the Trade, Proceedings. Springer Verlag. (1997) pp. 55–69
[26] S. Priesterjahn: Imitation-Based Evolution of Artificial Game Players. ACM SIGEVOlution, 2(4). (2009) pp. 2–13
[27] C. Ryan and R. M. A. Azad: Sensible initialisation in grammatical evolution. In: Barry, A.M. (ed.) GECCO 2003: Proceedings of the Bird of a Feather Workshops. AAAI (2003) pp. 142–145.
[28] K. Sastry, U. O'Reilly, D. E. Goldberg and D. Hill: Building Block Supply in Genetic Programming. Genetic Programming Theory and Practice, Chapter 4. Kluwer Publishers (2003) pp. 137–154
[29] C. Thurau, C. Bauckhauge and G. Sagerer: Combining Self Organizing Maps and Multiplayer Perceptrons to Learn Bot-Behavior for a Comercial Game. In: GAME-ON'03 Conference, Proceedings. (2003)
[30] J. Togelius, S. Karakovskiy and R. Baumgarten: The 2009 Mario AI Competition. In: IEEE Congress on Evolutionary Computation, Proceedings. IEEE Press (2010)
[31] J. Togelius, S. Karakovskiy, J. Koutnik and J. Schmidhuber: Super Mario Evolution. In: IEEE Symposium on Computational Intelligence and Games, Proceedings. IEEE Press (2009)
[32] S. Winkler, M. Affenzeller, and S. Wagner: Selection Pressure Driven Sliding Window Behavior in Genetic Programming Based Structure Identification. In: EUROCAST'07 Conference, Proceedings. Springer-Verlag (2007) pp. 788–795