Genetic Algorithm Development Platform

R. Nicholas Vandemark
University of Maryland
ENPM690 - Robot Learning, Project
nickvand@umd.edu

***Abstract*—Research on generalization of genetic algorithms pertaining to robot learning is discussed. Algorithms utilizing these generalizations were designed, and a platform implementing these algorithms was developed to autonomously train the agents (agnostic to robot structure and its objective) in simulation via a simulation suite.**

***Index Terms*—Genetic algorithms, robot learning, Gazebo, ROS2 control**

I. Introduction

A genetic algorithm (GA) is an optimization technique that is inspired by Darwinian evolution. Iterations of the algorithm test the performance of each "chromosome" (a possible solution to the function which is to be optimized) in the "population" (a list of these solutions), which is initially randomly generated. After evaluating the performance of each chromosome, some which had the best performances "reproduce" to create new chromosomes, which replace those which had the worst performances in the population. This emulates survival of the fittest in nature. Each time the population goes through this reproduction cycle, these chromosomes are given a chance to mutate by some small amount because these slight differences could produce even better performance, which emulates random biological mutation that persists. This procedure converges when a chromosome performs well enough (according to desired training criteria).

The objective for this project was to create a platform that can be used to simultaneously develop/tune robot controllers that learn their behaviors from GA's via simulation and then seamlessly use this tuned controller on any medium (simulated, hardware, etc.). Furthermore, the core of this platform should be able to be used for many different types of robots and objectives (i.e., it should be agnostic to the structure of a robot and the behavior it is learning), so that it can be used for more than one specific scenario. The core functionality should have an emphasis on generalizing the pipeline for training this type of controller, such that training new agents and/or behaviors is as lightweight in development as possible, but that the platform remains *entirely* agnostic to the structure and interfaces of robotic agents. Examples would be motion controllers for serial, parallel, or differential drive robots with any types of completion criteria.

II. Methodology

A. Technologies

This platform is implemented on top of the *ROS2* software stack, and the chosen simulation suite was Gazebo. These choices were made with many considerations in mind.

The *ros2_control* packages [2] offer interfaces to robot control theory on top of the *ROS2* software stack, with a strong emphasis on agnosticism to the system that the controllers are being developed towards in an effort to support as many configurations as possible. Similar to how connections between publishers and subscribers in a ROS2 network are "named", the interfaces of the controllers owned by a *ROS2 control*'s *controller manager* node are also named. This allows for dynamically configurable connections between the robot's hardware and its controller(s), but unlike traditional publishers/subscribers, these connections only communicate decimal values, whose values make sense with the context of the control problem (these could be joint positions, joint velocities, battery charge percentages, etc.). Along with the tools in these packages are implementations of commonly used controllers and hardware interfaces. *ros2_control* was therefore chosen for its significant support for dynamic configurability and agnosticism to types of control, which is important in a system which aims to be as agnostic to the robot's structure/hardware as possible.

Gazebo was chosen for its compatibility with *ROS2*, but also primarily because of a plugin offered by the *gazebo_ros2_control* package [3] which allows for the utilities in the *ros2_control* packages to easily interface with the functionality of Gazebo. This plugin allows the developer to easily bind the telemetry of the simulated robotic agent to the state interfaces managed by a *controller manager* node, and also bears the burden of the controller manager node's lifecycle. Because Gazebo can be configured to use many different types of physics engines with different configuration parameters, this means that the controller's GA is being trained on as accurate a simulation as desired.

B. Design

The implementation of this platform consists of four main packages [1] (where the *ep* prefix simply stands for "ENPM690 project"):

- *ep_common*: This contains node interfaces / utilities used in both runtime and training configurations. Most notably, this includes the command interface controller interface, which is responsible for directly forwarding the outputs of a genetic algorithm's fitness function to the command interfaces requested in the training campaign.

- *ep_common_interfaces*: This contains basic ROS2 messages, services, etc used in both runtime and training configurations.
- *ep_training*: This contains node interfaces / utilities used only in training configurations. This includes many of the nodes used in the training configuration, such as the genetic algorithm controller interface (the master node of this configuration), the state observer interface, the fitness evaluator interface and an extension of it which depends on time (it subscribes to the clock of the simulation for ease of use), and the campaign orchestrator GUI.
- *ep_training_interfaces*: This contains ROS2 messages, services, etc used only in training configurations. This includes the custom action for executing training campaigns.

Figures 1 and 2 show the *ROS2* nodes needed for the runtime and training configurations of the system, respectively. Each node label that has the suffix "1..N" could theoretically have multiple instances, so the training campaign could be parallelized. The topics used by these nodes are also indexed according to the "i" suffix of those names.

Figure 1 shows that the runtime configuration uses a Gazebo instance, but this could also be a hardware robot, some other simulated version of it, etc. The only other node is the robot controller, which uses the fitness function of the GA with the requested configuration parameters (learned through previous training) to control the behavior of the robot.

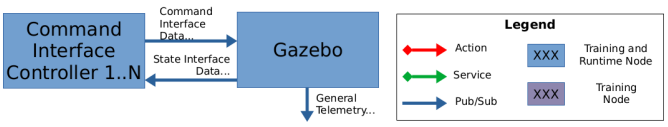

Fig. 1. The configuration of the system on runtime.

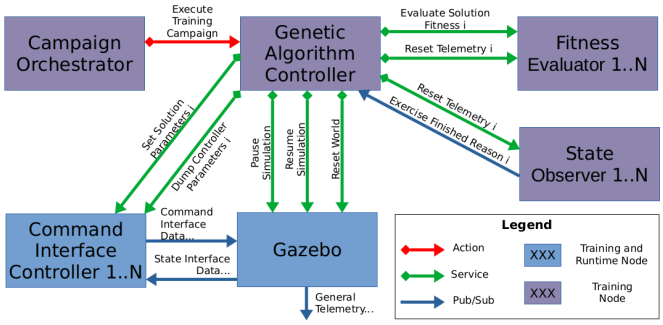

Fig. 2. The configuration of the system while training.

It can be seen that the architecture of Figure 2 is a superset of that in Figure 1, and this is because each iteration of the genetic algorithm controller node's main routine essentially runs a short-lived instance of the runtime configuration with different configuration parameters while other nodes observe its behavior during the simulation. These additional nodes can be described as the following:

- *Genetic Algorithm Controller*: This is essentially the master node of the training configuration. It implements the generalized genetic algorithm, including creating the initial randomly-generated population, the crossover routine, and the mutation routine. Again, this is agnostic to the structure of the robot and the behavior being trained, up until the exact datatype of the solution. As the training campaign runs, this node will take a chromosome out of the population, convert it to ROS2 parameters, set the configuration of the robot controller with these parameters, simulate its performance by manipulating the state of the simulation, and collect its quantified performance. This overall procedure is described in Algorithm 1, and the higher-level procedures referenced therein are broken out into Algorithms 2 to 5. The procedures in Algorithm 3 are implemented specific to the structure of a chromosome. While they must be implemented for each new datatype, they can be used to train any number of behaviors for a system that utilizes this datatype.
- *State Observer*: Once simulation starts, this node is responsible for detecting when an individual solution's completion criteria has been met by listening to the robot's telemetry. Once met, it informs the genetic algorithm controller which criteria was met. By default, this node will observe when the timeout has been met/exceeded, but observing other criteria is implemented for each desired behavior.
- *Fitness Evaluator*: This node performs any desired metrics on telemetry output by the robot over the course of the simulation to quantify the performance of a solution. This result is used by the genetic algorithm controller to compare the performances of the chromosomes in a population. Notably, this node does not have an implementation of the fitness function (only the robot controller does). This implies that the quantifiable performance of a chromosome is not the direct output of the fitness function, but how it manifests behavior in the simulated environment.
- *Campaign Orchestrator*: This node provides a user interface for executing training campaigns via the genetic algorithm controller. It provides user inputs for each of the hyperparameters in a custom ROS2 action (many of the inputs to the *ExecuteTrainingCampaign* procedure of Algorithm 1), and displays (read-only) feedback as the campaign proceeds. A screenshot of this user interface can be seen in Figure 3.

III. Results

Development for the platform made significant progress, but there are a number of technical issues that prevented completion of the platform in the limited timeframe. The way that this platform attempts to use the *ros2_control* and *gazebo_ros2_control* packages seems to be a use case that was not originally supported by them, so extra work will have to go into updating those third-party packages to cope with what this application is intending for.

Algorithm 1 Genetic algorithm controller training routine

```
1: procedure EXECUTETRAININGCAMPAIGN(gen_count, gen_soln_count, soln_time_limit, conv_th, num_offspring, mutability, cmnd_if, state_if)
2:     gen_number ← 0
3:     pop ← Null                          ▷ The population of solutions
4:     PauseSimulation()
5:     loop
6:         if gen_number = 0 then
7:             pop ← InitPopulation(gen_soln_count)
8:         else
9:             pop ← Select(pop)
10:            pop ← Crossover(pop, num_offspring)
11:            pop ← Mutate(pop, mutability)
12:        for all soln in pop do
13:            params ← GetParams(soln, cmnd_if, state_if)
14:            SetControllerParams(params)
15:            ResetSimulation()
16:            RestartStateObserver(soln_time_limit)
17:            RestartFitnessEvaluator()
18:            PlaySimulation()
19:            WaitForSimulatedSolnCompletionCriteria()
20:            PauseSimulation()
21:            soln_fitness ← EvaluateSolnPerformance()
22:            if CompareFitness(soln_fitness, conv_th) then
23:                return gen_number, soln, soln_fitness
24:        increment gen_number
25:        if gen_number = gen_count then
26:            return Null
```

A. Current Design

The base design for all of the types of nodes seen in Figure 2 were completed, with their implementations in the *ep_common* and *ep_training* packages.

An example package *ep_gantry_velocity_control_demo* was being developed, which is a scenario which uses this platform to train the K_p, K_i, and K_d parameters of a one degree of freedom gantry's PID velocity controller. Nodes which implement the *robot / command interface controller*, *state observer*, and *fitness evaluator* interfaces were created for this robotic agent and the desired behavior (maintain a given trapezoidal velocity profile). A launch file was created to properly instantiate all of the controllers and nodes, start Gazebo, and set the node parameters.

However, as mentioned above, there were technical issues which prevented the completion of the core platform, and therefore this example which demonstrates its use as well. The README.md file [1] on this repository's development branch describes how the system can be started, but will fail to proceed.

B. Future Development

There is a critical issue implementing the core of this platform's design, which after initial investigation seems to

Algorithm 2 Genetic algorithm controller secondary routines

```
1: procedure INITPOPULATION(gen_soln_count)
2:     pop ← Empty List
3:     for i = 0 to gen_soln_count do
4:         rand_soln ← GetRandomSoln()
5:         append rand_soln to pop
6:     return pop
7: procedure SELECT(pop)
8:     pop_sorted ← pop sorted via CompareFitness
9:     return pop_sorted
10: procedure CROSSOVER(pop, num_offspring)
11:    num_solns ← size(pop)
12:    for i = 0 to num_offspring do
13:        parent0 ← i
14:        parent1 ← i + 1
15:        offs0 ← num_solns - parent0
16:        offs1 ← num_solns - parent1
17:        rand_crossover_point ← Random(0.0, 1.0)
18:        pop[offs0], pop[offs1] ← DoCrossoverAt(
               pop[parent0],
               pop[parent1],
               rand_crossover_point)
19:    return pop
20: procedure MUTATE(pop, mutability)
21:    for i = 0 to size(pop) do
22:        pop[i] ← MutateSoln(pop[i], mutability)
23:    return pop
24: procedure WAITFORSIMULATEDSOLNCOMPLETION-CRITERIA
25:    finished_reason ← NOT_FINISHED
26:    while finished_reason = NOT_FINISHED do
27:        msg ← SolnCompletionCriteriaCallback()
28:        finished_reason ← msg.value
```

Algorithm 3 Genetic algorithm controller routines that are specific to solution type

```
1: procedure GETRANDOMSOLN ▷ Returns a randomly generated solution.
2: procedure COMPAREFITNESS(f0, f1) ▷ Returns true if f0 is a better fitness than f1, false otherwise.
3: procedure DOCROSSOVERAT(soln0, soln1, crossover) ▷ Returns the two offspring of the two given chromosomes at the given crossover [0.0,1.0].
4: procedure MUTATESOLUTION(soln, mutability) ▷ Returns the given solution randomly mutated within the boundaries of the given mutability.
5: procedure GETPARAMS(soln, cmnd_if, state_if) ▷ Returns the set of parameters to fully define the robot control parameters, consisting of at least the given robot command and state interface names.
```

Algorithm 4 Genetic algorithm controller ROS2 subscription callbacks

1: **procedure** SOLNCOMPLETIONCRITERIACALLBACK ▷ ep_training_interfaces/ExercisingSolutionFinishedReason

Algorithm 5 Genetic algorithm controller ROS2 service calls

1: **procedure** PAUSESIMULATION ▷ std_srvs/Empty
2: **procedure** RESETSIMULATION ▷ std_srvs/Empty
3: **procedure** PLAYSIMULATION ▷ std_srvs/Empty
4: **procedure** SETCONTROLLERPARAMS ▷ rcl_interfaces/SetParameters
5: **procedure** RESTARTSTATEOBSERVER ▷ ep_common_interfaces/SetDouble
6: **procedure** RESTARTFITNESSEVALUATOR ▷ std_srvs/Empty
7: **procedure** EVALUATESOLNPERFORMANCE ▷ ep_common_interfaces/GetDouble

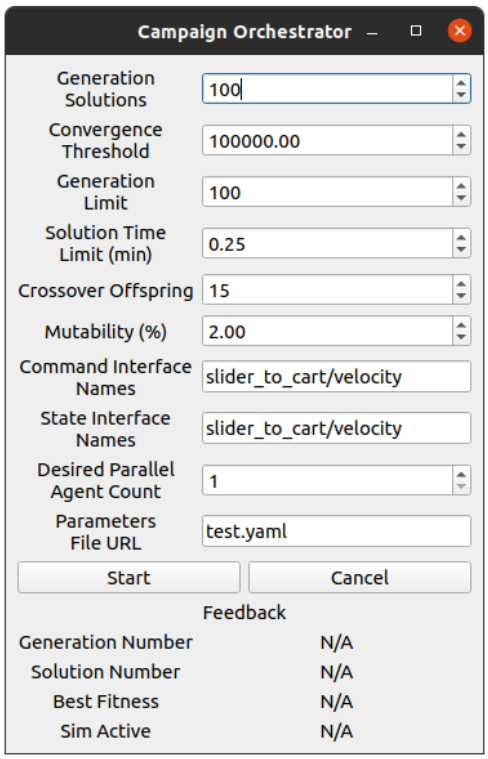

Fig. 3. An example of the user interface used to request training campaigns with the platform.

have to do with how the *ROS2* controller manager node and the node which wraps around Gazebo are executed. Manipulating the state of the physics simulation (pausing, resuming, and resetting) are tricky, but resetting the world and/or simulation somehow disconnects the controllers from the simulated telemetry. Furthermore, this setup does not allow for the controllers under the *controller manager* node to handle callbacks to subscriptions, service calls, etc. while the simulation is paused. The executor does not update if the simulation is paused, which is just fine for the Gazebo node wrapper, but this also prevents the *controller manager* node from updating. This means that while the simulation is paused, the parameters for the robot controller that are a function of the GA cannot be updated. This makes continuous simulation of different solutions/chromosomes impossible at the moment, until this is resolved. Doing so is top priority for future development.

Less critical but still important is the ability to parallelize solutions while in the training configuration, as this would speed up training by a factor directly proportional to the number of parallel agents. This would require support in the genetic algorithm controller and, if the collision bitmask offered in previous versions of Gazebo is still supported, automated generation of this stanza in the robots' URDFs. A nice-to-have would be a visual cue as to which is which, such as color-coding of the entire model.

IV. DISCUSSION

The work-in-progress platform is a professionally developed set of packages with a strong potential to accomplish its goal. Once finished, it could be a powerful tool to assist with semi-automated robot learning for any agent that has support for *ROS2 control* and Gazebo and whose behavior can be reinforced with the hyperparameters offered by the campaign orchestrator's user interface.

This provides a solid foundation for generalizing the behavior of GA's, but perhaps this could be built upon more to further generalize the behavior of the larger class of evolutionary algorithms, and the genetic algorithm controller can be broken out into different tiers to cater to different types of evolutionary algorithms. Much of the core behavior that was difficult to organize and implement (coordinating simulation state, packaging and communicating chromosome and robot interface data as ROS2 parameters, standardizing passive telemetry observation, etc.) would remain the same, it would only be the structure around the *InitPopulation*, *Select*, *Crossover*, and *Mutate* routines in Algorithm 1 that would have to be updated to be more flexible. Not only would this be a useful addition to the platform, but learning how to generalize routines and finding commonality between different ones is a powerful learning tool.

The aforementioned technical issues that are a function of the chosen technologies could be further explored, and perhaps those technologies could be removed. For example, perhaps the plugin offered by the *gazebo_ros2_control* could be removed in favor of manually setting up instances of the

controller manager node and configuring the connections to use the telemetry coming out of the Gazebo simulation. This could have the side-effect of easier transition from simulated and physical robotic agents, as this package would have more control over the *ROS2* connections that for now are left to this third-party plugin.

V. Conclusion

Considerations were made on how to generalize the concepts of a GA, and then algorithms that are to a degree agnostic to the structure of the robotic agent and the behavior being learned were designed. A platform which implements these in a training configuration was designed on top of *ROS2* and with *Gazebo*, and this same platform could also be used to operate the robot controller with the learned behavior in a runtime configuration. A user interface was developed to execute these training campaigns.

There was significant progress on development for this platform, and a working solution would be ready with some more development to cope with technical issues. Learning not only how to develop genetic algorithms but generalizing their structure with wider applicability was extremely beneficial.

VI. Acknowledgments

This project was done individually, so all contributions belong to R. Nicholas Vandemark.

VII. List of Acronyms

GA genetic algorithm 1

References

[1] R. Nicholas Vandemark, https://github.com/rnvandemark/enpm690_project, 2023.
[2] ros2_control Maintainers, https://github.com/ros-controls/ros2_control/tree/galactic, 2023.
[3] gazebo_ros2_control Maintainers, https://github.com/ros-controls/gazebo_ros2_control/tree/galactic, 2023.