Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution

Diego Perez[1] Miguel Nicolau[2] Michael O'Neill[2]
Anthony Brabazon[2]
[1]Independent author
[2]Natural Computing Research & Applications Group
University College Dublin, Dublin, Ireland
diego.perez.liebana@gmail.com, Miguel.Nicolau@ucd.ie,
M.ONeill@ucd.ie, Anthony.Brabazon@ucd.ie

Abstract

This paper investigates the applicability of Genetic Programming type systems to dynamic game environments. Grammatical Evolution was used to evolve Behaviour Trees, in order to create controllers for the Mario AI Benchmark. The results obtained reinforce the applicability of evolutionary programming systems to the development of artificial intelligence in games, and in dynamic systems in general, illustrating their viability as an alternative to more standard AI techniques.

1 Introduction

Computer games are an extremely challenging testbed for Evolutionary Algorithms, and in fact for Artificial Intelligence in general. The challenges presented range from static path planning and one-on-one move optimisation, to adaptation in dynamic environments, 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 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, particularly in dynamic environments. There are also drawbacks, however, and traditionally, the games industry tends to adopt traditional, *hard AI* algorithms, such as *A**, *min-max*, and others.

The main objective of this paper is to investigate the applicability of Genetic Programming [Koz92] (GP) systems to evolve Behaviour Trees [CH07] (BTs), and their applicability to dynamic game environments. The Mario AI Benchmark was used, as it provides a challenging dynamic environment, with

a series of obstacles to bypass, all the while avoiding (or eliminating) enemies and collecting bonuses. The reactive nature of BTs can be seen as a powerful representation for this kind of environment, and the flexibility of Grammatical Evolution [OR03] (GE) facilitated their evolution, and subsequent evaluation in a live play scenario.

The best evolved bot was sent to the gameplay track of the 2010 Mario AI competition [TKB10], where the bots are required to navigate through unseen levels. The results obtained show the viability of the technique presented; pitted against fierce competition, it reached fourth place, very close to the top three.

This paper starts by giving some literature background, followed by an introduction to GE. It then details the environment of the Mario AI Benchmark, followed by a section introducing Behaviour Trees, and their specific application to the problem. Finally, the experimental setup and results are presented.

2 Relevant Literature

The literature provides us with some examples on the use of using evolutionary computation techniques for controlling AI agents in game environments. In terms of anticipating and reacting behaviour, examples include the work of Nason and Laird[NL04], who proposed an approach to add anticipation to bots in the "Quake" game, using reinforcement learning; and that of Thurau et al. [TBS04], who produced agents that try to learn desirable behaviour based on imitation of existing players, by training a neural network on data gained from human players. Pristerjahn[Pri09] 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; and finally, Mora et al. [MMM10] used a Genetic Algorithm to fine-tune parameters of existing AI bot code, and Genetic Programming to change the default set of rules or states that define a behaviour.

The work of Lim et al. [LBC10] is the only one that specifically deals with evolving behaviour tree structures. It used Genetic Programming [Koz92] (GP) to evolve AI controllers for the "DEFCON" game. It starts with a set of handcrafted trees, encoding feasible behaviours for each of the game's five parts; separate GP runs are then used for each part, creating new behaviours from the original set. The final combined tree, after evolution, was pitted against the standard AI controller that comes with the game, 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, such as Grammatical Evolution, presented next.

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

Figure 1: Illustrative grammar for simple approach to a generic shooting game.

3 Grammatical Evolution

Grammatical Evolution [OR03] (GE) is a grammar-based form of GP [MNW10] that specifies the syntax of possible solutions through a context-free grammar, which is then used to map binary strings to syntactically correct solutions. Those binary 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.

GE employs a genotype-to-phenotype mapping process: variable-length integer strings are evolved, typically with a Genetic Algorithm [Gol89], 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.

3.1 Example Mapping Process

To illustrate the mapping process, consider the grammar in Fig. 1. Using the integer string `(4, 5, 3, 6, 8, 5, 9, 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 $6\%5 = 2$ 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.

4 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. [TKK09], and was also used in the 2010 Mario AI Competition. 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.

4.1 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. Different detail levels can be specified in each array: for instance, `zoom level 2` gives the data represented in a binary array, stating the presence or absence of enemies (or obstacles), whereas `level 0` gives a very detailed view of the environment, with each kind of enemy or block in the game.

More information about the current state of the game is available, such as: the Mario **position**; its **status** (running, win or dead); its **mode** (small or big, affecting ability to fire); **state indicators** (such as the ability to jump and shoot, time left, etc); and finally, some Mario **kills** statistics are also available, like enemies killed and how they died.

4.2 Mario effectors

The actions that can be performed by Mario are the inputs that a human player could use with a control pad: the four movement directions (*Left, Right, Up, Down*), the *Jump* control, and a common button to *Fire* and *Jump*. This last effector can be also used to make Mario go faster; jumps while pressing this button also make Mario reach farther platforms.

5 Behaviour Trees

5.1 Introduction

Behaviour Trees (BTs) were introduced as a means to encode formal system specifications [CH07]. Recently, they have also been used to encode game AI in a modular, scalable and reusable manner [CDC10]. They have been used in high-revenue commercial games, such as "Halo" [Isl05] and "Spore" [Mch07], smaller indie games, such as "Façade" [MS04], and many other unpublished uses [CDC10], illustrating their growing importance in the game AI world.

BTs provide a hierarchical way of organising behaviours in 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 moving 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*. The first drive the execution flow through the tree, deciding which node

to execute next; for instance, **Sequence** nodes execute all their children from left to right until one fails (behaving like a logic AND), while **Selector** nodes execute their children until one succeeds (the equivalent of an OR). **Filter** nodes can be also added to this group; they are decorators that modify the execution flow in different ways (like loops, negating the result of a node, etc.). Leaf nodes are typically **Conditions** and **Actions**. The first usually make queries about the game state, while the second make decisions and carry out specific tasks.

5.2 Behaviour Trees for Mario

It is important to understand the engine mechanics when designing BTs for a specific game. In this case, at every cycle, a set of pressed buttons is required to move Mario. This impacts how to execute a given BT, as control nodes and conditions will be continuously 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:

- Sequences and Selectors, such as described above;
- Filters. These included: `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 Section 4.1), 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 pushable blocks, jumpable platforms, etc. Examples include *EnemyAhead*, and *IsJumpPlatformAhead*.
- **Actions**. These are the possible movements of Mario (see Section 4.2). 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) and *WalkLeft*. 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**, manually designed to solve specific problems. Jumps, for example, require the jump button to start unset, followed by several cycles with button pressed. Different sub-trees were programmed, from simple jumps (*JumpRightLong*, *VerticalJumpLong*, etc), to complex tasks like

UseRightGap (places Mario below a platform on his right) or *AvoidRightTrap* (detects a dead end in front of Mario, and tries to look for an escape route).

5.3 Incorporation into GE

The BT (XML) syntax was specified in the grammar, and all conditions (30), actions (8), sub-trees (19) and filters (4) were available. Evolution was free to combine these, as long as the syntax was respected. This approach proved to be too flexible, however; with no structural guidelines, most trees were quite inefficient (such as sequences of sequences, with non-firing conditions (or `NOP` instructions) at their leaves), practically impossible to read, and very system demanding to execute. To avoid these issues, three options were considered:

- Use of a repair or penalty mechanism, that either rewrites the phenotype result, or penalises phenotypes whose syntax is too cluttered;
- Use a version of context-sensitive grammars, to limit the usage of specific tree constructions when within a certain context;
- Limit certain rule combinations through the grammar.

The first option interferes with the evolutionary process, and was avoided. The second was also avoided, as it is a relatively recent approach to GE, not fully tested to this day. The option was thus to limit the syntax of BTs through the grammar. The trees that can be evolved, although still of variable size, are contrived to follow an *and-or* tree structure [Nil98], which is a recommended [Cha07] way of building BTs for game AI.

After some experimentation, 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);
- The main root selector has a final sub-tree labelled `defaultSequence`, with a sequence of actions but no conditions.

These work 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 `DefaultSequence` is executed[1]. 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.

[1]The existence of a default unconditioned behaviour is crucial; early tests without it resulted in most agents not moving, as none of the actions associated with each `BehaviourBlock` fired.

5.3.1 Block-Exchanging Genetic Operators.

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 [ND06] 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 [Koz92], but allowing the exchange of different numbers of blocks between individuals. Finally, we allowed an individual to crossover with himself, thus creating a sub-tree swap operation; this makes sense, as a potentially good `BehaviourBlock` might be located towards the end of the `rootSelector`, which would mean that its conditions are also present in previous blocks, and those blocks will be executed instead.

6 Experiments

6.1 Setup

The experimental parameters used are shown in Table 1. All individuals in the initial generation were valid [RA03], and the mutation rate was set such that, on average, one mutation event occurs per individual (regardless of their size).

Table 1: Experimental Setup

GE	Population Size	2000
	Generations	500
	Derivation-tree Depth (for initialisation)	35
	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 1 2 3 4 5 6 7 8
	Level Types	0 1

At each generation, each individual is evaluated on 18 levels (9 difficulty settings on two level types). To enforce generalisation, the set of maps for evaluation is changed at each generation; the parent population is re-evaluated with the new maps, and each individual's fitness is averaged between the previous scores and the new one. Although the offspring fitness is based on the new maps only, elitism ensures that a percentage of the (potentially more general) solutions from the parent population are kept for the next generation.

A series of runs with different random seeds were executed in parallel in a

small cluster. At the end of all runs, all best individuals were evaluated in 600 unseen maps, and the best overall solution was submitted to the competition.

6.2 Results

The four `BehaviourBlocks` of the best BT generated, are shown in Fig. 2. The first block is composed of two conditions, followed by a sequence of actions (not shown). The conditions check for any jumpable platform on the right, and if there are no obstacles in the way. The sequence of actions is composed of 15 actions and sub-trees, that make Mario jump, run to the right and fire.

The next block contains a very relevant sub-tree: `AvoidRightTrap`. It is used to escape from dead-ends, such as shown in Fig. 3; this is one of the hardest obstacles encountered in a level. The third block checks if Mario is stuck between a hole and an enemy, and if so executes a sequence of actions for jumping, shooting and running. Finally, the last (default) block contains the sequence `RunRightSafe`, `Fire` and `RunRightSafe`, instructing Mario to run to the right while shooting, and to avoid enemies and holes by jumping.

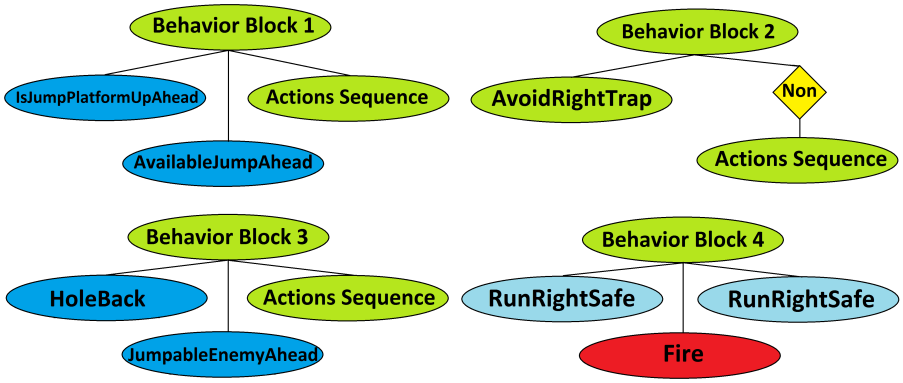

Figure 2: Behaviour tree blocks of the final individual.

This bot was sent to the competition, where the score is based on 672 trials of unseen maps, using a combination of different level types, difficulties and lengths. The results for all entries are shown in Table 2.

The first and third placed entries used variants of the A* algorithm (effectively combined with an evolutionary rule-based system, in the former), while the second placed entry used a neural-network algorithm; the entry described in this paper was the only one using an evolutionary algorithm. The results are quite close, particularly between the second and fourth place. It is worth noticing that our bot is the second entry with the most enemy kills; it suggests that the current setup is efficient at reactive behaviours, which, with the scoring system used at the competition, makes up for its lack of a path-planning

Figure 3: Level dead end. Mario has to come back and find another way.

Table 2: Competition results

Participants	Score	Disq.	Levels	Kills	Rank
S. Bojarski and C. Congdon	1789109.1	0	94	246	1
S. Polikarpov	1348465.6	4	82	156	2
R. Baumbarten	1253462.6	271	63	137	3
D. Perez and M. Nicolau	**1181452.4**	**0**	**62**	**173**	**4**
R. Reynolds and E. Speed	804635.7	0	16	86	5
A. Buck	442337.8	0	4	65	6
E. Wong	438857.6	0	0	27	7

approach.

7 Conclusions

This paper presented a novel application of a grammar-based form of Genetic Programming to the evolution of controllers for the Mario AI Benchmark, using a Behaviour Tree representation. The use of a grammar simplifies the task of encoding the syntax of BTs; not only that, but specific tree structures can be easily specified, such as and-or trees, which were used in this approach.

The encoding of crossover points in the grammar also worked to great effect in this approach. There has been a great dispute over the years as to the real exploitation nature of crossover, and in fact to the existence of exchangeable building-blocks in Genetic Programming [Ang97, SOG03]. In this work, they do exist, and the crossover operator was encoded to take full advantage of this fact.

These results obtained strengthen the idea that GP systems are serious alternatives to more traditional AI algorithms, either on their own or combined

into hybrid systems. While the current approach may not excel at planning, instead relying on high-level functions to manoeuvre challenging obstacles, it shows remarkable reactive behaviour capabilities, such as enemy shooting and close range obstacle avoidance. Future work should address this issue. A hybrid approach is under consideration, using a more effective algorithm for path planning, while retaining the remarkable reactiveness of the evolutionary approach using BTs.

Acknowledgments

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

References

[Ang97] P. Angeline: Subtree Crossover: Building Block Engine or Macromutation?. In: Genetic Programming 1997, Proceedings. Morgan Kaufmann (1997) pp. 9–17

[CDC10] A. Champandard, M. Dawe and D. H. Cerpa: Behavior Trees: Three Ways of Cultivating Strong AI. In: Game Developers Conference, Audio Lecture. (2010)

[Cha07] A. Champandard: Behavior Trees for Next-Gen Game AI. In: Game Developers Conference, Audio Lecture. (2007)

[CH07] R. Colvin and I. J. Hayes: A Semantics for Behavior Trees. ARC Centre for Complex Systems, tech. report ACCS-TR-07-01. (2007)

[Gol89] D. E. Goldberg: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley (1989)

[Isl05] D. Isla: Managing Complexity in the Halo 2 AI System. In: Game Developers Conference, Proceedings. (2005)

[Koz92] J. R. Koza: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)

[LBC10] C. Lim, R. Baumgarten and S. Colton: Evolving Behaviour Trees for the Commercial Game DEFCON. In: Applications of Evolutionary Computation, EvoStar 2010, Proceedings. (2010)

[MNW10] 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

[Mch07] L. McHugh: Three Approaches to Behavior Tree AI. In: Game Developers Conference, Proceedings. (2007)

[MMM10] 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, EvoStar 2010, Proceedings. (2010)

[MS04] M. Mateas and A. Stern: Managing Intermixing Behavior Hierarchies. In: Game Developers Conference, Proceedings. (2004)

[ND06] 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

[Nil98] N. J. Nilsson: Artificial Intelligence, A New Synthesis. Morgan Kaufmann Publishers. (1998)

[NL04] S. Nason and J. Laird: Soar-RL: Integrating Reinforcement Learning with Soar. In: International Conference on Cognitive Modelling, Proceedings. (2004)

[OR03] M. O'Neill and C. Ryan: Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language. Kluwer Academic Publishers (2003)

[Pri09] S. Priesterjahn: Imitation-Based Evolution of Artificial Game Players. ACM SIGEVOlution, 2(4). (2009) pp. 2–13

[RA03] 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. pp. 142–145. AAAI (July 2003)

[SOG03] 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

[TBS04] 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)

[TKB10] J. Togelius, S. Karakovskiy and R. Baumgarten: The 2009 Mario AI Competition. In: IEEE Congress on Evolutionary Computation, Proceedings. IEEE Press (2010)

[TKK09] J. Togelius, S. Karakovskiy, J. Koutnik and J. Schmidhuber: Super Mario Evolution. In: IEEE Symposium on Computational Intelligence and Games, Proceedings. IEEE Press (2009)