GUI-Based, Efficient Genetic Programming For Unity3D

GUI-Based, Efficient Genetic Programming For Unity3D

Robert Gold
robertgold@csail.mit.edu
ALFA, MIT CSAIL
Cambridge, MA, USA

Andrew Haydn Grant
haydn@mit.edu
ALFA, MIT CSAIL
Cambridge, MA, USA

Erik Hemberg
hembergerik@csail.mit.edu
ALFA, MIT CSAIL
Cambridge, MA, USA

Chathika Gunaratne
contact@chathika.com
ALFA, MIT CSAIL
Cambridge, MA, USA

Una-May O'Reilly
unamay@csail.mit.edu
ALFA, MIT CSAIL
Cambridge, MA, USA

ABSTRACT

Unity3D is a game development environment that could be co-opted for agent-based machine learning research. We present a GUI-driven, and efficient Genetic Programming (GP) system for this purpose. Our system, ABL-Unity3D, addresses challenges entailed in co-opting Unity3D: making the simulator serve agent learning rather than humans playing a game, lowering fitness evaluation time to make learning computationally feasible, and interfacing GP with an AI Planner to support hybrid algorithms that could improve performance. We achieve this through development of a GUI using the Unity3D editor's programmable interface, and performance optimizations. These optimizations result in at least a 3x speed up. We describe ABL-Unity3D by explaining how to use it for an example experiment using GP and AI Planning.

CCS CONCEPTS

• **Computing methodologies** → **Multi-agent planning**; **Simulation tools**; **Genetic programming**; • **Human-centered computing** → **Graphical user interfaces**.

KEYWORDS

Genetic programming, Unity3D, Simulator, AI Planning, GUI

ACM Reference Format:
Robert Gold, Andrew Haydn Grant, Erik Hemberg, Chathika Gunaratne, and Una-May O'Reilly. 2022. GUI-Based, Efficient Genetic Programming For Unity3D. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3520304.3534022

1 INTRODUCTION

There are many computational simulation environments that support the development and evaluation of agent-based learning algorithms, both academic and commercial [1]. While many projects center on the learning of game play, there has been growing interest in agent-based learning (ABL) in non-game contexts, such as network security and attack planning, or Covid-19 contagion interaction protocols [5, 8]. This implies the need to simulate more realistic environments. For this, there exists well-supported and capable game development environments, such as Unity3D [9]. However, these are often focused on human game play, not AI agent-based learning. Further, game development environments co-opted to non-game playing agents present the risk of expensive fitness evaluation and inappropriate interfaces for development and evaluation of agent-based learning algorithms. ML-Agents is a Unity3D reinforcement learning library that improves upon this [6].

We aim to enable different projects to run Genetic Programming (GP) in Unity3D. We introduce ABL-Unity3D to hopefully reduce the development effort, time, and reoccurring mistakes. Our primary goal is to provide an open-source, domain-targetable agent-based learning system that demonstrates GP and AI Planning, addresses agent evaluation efficiency, and pivots the human interaction lens of Unity3D to a GP-developer user-friendly GUI. We want the system to support AI experiments that are easy to design and efficient to execute. Our aim is a baseline for running AI experiments which require 3D simulation, not a unifying framework for all AI experiments. In this paper we present ABL-Unity3D, see Figure 1, which is our project centered on these goals. The code repository for ABL-Unity3D can be found at [4].

ABL-Unity3D interfaces a 3D physics-based simulation with Genetic Programming and AI Planning, to support specific methodological investigations in agent-based learning. It uses the Unity3D game engine editor and is written in C#. Unity3D provides an easy to use, efficient, and extendable GUI and API interface to run simulations and AI. ABL-Unity3D contains a simulation world state. This state can be input to the GP and Hierarchical Task Network (HTN) Planner components [3]. The GP and HTN components can, but are not required to, act as inputs to (or outputs from) each other to create a hybrid algorithm. The outputs of the GP (typically a candidate solution that needs evaluation in a 3D simulator) and/or the outputs of the HTN (typically a plan for an agent to follow in a 3D simulator) become inputs to the simulation. As a simulation progresses, the world state updates. These updates are returned to the learning components. ABL-Unity3D provides a GUI to set parameters for, and examine, the simulator and learning components.

Our main contribution is a simulator built to be used directly with GP and AI, is easy to configure and examine what the AI does, and is domain-targetable. The ABL-Unity3D GUI uses the Unity3D editor to provide the ability to view the simulation as it is running,

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9268-6/22/07.
https://doi.org/10.1145/3520304.3534022

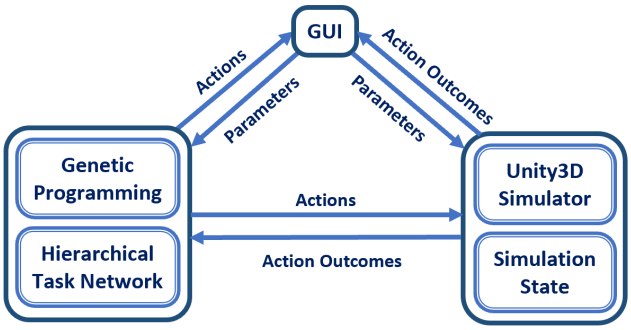

Figure 1: Overview graphic for ABL-Unity3D.

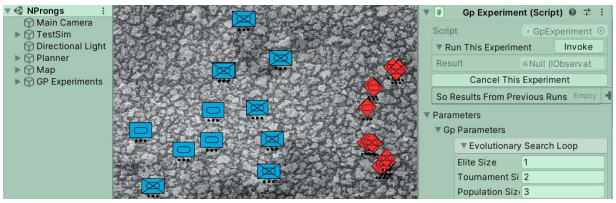

Figure 2: A sample view of a simulation and GP GUI.

and modify it. It also makes it easy to run, organize and design repeatable experiments, and save results.

2 ABL-UNITY3D

ABL-Unity3D is designed to be efficient and user-friendly, see Figure 2. For example, conducting a GP experiment requires no coding. It only requires setting parameters in GUI fields, such as selecting GP primitives and a fitness function. Though, creating new GP primitives and fitness functions requires coding.

2.1 Software Design

The primary design objectives for ABL-Unity3D are: (1) speed, via caching, fast copying and multithreading; (2) extendability, via object oriented design and decoupling of the AI and simulation components; and (3) usability, via a GUI.

Our simulator maintains and gives access to a world state that describes properties of the simulation. The simulation world state can contain two object types. One is an agent within the world state, of which there are two types: `SimUnit` and `SimGroup`. `SimUnit` is a single agent within the world state, and a `SimGroup` is a group of `SimUnit` s. The other object type is actions performed by agents. These are called `SimAction` s.

Generally, the simulation world state, and objects contained in it, are object oriented. However, simulation speed is priority. Thus, ABL-Unity3D uses abstractions to allow for more efficiency. E.g, ABL-Unity3D implements an abstraction similar to Model-View-Controller (MVC) with `SimUnit` and `SimUnitFollower`. This allows for faster copying of the simulation world state. `SimUnit` is similar to the model in MVC and `SimUnitFollower` is like the View. We do not make a strict separation for the Controller part of MVC, so both classes implement controller like behavior.

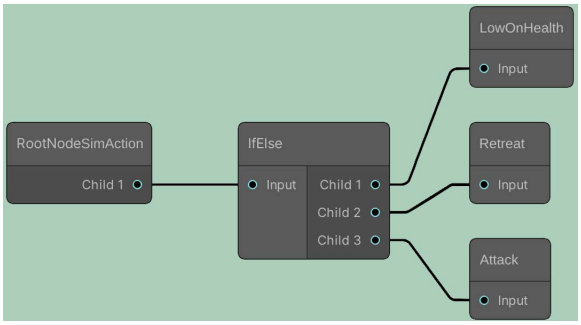

Figure 3: Visualization of a GP solution that represents a conditional statement which evaluates to a `SimAction` in ABL-Unity3D.

ABL-Unity3D improves performance by running AI Planning, GP, agent pathfinding, and the simulation in separate parallel processing threads. In addition, when running multiple GP experiments at once, each experiment is run in a separate thread. This also allows for these processes to be cancelled independently. The ABL-Unity3D GUI is object oriented, and exposes public properties to the user through the GUI. ABL-Unity3D makes it easy to swap out the simulation. The simulation world state is decoupled from the GP and the AI Planning. The only aspects of the GP and AI Planning that depend on the simulation world state are user-defined GP primitives, fitness functions, and AI Planning methods.

2.2 Simulator

ABL-Unity3D uses the Unity3D game engine. It is widely used, and provides a large and well documented API for designing simulations and GUIs. Unity3D has a large community-driven base which provides third-party functionality which is easily integrated using the Unity Asset Store. ABL-Unity3D uses a third-party A* agent pathfinding library [2]. To improve performance for pathfinding, ABL-Unity3D caches previously queried paths from different locations in the terrain.

2.3 GP

The key innovations of GP in ABL-Unity3D are the GP solution representation and fitness function. These are designed to achieve a fast, extendable and user-friendly GP framework.

2.3.1 GP Representations. ABL-Unity3D uses a standard strongly-typed GP implementation with the option for ramped initialization and the genetic operators crossover, and mutation. The algorithm for initialization can be modified. ABL-Unity3D represents GP solutions as tree data structures [7]. Each GP node in the tree has an evaluation type. When a GP node is evaluated, it returns that type. Figure 3 shows a visualization from ABL-Unity3D of a GP primitive written to represent a conditional statement. This implementation of a conditional statement evaluates to a `SimAction`.

The root node of the tree represents the evaluation type of the tree. The root node must have one child node with the same evaluation type of the root node. For example, the conditional statement in Figure 3 evaluates to a `SimAction`. This is shown by the root

```
1  public class Conditional : ExecutableNode<SimAction> {
2      public ExecutableNode<bool> Cond =>
           (ExecutableNode<bool>) this.children[0];
3      public ExecutableNode<SimAction> TrueBranch =>
           (ExecutableNode<SimAction>) this.children[1];
4      public ExecutableNode<SimAction> FalseBranch =>
           (ExecutableNode<SimAction>) this.children[2];
5
6      public Conditional(ExecutableNode<bool> cond,
           ExecutableNode<SimAction> trueBranch,
           ExecutableNode<SimAction> falseBranch) :
           base(cond, trueBranch, falseBranch) { }
7
8      public override SimAction Execute(GpFieldsWrapper
           gpFieldsWrapper) {
9          return Cond.Execute(gpFieldsWrapper) ?
               TrueBranch.Execute(gpFieldsWrapper) :
               FalseBranch.Execute(gpFieldsWrapper);
10     }
11 }
```

Figure 4: Sample code to define a GP primitive for a conditional statement which evaluates to a `SimAction`.

node type "RootNodeSimAction". The other nodes can have any number of children nodes and can be named anything.

The class `ExecutableNode<T>` (line 1, Figure 4) is the superclass for all GP primitives. `T` is the evaluation type of the GP primitive. `ExecutableNode<T>.children` is a list of child nodes for a GP primitive. Child nodes must extend `ExecutableNode<T>`. A GP primitive must define a constructor which passes all child nodes to be evolved to the primitives base constructor. A GP primitive must also override the function `Execute`, which returns the result of the GP primitive upon evaluation. The return type of `Execute` must be the same as the evaluation type `T` for the GP primitive.

Figure 4 shows sample code for defining the conditional statement GP primitive in Figure 3. The GP primitive `Conditional` has an evaluation type of `SimAction` (line 1, Figure 4). Lines 2-4 are helper properties used for visualization. Lines 6-9 define the constructor for `Conditional` class. Lastly, lines 11-12 define the `Execute` method. Line 12 first evaluates the child node which represents the condition for the conditional. If true, return the evaluation of the child node `TrueBranch`. Otherwise, return the evaluation of the child node `FalseBranch`.

Child nodes of a GP primitive can be immutable; they can not be modified by genetic operators. For example, in replace line 5 with a hard coded instance of a subclass of `ExecutableNode<T>` in Figure 4. E.g. an instance of the GP primitive "Attack".

2.3.2 Fitness Function. To define a fitness function, define a class which extends `FitnessFunction`. This sub-class must define a constructor and override the method `GetFitnessOfIndividual`. `GetFitnessOfIndividual` returns the fitness of a GP individual.

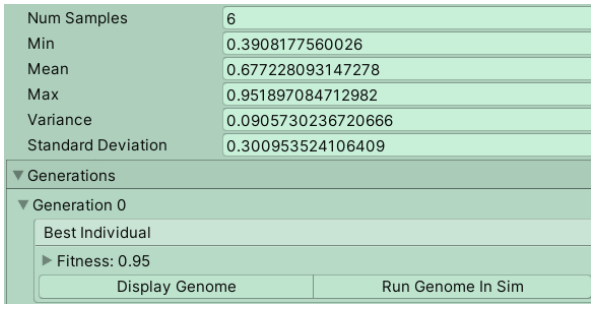

Figure 5: Part of the UI for viewing the results of a GP experiment.

Figure 6: UI for viewing attributes of `SimUnit`s, such as location, health, etc.

2.4 AI Planning

ABL-Unity3D implements an HTN that generates all possible plans for agent behavior, as opposed to finding a single plan that satisfies the given goal method. To do this, The planner decomposes a goal method into a set of sub-methods which achieve the given goal. The methods chosen are then implemented using concrete `SimAction`s, which are then executed in the simulation. It is possible to use the AI Planner in combination with GP in ABL-Unity3D. ABL-Unity3D provides a feature to visualize a plan that an agent is executing, see Figure 8. ABL-Unity3D also provides a feature to visualize a plan using the same graph view interface as Figure 3.

2.5 GUI

The ABL-Unity3D GUI facilitates interaction with the simulation and AI, with the goal to be user-friendly. Its capabilities include:

- Defining simulation parameters;
- Running GP experiments and AI Planning experiments. Figure 5 shows the UI for examining the result of a run of a GP experiment. Figure 7 shows the UI for defining GP experiment parameters;
- Examination of the simulation world state, as well as results generated from experiments. Figure 6 shows UI which can be used to examine agent attributes.

3 ABL-UNITY3D EXAMPLE USE CASE

One use case for ABL-Unity3D is for generating strategic plans. Consider the following scenario: "Blue" team is defending a position, and "Red" team is attempting to take over that position. We want to preserve"Red" team health while reducing "Blue" team health.

One approach is to split the "Red" team into 3 groups. These 3 groups can then attack the position from 3 different angles. In particular, each group can attack up one waypoint along its path

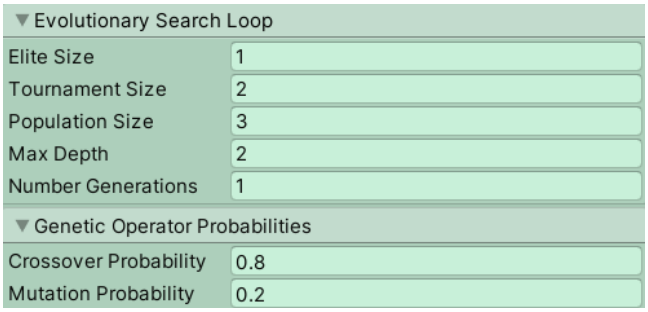

Figure 7: Part of the UI to define and run a GP experiment, and view and save results.

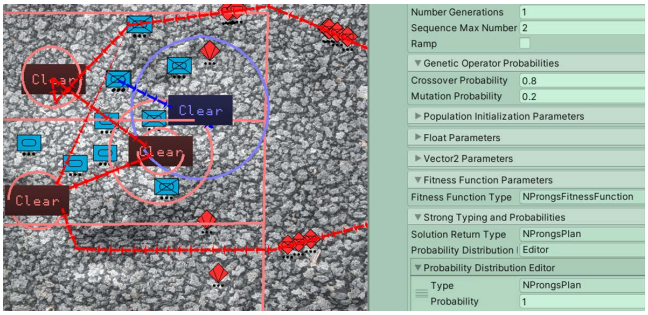

Figure 8: UI for the 3 prongs scenario.

to the position. We call these waypoints "prongs". B is the set of blue team agents, and R is the set of red team agents. Let C be the prongs chosen, and P be the set of prongs to choose from, $C \subseteq P$. There can be more than 3 prongs in P, but there must be at least 3 prongs in P, $|P| \geq 3$. There must at least be one prong in C, but there can be no more than 3 prongs, $1 \leq |C| \leq 3$. Altogether, we can say $1 \leq |C| \leq 3 \leq |P|$. We also must assign a mapping of red team agents to prongs. Let us call this mapping $M : B \rightarrow C$. We call this the 3 prongs scenario. Our goal is to find an optimal C and M to maximize $f(B, R)$ given B, R, and P. In other words, to maximize the total "Red" team health, and minimize the total "Blue" team health, we must find the best prongs to choose and the best assignment of red team groups to those chosen prongs.

ABL-Unity3D can generate solutions to this problem using GP, AI Planning or a combination. Arguably, when $|P|$ is small, AI Planning is the most effective to solve this because it can feasibly exhaust the search space. Though, if $|P|$ is large, the search space will increase in size exponentially. Exhausting the search space through AI Planning may take too long. GP can be used to reduce the search space size by deriving C. This will not always generate the optimal C, but given some constraints, it may be sufficient. We can then use AI Planning to determine what the optimal M is given C. ABL-Unity3D scores and sorts results according to $f(B, R)$.

We benchmark ABL-Unity3D speed by running the three prongs scenario with the AI Planner, and with the AI Planner and GP in combination. We measure mean speed in seconds with and without pathfinding caching, multithreading, and both. The HTN1 and GP+HTN2 rows cannot be compared because they have different HTN parameters. With both optimizations, the AI Planner receives a 25x speed up, and the GP and AI Planner combination receives a 3x speed up, see Table 1. Each test was ran 6 times on a PC with an i7 8700k CPU.

Setting	Base	Cache	Multithread	Both
HTN1	401.46	120.96 (x3)	18.23 (x22)	15.68 (x25)
GP+HTN2	30.85	27.71 (x1.1)	11.59 (x2.7)	10.36 (x3)

Table 1: Average speed (second) and speedup for the HTN (AI Planner) and GP without (base) and with pathfinding caching, multithreading, and both. Note: the HTN1 and GP+HTN2 rows cannot be compared because they have different HTN parameters.

4 DISCUSSION & FUTURE WORK

We also tested GP in ABL-Unity3D with an experiment where GP determines the path for an agent to approach another. This involved lower-level GP primitives involving vector and floating point manipulation. This showed that GP worked, but reinforced that GP is not effective for certain problems.

We plan to implement more features. For example, the AI planner generates plans before running the simulation, but the planner cannot react on-the-fly to unexpected enemy behavior. To improve this, we will implement replanning [3]. Other examples are the ability to generate evolution statistic graphics for GP experiments, and adding a GUI for defining new primitives to reduce coding for domain-specific GP experiments. Finally, we plan to conduct human interaction tests to determine how user-friendly ABL-Unity3D is.

ACKNOWLEDGMENT

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. 2017. Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review* 24 (2017), 13–33.
[2] Aron Granberg. [n.d.]. *A* Pathfinding Project.* https://www.arongranberg.com/astar/
[3] Ilche Georgievski and Marco Aiello. 2014. An Overview of Hierarchical Task Network Planning. https://doi.org/10.48550/ARXIV.1403.7426
[4] Robert Gold, Andrew Haydn Grant, Erik Hemberg, Chathika Gunaratne, and Una-May O'Reilly. 2022. *ABL-Unity3D.* https://github.com/ALFA-group/ABL-Unity3D
[5] Chathika Gunaratne, Rene Reyes, Erik Hemberg, and Una-May O'Reilly. 2022. Evaluating efficacy of indoor non-pharmaceutical interventions against COVID-19 outbreaks with a coupled spatial-SIR agent-based simulation framework. *Scientific reports* 12, 1 (2022), 1–11.
[6] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. 2018. Unity: A General Platform for Intelligent Agents. https://doi.org/10.48550/ARXIV.1809.02627
[7] David J. Montana. 1995. Strongly Typed Genetic Programming. *Evolutionary Computation* 3, 2 (Summer 1995), 199–230. http://vishnu.bbn.com/papers/stgp.pdf
[8] Una-May O'Reilly, Jamal Toutouh, Marcos Pertierra, Daniel Prado Sanchez, Dennis Garcia, Anthony Erb Luogo, Jonathan Kelly, and Erik Hemberg. 2020. Adversarial genetic programming for cyber security: A rising application domain where GP matters. *Genetic Programming and Evolvable Machines* 21, 1 (2020), 219–250.
[9] Unity Technologies. 2020. *Unity3D.* https://unity3d.com