Single Fitness Layered-Incremental Gene Expression Programming for Robot AI in Minecraft

Connor Hillen
connor.hillen@carleton.ca

December 16, 2016

Abstract

Evolutionary approaches to robot behaviour generation has been widely studied, however gene expression programming is a technique which has only recently emerged in the field. Additionally, practical robotics behaviour generation is limited to the capability of physical robotics. In this paper, I propose a method for testing GEP as a technique for robotics behaviour generation using a combination of incremental and layered GEP training and show how GEP can be implemented in the much richer environment of Mojang's *Minecraft*.

1 Introduction

Accounting for various environments in the real world is a difficult problem to solve using static programming and evolutionary approaches have been studied for decades with varying degrees of predetermined models [1]. Many techniques have been used to evolve programs, but genetic programming has been very widely adopted in evolutionary robotics and various different techniques for training have emerged. Gene expression programming seems to have only recently started up in the evolutionary robotics field, however Mwuara and Jonathon show promising results that GEP could function better than traditional methods while having a simpler implementation [2].

In this paper I have sought to research different techniques of training robotic behaviours using evolutionary approaches and testing that combined approach with Microsoft's *Malmo* framework [3] to see how evolutionary robotics could apply to richer simulated worlds.

1.1 Combined Approach

Various methods were examined when coming up with a combined approach to GEP training. These methods are closer looked at in the background and related works sections of this paper. A common theme is a constantly changing fitness function for more complex behaviour, however the medium the algorithm is working in is the videogame *Minecraft* [4] where the player's only goals are to explore and to survive. Given the wild complexity of behaviours that these goals can produce with humans when put into various scenarios, I was interested in whether this basic fitness evaluation of survive and explore would be capable of producing a monolithic AI comprising of various interesting, scenario specific behaviours without having to make new complex fitness functions for every scenario.

1.2 Minecraft and Malmo

In the methodology section I describe some of the implementation details of getting the GEP algorithm to function in Minecraft and some of the training scenarios used. The Malmo interface provided multiple different levels of sensory and actuator realism which will be briefly described. A major issue that arose in testing was the lack of simple parallelism of clients, meaning that it was difficult to get the populations needed for good GEP testing. To account for this, fewer scenarios were tested to speed up individual training.

2 Background

2.1 Gene Expression Programming

Gene expression programming is a technique developed by Ferreira in 2001[5] which combines the simple, linear string of fixed lengths seen in genetic algorithms with the complex, dynamic tree structures found in genetic programming. The techniques used are very similar to those in genetic algorithms,

however instead the whole chromosome being equal, the chromosome is broken up into a *head* and a *tail*. The head of a chromosome may contain operators or terminals, similar to that of a GP tree, but the tail can only hold terminals. This same rule applies to the operations performed during reproduction so that the chromosome will always be kept valid. An advantage of GEP over GP is that multiple genetic operators can be quickly operated on a chromosome in a generation, leading to highly diverse populations quickly. Additionally, the issue of bloat in GP is taken care of by having a fixed length string representing the chromosome without sacrificing variable length expression trees.

2.2 Minecraft and the Malmo Framework

Minecraft is one of the top selling games of all time with over 100 million copies sold internationally [6] and was acquired by Microsoft in 2014 [7]. The game features exploration, survival, and creation with little guidance. The player takes control of an avatar and is placed into the middle of a world which procedurally generates structures and environments as they explore. The player has a few basic goals: explore, survive, and acquire resources. The world consists of various types of blocks which the user can pick up, place, or destroy, as well as various fluids and non-player characters and items. In some game modes, the player requires food to survive which can be acquired via agriculture, husbandry, hunting, trading, or gathering. The game also features a complex circuitry platform, known as redstone, for developing mechanical constructs.

After the acquisition by Microsoft in 2014, Malmo was developed to give developers a way to interact with the Minecraft world as a robotics simulation. It has built in Q-learning [8] capabilities for basic AI testing as well as various challenges presented in an incremental way to train the developer. Each AI is referred to as an *agent* and an agent is distributed to a Minecraft *client* and given a *mission*. A mission is an xml document, a sample shown in **Figure 1** which contains information regarding the world to be generated, the conditions which terminates the mission, a description of what actuators and sensors the agent will use to navigate the world, and any reinforcement information if Q-learning is to be used.

```
<ServerSection>
  <ServerInitialConditions>
      <Time><StartTime>1</StartTime></Time>
  </ServerInitialConditions>
  <ServerHandlers>
    <FlatWorldGenerator forceReset="true" generatorString="3;7,220*1,5*3,2;3;,biome_1"/>
    <DrawingDecorator>
      <!-- coordinates for cuboid are inclusive -->
      <DrawCuboid x1="-5" y1="41" z1="-5" x2="5" y2="50" z2="5" type="air" />
      <DrawCuboid x1="-5" y1="50" z1="-5" x2="5" y2="50" z2="5" type="glowstone" />
      <DrawCuboid x1="-5" y1="45" z1="-5" x2="5" y2="45" z2="5" type="sandstone" />
      <DrawCuboid x1="-2" y1="45" z1="-2" x2="2" y2="45" z2="2" type="lava" />
    </DrawingDecorator>
    <ServerQuitFromTimeUp timeLimitMs="4000"/>
    <ServerQuitWhenAnyAgentFinishes/>
  </ServerHandlers>
</ServerSection>
```

Figure 1: Malmo world generation code from a lava mission used in the implementation.

3 Related Work

The previous section described valuable knowledge of the tools and techniques used in this particular project. This section focuses on the various ways that evolutionary techniques have been used in the past to evolve robotics behaviour. Note that techniques used to train genetic programs can also be used with gene expression programming and will be looked at similarly [2].

3.1 Incremental Training

Incremental training has been used in a variety of AI fields, including evolutionary robotics [9] and neural networks [10]. Incremental training is a technique to develop high level behaviours using low level scenario tests. In practice, it is capable of finding more interesting behaviours and produces better results than traditional approaches [9] [10][2].

In incremental training the AI being trained is first presented with an environment and a fitness score intended to train one particular low level behaviour, such as simple wall avoidance. Once this has been trained it is then trained on a new environment with a new fitness. These training scenarios will build a converged AI capable of achieving multiple different sub-behaviours. The difficulty in developing an incremental training set is making sure that the environments and fitness functions chosen promote con-

vergence. A standard approach is to make the increments simply environments of increased complexity and hopefully some members of the population will survive. [9].

This approach seems to work well with this paper's goal of making a more generalized survival agent; however, simply increasing the complexity of an environment each test to guarantee the cohesiveness of sub-behaviours seems like a sacrifice when the world of Minecraft is so heavily varied.

3.2 Multi-Objective Training

Multi-Objective algorithms have been used for many different problems and evolutionary algorithms offer a unique perspective on solving them [11]. The multi-objective approach is simple; rather than maximizing a single goal, there are multiple different goals which must be maximized and converged upon. There are many techniques available to deal with converging the fitness scores at the end of training to come up with a solution that promotes both sub-behavioural strength and diversity [12]. The simplest way is of course to simply take a weighted sum of each individual fitness score, but this might not properly reflect the problem space.

This simple weighting scheme would appear to work well when the different objectives are very similar. Multi-objective optimization is one of the main inspirations and the closest to what this paper is working to emulate. Additionally, there is a variant of MOO which uses static, staged fitness evaluators [12] which was hard to find literature on, but seemed very appealing.

3.3 Staged Evaluation for MOO

Staged evaluators were used by Koza in 1999 [13] while designing circuits due to the many parameters faced in circuit design. It seemed appropriate to only measure against some of the parameters until the system was already well versed in simpler cases.

This seems like a very appropriate analog for a Minecraft AI where there are many different parameters which can influence simple tasks like wall avoidance and exploration.

3.4 Layered Learning

Layered learning has been used with GAs and evolutionary robotics for a variety of tasks, including cooperative robotics soccer games [14]. Layered learning follows a subsumption architecture, in which each behaviour is a separately trained layer which specializes at a particular task. Each fitness is run separately with a different evaluator and stops when the individual behaviour has a particular fitness. This runs into the issues of rigidity and long run times as calculating multiple fitness functions for each behaviour is lengthy when simulating robotics [2].

The layered approach is interesting, as while evolving varied behaviours at the same time takes a long time to test it does mean that each behaviour has an equal say in the development of the robot's monolithic behaviour, at least as determined by the behavioural arbitrator in the subsumption architecture.

4 Methodology

So far in this paper I've described various techniques used to evolve robots in the past and described some interesting cases for testing these techniques outside of simply robotics. Based on these techniques and the goals of the Minecraft case, this section will describe the methods of combining some of these ER algorithms for use in GEP with Minecraft.

4.1 Type of Agent Used

In Malmo, there are various different configurations of agents that can be used. In particular, an agent can move either discretely, from block to block, or continuously. Additionally each agent can either take sensory input purely from video feed, video feed plus depth map, or simply be given an array of blocks in a particular range.

For this implementation, a simple discrete agent with a small, 3x3x3 visual range is used, where all actions are simplified as described below.

4.2 GEP Parameters

GEP, like GP, has a set of functions and a set of terminals. GEP also has several evolutionary operators available which helps maintain a good amount of diversity.

The functions set in evolutionary robotics is generally kept minimal and only perform logical operations or simple arithmetic. The terminal set is usually restricted to sensors and/or actuators.

As there are multiple different categories the terminals can represent, either movement or sensory input, there are different approaches to handling what is included in the set. Pilat and Oppacher use a terminal set containing only sensor reading and use arithmetic operators to convert these values into motor output [15]. In this paper's implementation for Minecraft, the operator and terminal set are based on Lazarus and Hu's GP wall following parameters [16].

Lazarus and Hu's terminal set is defined as two sets of terminals; sensor terminals and action terminals. Rather than treating these as different genes, however, Mwuara adapts these to use the same symbols and the symbols are interpreted differently depending on the position they are used in the function calling them [2]. For example, a simple *if(x,y,z)* operator could have the inputs *if(F,F,M)*. The first F would be interpreted as a *sensor* while the second F would refer to an actuator output. This makes it so that operators do not have to have an opinion on what category the terminal is, much the same as Pilat and Oppacher's implementation. This prevents a lot of maintenance overhead and works well with a GEP implementation which already reduces the maintenance overhead of tree management. Mwaura was able to show that simple wall avoidance could be done with a single function, *IFLTE(w,x,y,z)*, which acts as: *If* $\boldsymbol{w}$ *less than* $\boldsymbol{x}$*, then* $\boldsymbol{y}$*, else* $\boldsymbol{z}$. Additionally, Mwaura also used a method where an action terminal in a sensor terminal location would simply force the operation to take the action represented by the action terminal.

There are various techniques available and these had to be adapted for use in Minecraft.

4.2.1 Terminal Set Used

Lazarus and Hu were working in a very simple simulation consisting of only obstacles and discrete movements. The Minecraft agent will have rotation, a frontal view, and will have to differentiate between multiple types of obstacles. Thus, there are three types of terminals: sensor terminals, block terminals, and action terminals. Sensor terminals represent the grid that the agent can see with and return a block type. Block terminals are analogous to real values in sensor output, as they are constant sensor outputs. Action

terminals are a simplified action set that the agent can take.

The sensor set was simplified to have only three sensors: A (ahead), D (ahead down), and U (ahead under). The block set contains a few blocks used in testing: L (lava), A(air (nothing)), S(sandstone (floors)), and T(stone (walls)); block and sensor terminals are interchangeable. The action set was also kept simple: F (move forward), B(move backward), L(turn left), R(turn right), 0(do nothing).

4.2.2 Function Set Used

To keep things simple in testing, Mwuara's IFLTE operator was adapted and is the only operator which ended up used. If (block = block), then action else action.

4.2.3 Fitness Used

This is also a simple function, slightly modified from Mwuara. Since the goal of the robot is survival and exploration, collisions (C, trying to move forward and failing) are bad, death (D) is very bad, new tiles ($p_i(x_i, y_i, z_i) = 1$ such that (x,y,z) are unexplored) are good, and stagnation (S) is extremely bad.

$$fitness = (\sum_{i=1}^{n} p_i(x_i, y_i, z_i)) - C - D - S$$

4.3 Combining Training Behaviours

To train the model, a combination of techniques was used. The agent is sequentially run through multiple different environments, each with a different component. Each environment has multiple increments, shown in **Figure 2**. The fitness is simply a weighted sum of each, where each weight normalizes the fitness according to the maximum fitness of each environment.

5 Discussion

While a few runs were made, little progress could be made in achieving results due to issues in distributing the population and this project turned largely into a literature review and a hypothetical. Malmo proved to be easy

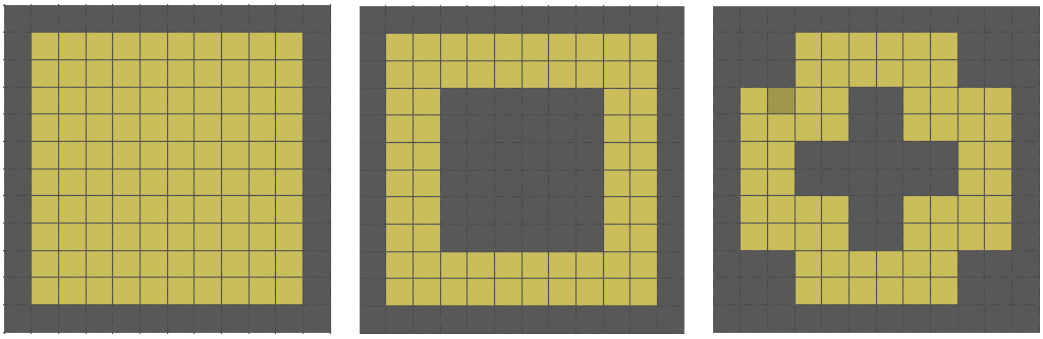

Figure 2: Three increments in the wall environment, training wall avoidance.

to use, setting up and running missions was easy, however Malmo is a bit difficult to parallelize agent distribution. Built in methods for adding new clients to run on work well, but getting agents to deploy when clients became available while other clients ran their missions was challenging. Additionally, serious cutbacks had to be made to even run a few tests, including cuts to the population size beyond anything reasonable, cuts to the amount of missions being tested, and huge reductions in agent complexity. Additionally, existing libraries for GEP in Python are a decade out of date and appear to have no modern maintenance, meaning that any GEP had to be hacked together with old beta libraries with little documentation.

This method might work under specific parameters, but running these already slow techniques in such a power intensive environment requires a lot of planning and distribution capabilities. During literature review, there was certainly a lot of promise in using this technique due to its similarities to MOO and layered GP, both of which have a lot of very satisfactory results.

6 Conclusions and Future Work

The field of evolutionary robotics is huge and constantly expanding. The introduction to GEP in the field in 2011 showed a lot of promise, however not many people seem to have worked upon this. Using it in the complex realm of Minecraft certainly does seem feasible and has potential to push the boundaries of ER, however there is a large run time and preparatory phase required to coordinate.

In the future, I plan to continue implementation efforts, updating Python's

GEP capabilities, and finding better ways of distributing the Malmo platform. Once the technical issues are out of the way, it would be interesting to see how previous approaches like those of Lazarus and Hu or Pilat and Oppacher might translate into this new platform and how they might be able to be expanded upon. Finally, it would be interesting to see how complex evolutionary behaviour algorithms like those found in Gustafson's layered RoboCup AI might be able to develop in the complex world of Minecraft, where there are many different behaviours a subsumption model could take advantage of.

References

[1] John Grefenstette and Alan Schultz. An evolutionary approach to learning in robots. Technical report, DTIC Document, 1994.

[2] Jonathan Mwaura. Evolution of robotic behaviour using gene expression programming. 2011.

[3] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *International joint conference on artificial intelligence (IJCAI)*, page 4246, 2016.

[4] Mojang. Minecraft. (Windows), November 2011.

[5] Cândida Ferreira. Gene expression programming: A new adaptive algorithm for solving problems. In *Complex Systems*, volume 13, issue 2, pages 87–129, 2001.

[6] Owen Hill. We've sold minecraft many, many times! look! https://mojang.com/2016/06/weve-sold-minecraft-many-many-times-look/, June 2016. (Accessed on 12/11/2016).

[7] Phil Spencer. Minecraft to join microsoft - xbox wire. http://news.xbox.com/2014/09/15/games-minecraft-to-join-microsoft/, September 2014. (Accessed on 12/16/2016).

[8] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[9] Dhiraj Bajaj and Marcelo H Ang Jr. An incremental approach in evolving robot behavior. In *Proceedings of the Sixth International Conference on Control, Automation, Robotics and Vision*, 2000.

[10] Stephan K Chalup and Alan D Blair. Incremental learning for rnns: How does it affect performance and hidden unit activation? 2003.

[11] Carlos A Coello Coello and Gary B Lamont. *Applications of multi-objective evolutionary algorithms*, volume 1. World Scientific, 2004.

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

[13] John Koza, F Bennett III, David Andre, and M Keane. The design of analogue circuits by means of genetic programming. *Evolutionary Design by Computers*, pages 365–385, 1999.

[14] Steven M Gustafson and William H Hsu. Layered learning in genetic programming for a cooperative robot soccer problem. In *European Conference on Genetic Programming*, pages 291–301. Springer, 2001.

[15] Marcin L Pilat and Franz Oppacher. Robotic control using hierarchical genetic programming. In *Genetic and Evolutionary Computation Conference*, pages 642–653. Springer, 2004.

[16] Christopher Lazarus and Huosheng Hu. Using genetic programming to evolve robot behaviours. In *Proceedings of the 3rd British Conference on Autonomous Mobile Robotics & Autonomous Systems*, volume 5, 2001.