Learning from Play: Facilitating character design through genetic programming and human mimicry

Swen E. Gaudl, Joseph C. Osborn, and Joanna J. Bryson

[1] University of Bath, Dept. of Computer Science swen.gaudl@gmail.com
[2] University of California SC, Baskin School of Engineering jcosborn@soe.ucsc.edu
[3] University of Bath, Dept. of Computer Science jjb@bath.ac.uk

Abstract. Mimicry and play are fundamental learning processes by which individuals can acquire behaviours, skills and norms. In this paper we utilise these two processes to create new game characters by mimicking and learning from actual human players. We present our approach towards aiding the design process of game characters through the use of genetic programming. 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. Computational creativity approaches this issue with fully autonomous character generators, replacing most of the design process using black box solutions such as neural networks. Our GP approach to this problem not only mimics actual human play but creates character controllers which can be further authored and developed by a designer. This keeps the designer in the loop while reducing repetitive 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 and preliminary results supporting our claim.

Keywords: agent design, machine learning, genetic programming, games

1 Introduction

Designing intelligence is a sufficiently complex task that it can itself be aided by the proper application of AI techniques. Here we present a system that mines human behaviour to create better Game AI. We utilise genetic programming (GP) to generalise from and improve upon human game play. More importantly, the resulting representations are amenable to further authoring and development. We introduce a 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) [6]. JGAP provides a system to evolve agents when given a set of command genes, a fitness function, a genetic selector and an interface to the target application. Thereafter, our system generates players by creating and evolving Java program code which is fed into the platformerAI toolkit and evaluated using our player-based fitness function.

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 3 describes our system and its core components, including details of our fitness function. We conclude our work by describing our initial results and possible future work.

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. This 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 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 from science. In 2002 Isla introduced the BehaviourTree (BT) for the game Halo, later elaborated by Champandard [2]. BT has become the dominant approach in the industry. BTs are a combination of a decision tree (DT) with 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 [4, 7] build models to create better and more appealing agents. In turn, a generative agent uses machine learning techniques to increase its capabilities. Using data derived from human interaction with a game—referred to as human play traces—can allow the game to act on or *react* to input created by the player. By training on such data it is possible to derive models able to mimic certain characteristics of players. 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, 7] our system combines features which allow the generation and development of truly novel agents. The first is the use of un-authored recorded player input as direct input into our fitness function. This allows the specification of agents only by playing. The second feature is that our agents are actual programs in the form of java code which can be altered and modified after evolving into a desired state, creating a white box solution. While Stanley and Miikkulainen [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 comparable approach is that of Perez et al. [9] which uses grammar based evolution to derive BTs given an initial set and structure of subtrees. In contrast, we start with a clean slate to evolve novel 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. 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. Perez et al. [9].

Agent Control 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. 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. Our system has a gene for each element 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 on creation 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.

Evaluation of Fitness in our system is done using the Gamalyzer-based play trace metric which determines the fitness of individual chromosomes based on human traces as an evaluation criterion. For finding optimal solutions to a problem statistical fitness functions offer near-optimal results when optimality can be defined. We are interested in understanding and modelling human-like or human-believable behaviour in games. There is no known algorithm for measuring how human-like behaviour is; identifying this may even be computationally intractable. A near-best solution for the problem space of finding the optimal way through a level was given by Baumgarten [14] using the A^* algorithm. This approach produces agents which 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. For games and game designers a less distinguishable approach is normally more appealing—based on our initial assumptions.

4 Fitness Function

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. Our *Player Based Fitness* (PBF) uses multiple traces of human, t_h, and agent, t_a, players to derive a fitness value by judging their similarity. For that purpose we integrate the Gamalyzer Metric—a game independent measurement of the difference between two play traces. It is based on the syntactic edit distance d_{dis} between pairs of sequences of player inputs [8]. It takes pairs of sequences of events gathered during a game play along with designer-provided rules for comparing individual events and yields a numerical value in $[0, 1]$. Identical traces have distance $d_{dis} = 0$ and incomparably different traces $d_{dis} = 1$. Gamalyzer finds the least expensive way to turn one play trace into another by repeatedly deleting an event from the first trace, inserting an event of the second trace into the first trace, or changing an event of the first trace into an event of the second trace. The game designer or analyst must also provide a comparison function which describes the difficulty of changing one event into another. The other important feature of Gamalyzer, warp window ω, is a constraint that prevents early parts of the first trace from comparing against late parts of the second. This is important for correctness (a running leap at the beginning of the level has a very different connotation from a running leap at the pole at the end of each stage). For our purpose, only the input commands players use to control the agent are encoded—the six commands introduced earlier. This allows us to compare against direct controller input for future studies and to help designers sitting in front of the controls analysing the resulting character program. The PBF currently offers two parameters: the chunk size, cpf, and the warp window size, ω. The main advantage over a pure statistical fitness function is that a designer can feed our system specific play traces of human players without having to modify implicit values of a fitness score.

To make a stronger emphasis on playing the game well, we create a multi-objective problem using an aggregation function g to take Δd—the moved distance—and the fitness $f(a)$ for an agent using the playerbased metric PBF into account, see formula (1). Using g we were able to put equal focus on the trace metric, $f_{ptm} \in [0 \dots 1] \subset \mathbb{R}$, and the advancement along the game, $\Delta d \in [0 \dots 256] \subset \mathbb{N}$.

$$f(a) = g(f_{ptm}(t_a, t_h), \Delta d) \tag{1}$$

5 Preliminary Results & Future Work

Using our experimental configuration and the PBF fitness function we are now able to execute, evaluate and compare platformerAI agents against human traces.

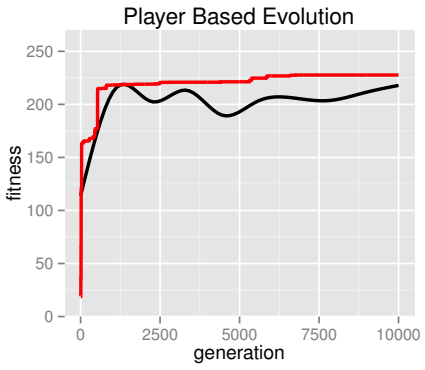

Fig. 1. The evolved agents' fitness using PBF (10000 generations), in red the fittest individuals, in black the averaged fitness of all agents per generation.

Table 1. GP parameters used in our system.

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 $Wait$, $CanJump$, $CanShoot$, ...

We are using the settings supplied in table 1. As a selection mechanism, the weighted roulette wheel is used and we additionally preserve the fittest individual of a generation. 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. Figure 1 illustrates the convergence of the program pool against the global optimum. Good solutions are on average reached after 700 generations, when an agent finishes the given level. Our first experiments show that our approach is able to train on and converge against raw human play traces without stopping at local optima, visible in the two dents of the averaged fitness (black) diverging from the fittest individual (red). 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 expansion of the model underlying Gamalyzer to model specific events within the game rather than pure input actions. This should provide interesting feedback and offer a better matching of expressed player behaviour and model generation. 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) [5] into our system.

References

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

[2] Champandard, A.J.: AI Game Development. New Riders Publishing (2003)

[3] Gaudl, S.E., Davies, S., Bryson, J.J.: 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] Holmgard, C., Liapis, A., Togelius, J., Yannakakis, G.: Evolving personas for player decision modeling. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 1–8 (Aug 2014)

[5] Krawiec, K., O'Reilly, U.M.: 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)

[6] Meffert, K., Rotstan, N., Knowles, C., Sangiorgi, U.: Jgap-java genetic algorithms and genetic programming package. last viewed:01.2015 (09 2000), `http://jgap.sf.net`

[7] Ortega, J., Shaker, N., Togelius, J., Yannakakis, G.N.: Imitating human playing styles in super mario bros. Entertainment Computing 4(2), 93 – 104 (2013)

[8] Osborn, J.C., Mateas, M.: A game-independent play trace dissimilarity metric. In: Proceedings of the Foundations of Digital Games. Society for the Advancement of Science of Digital Games (2014)

[9] Perez, D., Nicolau, M., ONeill, M., Brabazon, A.: Evolving behaviour trees for the mario ai competition using grammatical evolution. In: Di Chio, e. (ed.) Applications of Evolutionary Computation, Lecture Notes in Computer Science, vol. 6624, pp. 123–132. Springer Berlin Heidelberg (2011)

[10] Poli, R., Langdon, W.B., McPhee, N.F., Koza, J.R.: A field guide to genetic programming. Lulu. com (2008)

[11] Schwefel, H.P.P.: Evolution and optimum seeking: the sixth generation. John Wiley & Sons, Inc. (1993)

[12] Smit, S.K., Eiben, A.E.: Comparing parameter tuning methods for evolutionary algorithms. In: Evolutionary Computation, 2009. CEC'09. IEEE Congress on. pp. 399–406. IEEE (2009)

[13] Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation 10, 99–127 (2002)

[14] Togelius, J., Karakovskiy, S., Baumgarten, R.: The 2009 mario ai competition. In: Evolutionary Computation (CEC), 2010 IEEE Congress on. pp. 1–8. IEEE (2010)

[15] Togelius, J., Yannakakis, G., Karakovskiy, S., Shaker, N.: Assessing believability. In: Hingston, P. (ed.) Believable Bots, pp. 215–230. Springer Berlin Heidelberg (2012)