A Genetic Programming Framework for 2D Platform AI

Swen E. Gaudl[1]

Abstract. There currently exists a wide range of techniques to model and evolve artificial players for games. Existing techniques range from black box neural networks to entirely hand-designed solutions. In this paper, we demonstrate the feasibility of a genetic programming framework using human controller input to derive meaningful artificial players which can, later on, be optimised by hand. The current state of the art in game character design relies heavily on human designers to manually create and edit scripts and rules for game characters. To address this manual editing bottleneck, current computational intelligence techniques approach the issue with fully autonomous character generators, replacing most of the design process using black box solutions such as neural networks or the like. Our GP approach to this problem creates character controllers which can be further authored and developed by a designer it also offers designers to included their play style without the need to use a programming language. This keeps the designer in the loop while reducing repetitive manual labour. Our system also provides insights into how players express themselves in games and into deriving appropriate models for representing those insights. We present our framework, supporting findings and open challenges.

1 Introduction

The design of intelligent systems is a complex task which in itself can benefit from the application of AI techniques. Here we present a system that offers the developer the option to mine human behaviour and include it into the system to create better Game AI. We detail a genetic programming (GP) system that generalises from and improve upon human game play. More importantly, the resulting representations are amenable to further authoring and development. We discuss our GP system for evolving game characters by utilising recorded human play. The system uses the platformerAI toolkit, detailed in section 3, and the JAVA genetic algorithm and genetic programming package (JGAP) [7]. JGAP provides a system to evolve computer programs and their representations as decision tree when given a set of command genes, a fitness function, a genetic selector and an interface to the target application. Once the system it set up by including those components, it generates artificial players by creating and evolving JAVA program code which is fed into the PLATFORMERAI toolkit and evaluated using our fitness function which is detailed in [4].

The rest of this paper is organised as follows. In section 2 we describe how our system derives from and improves upon the start of the art. Section 4 describes our system and its core components, including details on our the design of fitness functions. We conclude our work by describing our findings and possible open challenges.

[1] MetaMakers Institute, UK, email: swen.gaudl@gmail.com

2 Background & Related Work

In practice, making a good game is achieved by a good concept and long iterative cycles in refining mechanics and visuals, a process which is resource consuming. It requires a large number of human testers to evaluate the qualities of a game. Thus, analysing tester feedback and incrementally adapting games to achieve better play experience is tedious and time-consuming. Reducing some part of the laborious work is where our approach comes into play by trying to minimise development, manual adaptation and testing time, yet allow the developer to remain in full control.

Agent Design was initially no more than creating 2D shapes on the screen, e.g. the aliens in SPACEINVADERS. Due to early hardware limitations, more complex approaches were not feasible. With more powerful computers it became feasible to integrate more complex approaches such as finite state machines (FSMs). In 2002 Isla introduced the BEHAVIOURTREE (BT) for the game Halo, later elaborated by Champandard [2]. BT uses a directed acyclic graph to represent the reasoning process within the game logic. It integrates hierarchical structures as well offering the system to scale based on the requirements but does not have the same disadvantages of FSMs, namely the exponential amount if transition checks required to verify the functionality of the FSM. BT has become the dominant approach in the industry. BTs can be represented as a combination of a decision tree (DT) using a pre-defined set of node types. A related academic predecessor of the BT were the POSH dynamic plans of BOD [1, 3].

Generative Approaches build models to create better and more appealing agents. To achieve their goal, a generative agent uses machine learning techniques to increase its capabilities by testing and updating its components. Using data derived from human interaction with a game—referred to as human play traces—can allow the game to act on or *re-act* to input created by the player. By training on such data, it is possible to derive models able to mimic certain characteristics of players [5, 8] . One obvious disadvantage of this approach is that the generated model only learns from the behaviour exhibited in the data provided to it. Thus, interesting behaviours are not accessible because they were never exhibited by a player.

In contrast to other generative agent approaches [9, 15, 8] our system combines features which allow the generation and development of truly novel agents. Thus, the system presents the first use of unauthored recorded player input as direct input into our fitness function. It allows the specification of agents only by playing. The second feature of the system is that our agents are actual programs in the form of either JAVA code or decision tree representations which can be altered and modified after evolving into a desired state, creating a white box solution. While [13] use neural networks (NN) to create better agents and enhance games using Neuroevolution, we utilise genetic programming [10] for the creation and evolution of artificial players in human readable and modifiable form. The most compa-

rable approach is that of [9] which use grammar based evolution to derive BTs given an initial set and structure of subtrees. In contrast, we start with a clean slate to evolve our agents as directly executable programs.

3 Setting and Environment

Evolutionary algorithms have the potential to solve problems in vast search spaces, especially if the problems require multi-parameter optimisation [11, p.2]. For those problems, humans are generally outperformed by programs [12]. Our GP approach uses a pool of program chromosomes P and evolves those in the form of decision trees (DTs) exploring the possible solution space. For our experiments the PLATFORMERAI toolkit (`http://www.platformersai.com`) was used which is entirely written in *Java* and freely available. It consists of a 2D platformer game, similar to existing commercial products and contains modules for recording a player, controlling agents and modifying the environment and rules of the game.

The *Problem Space* is defined by all actions an agent can perform. Within the game, agent A has to solve the complex task of selecting the appropriate action each given frame. The game consists of A traversing a level which is not fully observable. A level is 256 spatial units long, and A should traverse it left to right. Each level contains objects which act in a deterministic way. Some of those objects can alter the player's score, e.g. coins. Those bonus objects present a secondary objective. The goal of the game, move from start to finish, is augmented with the objective of gaining points. A can get points by collecting objects or jumping onto enemies. To make it comparable to the experience of similar commercial products we use a realistic time frame in which a human would need to solve a level, 200 time units. The level observability is limited to a 6×6 grid centred around the player, cf. [9]. The restriction to a smaller grid is only necessary to reduce the number of generations the system needs to converge towards good results as the grid size has an exponential affect on the convergence time.

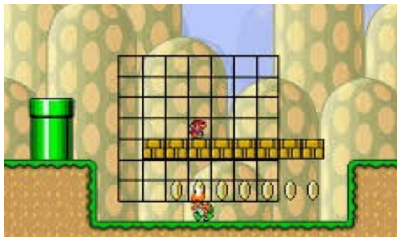

Figure 1: A visual representation of the PLATFORMERSAI toolkit with the vision grid around the agent.

Agent Control within the platformersAI toolkit is handled through a 6-bit vector C: $left$, $right$, up, $down$, $jump$ and $shoot|run$. The vector is required each frame, simulating an input device to control the agent in Figure1. However, some actions span more than one frame. This is a simple task for a human but quite complex to learn for an agent. One such example, the high jump, requires the player to press the jump button for multiple frames. Those long action sequences mean that the agent needs to anticipate future events and actions to trigger actions spanning multiple reasoning cycles. Our system has genes for each of the elements of C plus 14 additional genes formed of five gene types: sensory information about the level or agent, executable actions, logical operators, numbers and structural genes. All those are combined at execution time into a chromosome represented as a DT using the grammar underlying the JAVA language. Structural genes allow the execution of n genes in a fixed sequence, reducing the combinatorial freedom provided by JAVA. Our system uses the JGAP framework, which allows us to add new genes to enrich the search space and the agent capabilities by writing self-contained JAVA methods and adding them to the Agent class. However, adding more genes increases the search space resulting potentially in longer conversion times.

Parameter	Value
Initial Population Size	100
Selection	Weighted Roulette Wheel
Genetic Operators	Branch Typing CrossOver and Single Point Mutation
Initial Operator probabilities	0.6 crossover, 0.2 new chromosomes, 0.01 mutation, fixed
Survival	Elitism
Function Set	$ifelse$, not, &&, ||, sub, $IsCoinAt$, $IsEnemyAt$, $IsBreakAbleAt$, ...
Terminal Set	Integers [-6,6], $\leftarrow$, $\rightarrow$, $\downarrow$, $IsTall$, $Jump$, $Shoot$, Run $Wait$, $CanJump$, $CanShoot$, ...

Table 1: GP parameters used in our system.

4 Fitness Evaluation

The evaluation is done in our system using the Gamalyzer-based play trace metric which determines the fitness of individual chromosomes based on human traces as an evaluation criterion, see [4]. For finding optimal solutions to a problem, statistical fitness functions offer near-optimal results when optimality can be defined. A near-best solution for the problem space of finding the optimal way through a level in the platformersAI toolkit was given by Baumgarten [14] using the A^* algorithm. This approach produces agents who are extremely good at winning the level within a minimum amount of time but at the same time are clearly distinguishable from actual human players. Contrasting the goal of finding optimal solutions, we are interested in understanding and modelling human-like or human-believable behaviour in games. Thus, using statistical functions is difficult, as there currently is no known algorithm for measuring how human-like behaviour is; identifying this may even be computationally intractable. For games and game designers a less distinguishable approach is normally more appealing—based on our initial assumptions. Additionally, having an approach which produces readable and amenable representations of the behaviour might not just aid its understanding but might offer different insights into the design of the game as well.

Based on the biological concept of selection, all evolutionary systems require some form of judgement about the quality of a specific individual—the fitness value of the entity. Within our framework, agents are evaluated after each run of an entire level of the game as intermittent evaluation of games where actions can span multiple cycles is difficult to evaluate. Within the original JGAP framework evaluation can be done at arbitrary times but it an important consideration that the evaluation (running the program to receive a result) is normally the most expensive cost within a GP.

In table 1 the settings we use for GP within our framework are given. As a selection mechanism, the weighted roulette wheel is

used which attributes each chromosome a position and then weights all chromosomes according to their fitness giving fitter individuals slightly more space. We additionally preserve the fittest individual of a generation. Preserving the best individual is crucial as mutation can be destructive to the chromosome We use single point tree branch crossover on two selected parent chromosomes and expose the resulting child to a single point mutation before it is put into the new generation. We also add 20% new randomly generated chromosomes to the pool to bring in some "fresh blood" or to be more precise keep the pool from stopping in a homogeneous state. Even through mutation is potentially destructive, it helps exploring the vast gene space better than relying entirely on the cross-over operation. However, within our experiments [4], using the more stable cross-over as the main driving force for the evolution gave better and more reliant results than switching entirely to random exploration using a stronger mutation coefficient.

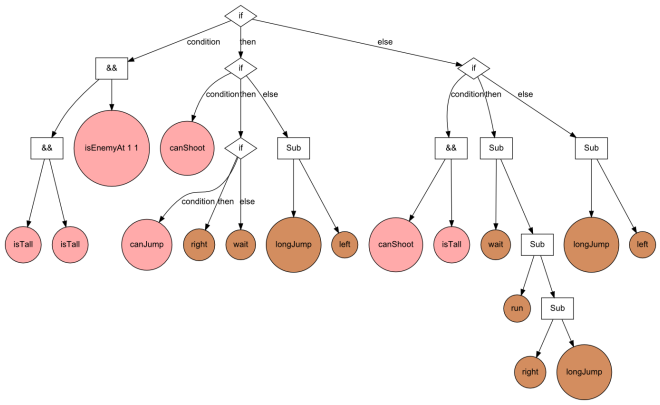

Figure 2: An evolved agent after 694 generations, represented as decision tree by our system.

In Figure 2 one of the resulting agents is presented in its DT form. The visual representation was generated by the system using GRAPHVIZ (`http://www.graphviz.org/`). As the aim of our approach was to derive meaningful representations of agent behaviour, visual representation of the result is of utmost importance. Using the rendered DT allows a designer to either alter the agent or to understand why it behaved in a certain way.

5 Findings & Open Challenges

Using our experimental configuration and the PBF fitness function [4] we are now able to execute, evaluate and compare platformerAI agents against human traces. Using human play traces to drive the evolution resulted in agents which are able to beat some but not all of the test levels. However, there is still potential using different ways to integration human knowledge into the evaluation. The JGAP framework proved to a useful and easy to use and robust framework for developing genetic programs, even though it has some weaknesses compared to other frameworks. If you care for running the GP on a cluster you might decide to use a different framework which offers better support for spitting up both the evaluation of chromosomes and the handling large data structures. Most of the GP systems let you also run or communicate external libraries. In our case, we included the PLATFORMERSAI toolkit to evaluate our agents. This toolkit does not support parallel instantiations of multiple levels well but can be tweaked easily and offers also support for using a genetic approach to evolve levels. A next step would be to investigate the generated modifiable programs further and analyse their benefit in understanding players better. However, our current solution already offers a way to design agents for a game by simply playing it and creating learning agents from those traces. Other possible directions could be the comparison of different fitness functions and how different interpretations of human play input might affect the convergence rate of agents within our framework. Our current agent model consists of an unweighted tree representation containing program genes. Currently subtrees are not taken into consideration when calculating the fitness of an individual. By including those weights it would be possible to narrow down the search space of good solutions for game characters dramatically, also potentially reducing the bloat of the DT. So, to enhance the quality of our reproduction component we believe it might be interesting to investigate the applicability of behavior-programming for GP (BPGP) [6] into our system.

REFERENCES

[1] Joanna J. Bryson and Lynn Andrea Stein, 'Modularity and design in reactive intelligence', in *Proceedings of the* 17^{th} *International Joint Conference on Artificial Intelligence*, pp. 1115–1120, Seattle, (August 2001). Morgan Kaufmann.

[2] Alex J. Champandard, *AI Game Development*, New Riders Publishing, 2003.

[3] Swen E. Gaudl, Simon Davies, and Joanna J. Bryson, 'Behaviour oriented design for real-time-strategy games – an approach on iterative development for starcraft ai', in *Proceedings of the Foundations of Digital Games*, pp. 198–205. Society for the Advancement of Science of Digital Games, (2013).

[4] Swen E Gaudl, Joseph Carter Osborn, and Joanna J Bryson, 'Learning from play: Facilitating character design through genetic programming and human mimicry', in *Portuguese Conference on Artificial Intelligence*, pp. 292–297. Springer, (2015).

[5] C. Holmgard, A. Liapis, J. Togelius, and G.N. Yannakakis, 'Evolving personas for player decision modeling', in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pp. 1–8, (Aug 2014).

[6] Krzysztof Krawiec and Una-May O'Reilly, 'Behavioral programming: a broader and more detailed take on semantic gp', in *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pp. 935–942. ACM, (2014).

[7] Klaus Meffert, N Rotstan, C Knowles, and U Sangiorgi. Jgap-java genetic algorithms and genetic programming package. last viewed:01.2015, 09 2000.

[8] Juan Ortega, Noor Shaker, Julian Togelius, and Georgios N. Yannakakis, 'Imitating human playing styles in super mario bros', *Entertainment Computing*, **4**(2), 93 – 104, (2013).

[9] Diego Perez, Miguel Nicolau, Michael ONeill, and Anthony Brabazon, 'Evolving behaviour trees for the mario ai competition using grammatical evolution', in *Applications of Evolutionary Computation*, ed., etal. Di Chio, volume 6624 of *Lecture Notes in Computer Science*, 123–132, Springer Berlin Heidelberg, (2011).

[10] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza, *A field guide to genetic programming*, Lulu. com, 2008.

[11] Hans-Paul Paul Schwefel, *Evolution and optimum seeking: the sixth generation*, John Wiley & Sons, Inc., 1993.

[12] Selmar K Smit and Agoston E Eiben, 'Comparing parameter tuning methods for evolutionary algorithms', in *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pp. 399–406. IEEE, (2009).

[13] Kenneth O. Stanley and Risto Miikkulainen, 'Evolving neural networks through augmenting topologies', *Evolutionary Computation*, **10**, 99–127, (2002).

[14] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten, 'The 2009 mario ai competition', in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1–8. IEEE, (2010).

[15] Julian Togelius, GeorgiosN. Yannakakis, Sergey Karakovskiy, and Noor Shaker, 'Assessing believability', in *Believable Bots*, ed., Philip Hingston, 215–230, Springer Berlin Heidelberg, (2012).