EFFICIENTLY EVOLVING SWARM BEHAVIORS USING GRAMMATICAL EVOLUTION WITH PPA-STYLE BEHAVIOR TREES

Aadesh Neupane & Michael A. Goodrich
Department of Computer Science
Brigham Young University
Provo, UT 84602, USA
adeshnpn@byu.edu, mike@cs.byu.edu

ABSTRACT

Evolving swarm behaviors with artificial agents is computationally expensive and challenging. Because reward structures are often sparse in swarm problems, only a few simulations among hundreds evolve successful swarm behaviors. Additionally, swarm evolutionary algorithms typically rely on ad hoc fitness structures, and novel fitness functions need to be designed for each swarm task. This paper evolves swarm behaviors by systematically combining Postcondition-Precondition-Action (PPA) canonical Behavior Trees (BT) with a Grammatical Evolution. The PPA structure replaces ad hoc reward structures with systematic postcondition checks, which allows a common grammar to learn solutions to different tasks using only environmental cues and BT feedback. The static performance of learned behaviors is poor because no agent learns all necessary subtasks, but performance while evolving is excellent because agents can quickly change behaviors in new contexts. The evolving algorithm succeeded in 75% of learning trials for both foraging and nest maintenance tasks, an eight-fold improvement over prior work.

1 INTRODUCTION

Bio-inspired models have produced efficient algorithms in various domains (Karaboga & Basturk, 2007; Dorigo et al., 2006). The potential benefits of bio-inspired algorithms are limited by the cumbersome task of observing animal behavior and creating mathematical models to describe both individual and collective behaviors (Sumpter & Pratt, 2003; Gordon, 2010). A promising alternative is to design novel swarm behaviors using evolutionary algorithms.

Evolving swarm behaviors in non-episodic setting requires group-level objectives and rewards to be divided into individual objectives and rewards, which is a form of the *credit assignment problem* (CAP) (Sutton, 1984). Existing swarm evolution algorithms address CAP by designing ad hoc fitness functions that give the artificial agents sufficient feedback to solve a particular task (Ferrante et al., 2013; Neupane et al., 2018). Unfortunately, designing an ad hoc fitness function for each task requires expert knowledge, and is subject to human biases (Nelson et al., 2009). Also, evolving swarm behaviors require the designer to choose and aggregate various controllers, evolutionary algorithms, and fitness functions. In many cases, only a few combinations are viable to evolve collective behaviors, and the success rate is low among those viable combinations.

A multi-agent Grammatical Evolution (GE) (O'Neill & Ryan, 2001) algorithm called GEESE evolved colony-level foraging behaviors, outperforming conventional GE and hand-coded solutions on a foraging task (Neupane et al., 2018). GEESE's controllers were Finite State Machines (FSMs) and the group fitness was the total food collected, but GEESE could only evolve foraging behaviors. GEESE-BT (Neupane & Goodrich, 2019b; Neupane, 2019) used a more expressive grammar that could evolve both foraging and cooperative transport behaviors. GEESE-BT's controllers were behavior trees (BTs), and GEESE-BT used ad hoc fitness functions from the theory of intrinsic moti-

vators. Unfortunately, the ad hoc reward structures were not sufficient to evolve successful behaviors with high success rate and required a new fitness function for each new task.

This paper presents the BeTr -GEESE ("better geese") algorithm, which produces successful swarm behaviors while learning even though no single agent has all necessary behaviors at any given point during the evolution. BeTr -GEESE improves GEESE-BT in three significant ways: (i) primitive behaviors use Postcondition-Precondition-Action (PPA) structures (Sprague et al., 2018); (ii) the grammar used PPA-style BT programs; and (iii) the task-specific ad hoc rewards were replaced with BT execution node status, enabling direct feedback from PPA nodes. The evolving algorithm succeeds in 75% of learning trials for two tasks (foraging and nest maintenance), an eight-fold improvement over GEESE-BT.

Two limitations are noteworthy. First, swarm success requires agents to perform ongoing evolution. This is problematic if agents experience catastrophic failures while learning, such as when a physical robot crashes in the world. Second, the algorithm is only applied to multiagent tasks that are *divisible* and *additive* tasks, meaning that the swarm's task can be broken into subtasks achievable by individual programs (Steiner, 1972).

2 Related Work

Early work in evolving robot controllers include (a) staged evolution of a complex motor pattern generator to control a walking robot (Lewis et al., 1992) and (b) evolving neural-network-based control architectures for visually guiding robots (Cli et al., 1993). Neural networks can directly map robot inputs to outputs (Lewis, 1996; Trianni et al., 2003), but tend to be data-hungry and not transparent (Fan et al., 2021). Kriesel et al. (2008) applied evolution to swarm systems and demonstrated that simple evolved individuals can produce effective swarm behaviors.

Evolutionary robotics algorithms differ from each other in the choice of the evolutionary algorithm and the choice of the agent's controller. Individual robot controllers have been designed using Neural Networks (NN), Finite state machines (FSM) and hierarchical FSMs (HFSM) (Petrovic, 2008; Pintér-Bartha et al., 2012; König et al., 2009; Brooks, 1986; Valmari, 1996). Behavior Trees (BT) often provide more readable, scalable, modular, and reactive structures than HFSMs and NNs (Colledanchise & Ögren, 2017).

Duarte et al. (2016) evolved swarm behaviors for physical robots and demonstrated scalability, flexibility, and resilience; results were shown for swarm-level dispersion, homing, clustering, and monitoring. GE and state machine-type controllers have been combined to evolve swarm behaviors (Neupane et al., 2018; Neupane & Goodrich, 2019a; Ferrante et al., 2013). Even though GE exploits prior knowledge in the form of grammars to learn better solutions much faster than conventional genetic programs (O'Neill & Ryan, 2001), results depend on the design of ad hoc reward structures.

The lateral genetic transfer of controllers in BeTr -GEESE is bio-inspired. Prior work claims that complex behaviors rarely evolve solely from crossover and mutation alone but require endosymbiosis or horizontal transfer (Jablonka & Lamb, 2014; Lane, 2015; Quammen, 2018). With the horizontal transfer, evolving single agents with sparse and delayed reward is more computationally efficient than evolving complex controllers (Lee, 1999; Engebråten et al., 2018).

PPA structures have been widely used (Fikes & Nilsson, 1971; Knoblock, 1995). Sprague et al. (2018) used PPA structures to construct a modular, versatile, and robust control architecture for mission-critical autonomous underwater vehicles. The PPA structure ensured that the execution of BT followed goal fulfillment priorities. Colledanchise et al. (2019) introduced a standard backward chaining algorithm to create a PPA structure automatically, and showed that the structure could skip actions that were already executed and only plan when the postconditions were not satisfied. Ögren (2020) proved convergence guarantees for a particular PPA-BT structure, and Parashar et al. (2021) presented a PPA layered strategy to transform a mission into decomposable tasks. BT-based collective behaviors have been used in non-evolutionary-based optimization. Kucking et al. (2018) used BTs to perform foraging and aggregation. Kuckling et al. (2021) extended that work with a set of behavioral modules within a predefined BT structure. The algorithm optimized swarm behaviors for foraging and marker aggregation tasks when agents could communicate.

3 GEESE-BT OVERVIEW

GEESE (Neupane et al., 2018) and GEESE-BT (Neupane & Goodrich, 2019b) are multi-agent grammatical evolution (GE) algorithms that are similar to standard GE in terms of initialization, genetic operators, and genotype-to-phenotype mapping. Each agent is initialized with a fixed-length array of binary strings called the genome. The genome is then grouped into fixed consecutive blocks called a codon. Each codon block can be converted into a codon value. The conversion of the genome to a program with the help of Backus-Naur form (BNF) grammar is the genotype-to-phenotype mapping. The *max-tree-depth* parameter controls the number of unexpanded non-terminals in the current parse tree.

GEESE agents uses three evolution steps: *sense*, *act*, and *update*. During the *sense* phase, agents exchange genome information with any (nearby) agents in the field of view. The willingness to transfer their gene is controlled by the INTERACTION_PROB parameter. During the *act* phase, an agent queries its storage pool to determine whether the pool size exceeds STORAGE_THRESHOLD parameter. Higher thresholds indicate that interactions with more agents are needed before evolving. Like prior GE work O'Neill & Ryan (2001), if the threshold is exceeded, agents apply genetic operations to the gene pool in the order: a) selection, b) crossover, and c) mutation. Selection samples a subset of genotypes based on the fitness value to form a new population. Crossover combines two genomes to produce a new genome. Mutation randomly flips the bits of the genome. During the *update* phase, an agent replaces its current genotype with a new genotype if there is a new genotype with higher fitness. Each evolution/learning time-step, all agents sense, act, and update.

Parameters	GEESE	GEESE-BT
Storage Threshold	NA	7
Interaction Probability	0.8	0.85
Parent-Selection	Tournament	Fitness + Truncation
Elite-size	1	N/A
Mutation Probability	0.01	0.01
Crossover Probability	0.9	0.9
Crossover	variable_onepoint	variable_onepoint
Genome-Selection	Tournament	Diversity
Number of Agents	100	100
Evolution Steps	284	12000
Max Tree Depth	10	10

Table 1: Evolution parameters used by GEESE and GEESE-BT.

GEESE-BT (Neupane & Goodrich, 2019b) replaced the state-machine controllers in GEESE with Behavior Trees (BT). Paraphrasing Neupane & Goodrich (2019b) for context, a BT is a directed rooted tree with internal *control flow nodes* and leaf *execution nodes* (Colledanchise & Ögren, 2018). BT execution starts at the root node by generating *ticks* at a fixed frequency. After each tick, a node returns *running*, meaning that processing is ongoing, *success*, meaning that the node's objective is achieved, or *failure*, meaning neither running nor success. Control nodes include *selectors*, which act as logical `or`, *parallel* and *sequence* nodes, which act as logical `and`, and *condition* nodes, which act as logical `if`.

4 BETR-GEESE

BeTr-GEESE shares the core evolutionary stages (sense, act, and update) and genetic operations as GEESE and GEEST-BT, but has three significant improvements: a) PPA-based primitive behaviors, b) PPA-style BNF grammars, and c) a BT-induced fitness function. Since each of these changes impact swarm performance, each is experimentally evaluated in Sections 4.4-4.6, respectively. BeTr-GEESE uses the same parameters as GEESE-BT (Table 1) to ensure unbiased comparison.

4.1 AGENT LIFECYCLE

The BeTr-GEESE algorithm runs for a particular number of steps, which is defined before the start of the simulation[1]. n agents are initialized in a grid environment as described in Section 4.2, and a random genome of length 100 is created. The genome is transformed to a BT controller by the genotype-to-phenotype mapping process using the swarm BNF grammar described in appendix B. Each agent interacts with the environment using the BT controllers and updates its fitness value using *diversity fitness* and an overall fitness function $\mathbf{A}_t$ described in Section 4.6. (*Diversity fitness* is the total number of unique behaviors nodes divided by the total behaviors defined in the grammar.) Then the agents independently perform the sense, act, and update methods described in Section 3.

4.2 SIMULATION PARAMETERS

A 100x100 grid environment was used with a hub of radius ten at the origin. The population had 100 learning agents. Agents moved with speed of 2 units per time step. Task performance is measured in terms of food or debris moved at the end of 12,000 learning steps. 12,000 learning steps was adopted from GEESE-BT (Neupane & Goodrich, 2019b), and from empirical analysis it was observed that for all three task reported in GEESE-BT obtained best performance when the simulation was run for at least 12,000 steps.

The swarm simulation environment was created using the **Mesa** agent-based modeling framework (Kazil et al., 2020). The simulator lacks the high fidelity physic engine, making it faster to run experiments with a large number of agents but greatly simplifies the robot-environment interactions. BT controllers for the agents were created using Behavior Tree framework **py_trees** (Stonier & Staniaszek, 2021) and **PonyGE2** (Fenton et al., 2017) was used to implement the Grammatical Evolution algorithms GEESE, GEESE-BT, and BeTr-GEESE. All experiments were performed on a machine with an i9 CPU, 64 GB RAM running 16 parallel threads.

4.3 EVALUATION METRICS

BeTr -GEESE is evaluated using two swarm tasks, *Foraging* and *Nest Maintenance*. The *Foraging* task requires agents to retrieve food from a source to a hub. A single foraging site of radius ten with multiple "food" objects is randomly placed at 30 units from the hub. The total food units is set to equal agent population. Task performance is the percentage of food at the hub.

The *Nest Maintenance* task requires agents to move debris near the hub to a place outside a fixed boundary. Multiple "debris" objects are placed within ten units radius of the hub. The desired boundary is set at 30 units radius away from the origin. The total debris units at the hub is equal to the agent population. Task performance is the percentage of debris that is outside the boundary.

The simulation is labelled *learning* when the agents are evolving controllers where the behaviors of the agent changes with time. The simulation is labelled *test* when the agent controllers from a *learning* simulation are transferred to a new simulation environment and the agents controllers remains static and do not evolve. For all the experiments reported in this paper, *test* simulation does not differ drastically but only in the positions of sites and obstacles placed randomly at the runtime. Foraging is deemed *successful* if more than 80% food is collected, and nest maintenance is successful if more than 80% debris is removed. *Success rate* is defined as the ratio of the number of successful trials to the total number of simulations. *Learning efficiency* is defined as foraging or maintenance percentage at the end of a learning simulation.

4.4 ADDING PPA TO PRIMITIVE BEHAVIORS

Figure 1(a) illustrates a standard Postcondition-Precondition-Action (PPA) Behavior Tree (BT), which is defined as a BT where the selector root node ("`?`") ensures that the *action* node on its right branch of the sequence control node ($\rightarrow$) is not carried out if the *postcondition* node is already satisfied Colledanchise & Ögren (2018). Figure 1(c) illustrates how GEESE-BT requires a precondition (*IsCarryable*) to be satisfied `and` the task (*Carry*) successfully performed. The `and` operator is implemented as a sequence BT node ($\rightarrow$). BeTr -GEESE uses the PPA structure in Figure 1(a) for

[1]BeTr-GEESE learning is not episodic like some GE algorithms; there are no terminal states and replay.

implementing all primitive behaviors (PB) described in appendix A. For illustration, *CompositeSingleCarry* PB is shown in Figure 1(b). The root selector node (represented as the "?") checks the postcondition (left branch, *AlreadyCarrying*) and calls the sequence node in the right child only if the postcondition is not met. The right child checks preconditions and implements the action. The postcondition ensures that the planner does not need to re-execute when the goal has been met.

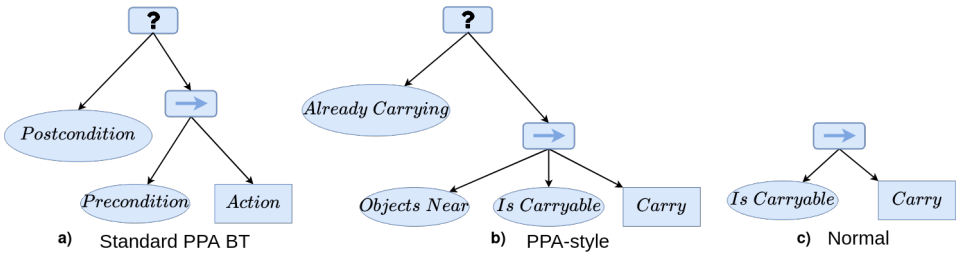
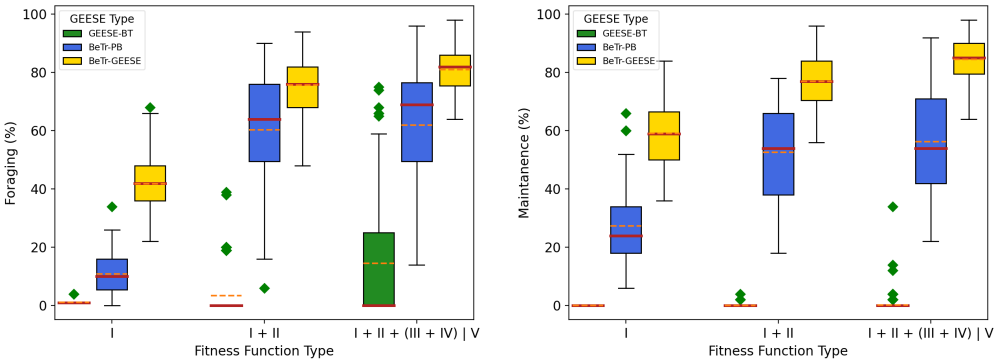


Figure 1: BTs for primitive behaviors. a) General PPA-style BT. *CompositeSingleCarry* primitive behavior represented with b) PPA-style and c) nominal trees.

The green and blue boxes in Figures 2a-2b compare learning efficiency (over a range of fitness function conditions, described below) between the GEESE-BT primitive behaviors (PB) and the PPA-based primitive behaviors (BeTr -PB) for the two tasks. The PPA structure improves performance across all fitness function conditions and for both tasks.



(a) Foraging experiments.

(b) Nest-maintenance experiments.

Figure 2: Learning efficiency, measured by the percentage of food/debris transported to/from the hub with respect to variations in primitive behaviors, grammar, and fitness function. **D**iversity is type (I), **E**xploration is type (II), Pr**o**spective is type (III), **T**ask-specific is type (IV), and **B**T feedback is type (V) fitness function in the x-axis.

4.5 Adding PPA to the Grammar

Appendix B presents the grammar used by BeTr -GEESE, which is modified from the GEEST-BT grammar to use the PPA structure. The blue and yellow boxes in the leftmost and middle groups of Figures 2a-2b both use PPA-based PBs, but the yellow boxes use PPA-based grammars, respectively. Changing the grammar to use PPA structures further improves learning efficiency for both tasks.

4.6 Replacing ad hoc Fitness with BT Status

Essential Fitness Elements. There are various choices of the fitness function in evolutionary robotics based on the type of controllers and amount of priory task knowledge the designer has (Nelson et al., 2009). GE requires a diverse genetic population, and both foraging and nest maintenance require exploring the world. Thus, two fitness elements are required: diversity (type I) and exploration (type II). The phenotype is an agent program learned from the grammar. For

GEESE-BT and BeTr -GEESE, the program is a behavior tree. *Diversity fitness* is denoted by $D : \{\text{Phenotypes}\} \rightarrow \mathbb{R}$, promotes diversity in the genotype, and is defined as the total number of unique behavior nodes in the BT divided by the total possible behaviors defined in the grammar. *Exploration fitness* is denoted by $E : \{\text{Locations}\} \longrightarrow \mathbf{R}$, promotes visiting new locations, and is defined as the number of unique world locations visited by the agent.

Ad hoc Fitness Elements. GEESE-BT uses ad hoc fitness functions. Prospective fitness (type III) rewards "intrinsic" actions like picking up, carrying, or dropping objects. Task-specific fitness (type IV) uses hand-tuned fitness functions designed to reward collective actions: total food at the hub and the total debris away from the hub, respectively.

BT Feedback Fitness. The BT feedback fitness (type V) is denoted by $B : \{\text{BehaviorTreeStatus}\} \longrightarrow \mathbf{R}$ and is defined as the sum of potstcondition, constraint, and selector node rewards. When a postcondition node status is *success*, a subjectively chosen reward of $+1$ indicates that some potentially useful condition in the world holds. A subjectively chosen reward of -2 occurs when a constraint node status is *failure*. A subjectively chosen reward of $+1$ is returned when the root selector node status success, indicating that some sub-task has been accomplished somewhere in the BT. Based on Nelson et al. (2009), BT feedback (type V) incorporates a low level of prior knowledge, whereas ad hoc fitness (type III & type IV) incorporates high or excessive prior knowledge. BT feedback is superior to ad hoc fitness because it requires less prior knowledge, is general for all swarm tasks, and remains the same while the task changes.

Blending Fitness Types. GEESE-BT blends types I–IV to produce overall fitness. Since types III and IV are ad hoc, overall fitness is ad hoc. By contrast, BeTr -GEESE rewards behaviors that promote genetic diversity, promote world exploration, observe or accomplish subtasks, or avoid constraint violations. Let $\mathbf{A}_t$ denote an agent's fitness, defined as the exponential blend $\mathbf{A}_t = \beta(\mathbf{A}_{t-1}) + (E_t + B_t)$, with β empirically set to 0.9 and $A_0 = D$. Historical blending overcomes the temporal sparsity of diversity, exploration, and BT status.

Fitness Results. The leftmost and middle group in Figures 2a-2b use only diversity or diversity plus exploration fitness. Performance improves with exploration fitness. The rightmost group in the figures uses diversity, exploration, and the ad hoc fitness for GEESE-BT (green) and for GEESE with PPA structures (blue). The yellow box in the rightmost group uses diversity and exploration, and replaces the ad hoc and hand-tuned rewards with BT status. The rightmost group shows that replacing ad hoc fitness with BT status yields substantial improvement. Out of 64 simulation runs, BeTr-GEESE succeed 75% of the time, whereas GEESE-BT succeeded only 9.3%.

5 Performance of Fixed Programs

BeTr -GEESE agents succeed while evolving, but they perform poorly once evolution stops (fixed agents). This section presents results for various mixtures of 100 fixed agents.

5.1 Homogeneous Populations of Best-Performing Agents

Create a homogeneous population by selecting the fittest agent at the end of the evolution and forming a population with 100 copies of that agent. Homogeneous populations of evolved BeTr -GEESE and GEESE-BT agents fail on both tasks. The performance of homogeneous populations is poor because of the types of programs evolved. The foraging task requires at least four PPA sub-trees: explore the environment and find the site, carry the food, bring the food back to the hub, and drop the food at the hub. Among 100 independent evolution experiments, 3235 different BeTr -GEESE programs were evolved, 97.8% had just one PPA sub-tree, and 2% had two PPA sub-trees. These static programs are not capable of solving the problem by themselves. By contrast, GEESE-BT more frequently produced agents with all four necessary subtrees but still failed, indicating heterogeneous controllers are needed.

5.2 Heterogeneous Populations of High-Performing Agents

The fitness of an individual agent can be deceiving because the agent might be fit only when other agents in a heterogeneous population are performing necessary supporting tasks (Page, 2010). Create a heterogeneous population by sorting the agents at the end of evolution by their fitness value,

identifying the top n% of the agents, and then cloning those agents to create 100 agents. As with the homogeneous agents, the performance of a heterogeneous population of BeTr -GEESE agents was terrible; less than 1% of the food available is brought to the hub so the results are not shown in the figure. Each agent is capable of only doing one subtask, and that means that agents cannot both pick up and drop objects. By contrast, Figure 3 shows that heterogenous blends of GEESE-BT agents often succeed, especially on the nest-maintenance task, precisely because they can learn more complicated programs. However, as more types of agents are added to the population, interference occurs, presumably because less fit agents perform only partial tasks and thus prevent top-performing agents from fully performing all tasks.

5.3 Heterogeneous Populations of Blended Agents

The better performance of the more complicated GEESE-BT programs suggests a way to blend the modular BeTr -GEESE programs so that they are more complicated as follows. First, sort agents most fit to least fit. Second, select the top $n\%$ of the agents. Each of these agents typically has a BT that performs only one subtask. Third, `or` these agents together by forming a root BT node with a parallel node and then adding agents as children to this root node. The parallel control node was chosen as it loosely acts as a logical `or`, meaning that the *blended agent* allows its modular BeTr -GEESE programs (sub-trees) to contribute to the behavior of the agent. For each agent, the order of the sub-trees is randomized so that different agents try execute subtasks in different orders.

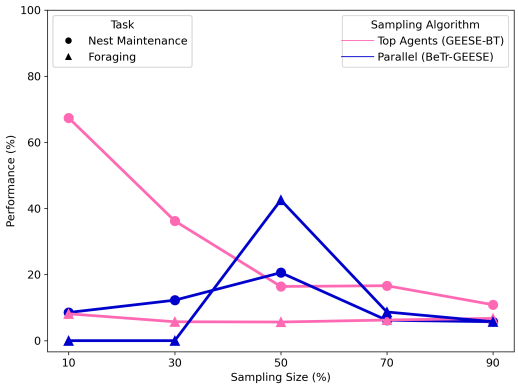


Figure 3: Population quality for populations created by sampling the top $n\%$ of agents for GEESE-BT and remixing BeTr -GEESE agents.

Figure 3 shows performance of heterogeneous populations of BeTr -GEESE agents formed using this blended approach. Heterogeneous populations of blended GEESE-BT agents are not shown because they perform no better than the heterogeneous populations formed in Section 5.2. The performance of blended BeTr -GEESE populations slowly increases and peaks at 50% and then decreases gradually for both tasks. Blending works precisely because the resulting agents capture pieces of agents capable of successfully performing needed subtasks.

6 Conclusion and Future Work

The PPA-structures used in BeTr -GEESE cause agents to learn simple programs capable of performing required subtasks. The evolving population of BeTr -GEESE agents succeeds because mutation, crossover, and lateral transfer cause the agents to change between simple programs while learning, effectually "time-multiplexing" between the simple programs. Subjective observations of animations of the evolution reveal a type of time-multiplexing that allows agents to adapt to different circumstances in their learning environment, effectually negating the need to rely on static strategies. GEESE-BT's success while learning is low because it is difficult to learn complicated programs even with the ad hoc reward structures designed to promote efficient learning. Heterogeneous populations of fixed BeTr -GEESE agents perform poorly because each agent learns only a portion of the entire task. The success of the heterogeneous population can be improved by blending the simple agents together, but performance is still not satisfactory.

Future work should explore BeTr -GEESE agents could rapidly adapt to changing conditions, provided that the grammar has a sufficiently rich set of execution nodes and primitive behaviors. Additionally, future work should explore genetic algorithm hyper-parameter settings that might enable BeTr -GEESE to evolve complex, resilient behaviors more efficiently.

A PRIMITIVE BEHAVIORS

The *CompositeDrop* behavior enables agent to drop the item it is currently carrying at the current location, and the *CompositeSingleCarry* allows an agent to pickup an object at its current location. The *MoveTowards* primitive behavior allows the agent to move a unit step towards a particular object in the environment, and *MoveAway* moves the agent a unit step away from an object. *Explore* moves the agent a unit step in a random direction. Obstacle avoidance in *MoveTowards*, *MoveAway*, and *Explore* use the simple bug following algorithm from Lumelsky & Stepanov (1987), which checks for obstacles or traps in its line-of-sight. If an object is detected, the agent moves one unit distance parallel to the object's surface.

B PPA GRAMMAR

$\langle root \rangle$ ::= $\langle sequence \rangle$ | $\langle selector \rangle$ (1)

$\langle sequence \rangle$ ::= [Sequence]$\langle ppa \rangle$[/Sequence]| [Sequence]$\langle root \rangle\langle root \rangle$[/Sequence] (2)
[Sequence]$\langle sequence \rangle\langle root \rangle$[/Sequence]

$\langle selector \rangle$ ::= [Selector]$\langle ppa \rangle$[/Selector]|[Selector]$\langle root \rangle\langle root \rangle$[/Selector] (3)
[Selector]$\langle selector \rangle\langle root \rangle$[/Selector]

$\langle ppa \rangle$ ::= [Selector]$\langle postconditions \rangle\langle ppasequence \rangle$[/Selector] (4)

$\langle postconditions \rangle$ ::= $\langle SuccessNode \rangle$ | $\langle ppa \rangle$ |[Sequence]$\langle postcondition \rangle$[/Sequence] (5)

$\langle postcondition \rangle$ ::= $\langle postcondition \rangle$[PostCnd]$\langle postconditiont \rangle$ (6)
[/PostCnd]|[PostCnd]$\langle postconditiont \rangle$[/PostCnd]

$\langle postconditiont \rangle$ ::= NeighbourObjects_$\langle objects \rangle$|NeighbourObjects_$\langle sobjects \rangle$|IsCarrying_$\langle dobjects \rangle$ (7)
NeighbourObjects_$\langle dobjects \rangle$|DidAvoidedObj_$\langle sobjects \rangle$|IsVisitedBefore_$\langle sobjects \rangle$

$\langle ppasequence \rangle$ ::= [Sequence]$\langle preconditions \rangle$[Act]$\langle action \rangle$ [/Act][/Sequence]|[Sequence]$\langle constraints \rangle$ (8)
[Act]$\langle action \rangle$[/Act][/Sequence]|[Sequence]$\langle preconditions \rangle\langle constraints \rangle$ [Act]$\langle action \rangle$[/Act][/Sequence]

$\langle preconditions \rangle$ ::= [Sequence]$\langle precondition \rangle$[/Sequence] (9)

$\langle precondition \rangle$ ::= $\langle precondition \rangle$[PreCnd]$\langle preconditiont \rangle$ [/PreCnd]| [PreCnd]$\langle preconditiont \rangle$[/PreCnd] (10)

$\langle preconditiont \rangle$ ::= IsDropable_$\langle sobjects \rangle$|NeighbourObjects_$\langle objects \rangle$|IsVisitedBefore_$\langle sobjects \rangle$| (11)
NeighbourObjects_$\langle objects \rangle$_invert|IsVisitedBefore_$\langle sobjects \rangle$_invert|
IsCarrying_$\langle dobjects \rangle$|IsCarrying_$\langle dobjects \rangle$_invert

$\langle constraints \rangle$ ::= [Sequence]$\langle constraint \rangle$[/Sequence] (12)

$\langle constraint \rangle$ ::= $\langle constraint \rangle$[Cnstr]$\langle constraintt \rangle$[/Cnstr]|[Cnstr]$\langle constraintt \rangle$[/Cnstr] (13)

$\langle constraintt \rangle$ ::= CanMove|IsCarryable_$\langle dobjects \rangle$| IsDropable_$\langle sobjects \rangle$ (14)

$\langle action \rangle$ ::= MoveTowards_$\langle sobjects \rangle$|Explore| (15)
CompositeSingleCarry_$\langle dobjects \rangle$|CompositeDrop_$\langle dobjects \rangle$| MoveAway_$\langle sobjects \rangle$

$\langle objects \rangle$ ::= $\langle sobjects \rangle$|$\langle dobjects \rangle$ (16)

$\langle sobjects \rangle$ ::= Hub|Sites (17)

$\langle dobjects \rangle$ ::= Food|Debris (18)

$\langle SuccessNode \rangle$ :: = [PostCnd]DummyNode[/PostCnd] (19)

Productions 1–4 define control nodes as part of a hierarchical set of PPA-structured sub-trees. Productions 5–7 define postconditions, which can be dummy nodes, an embedded PPA sub-tree, or condition nodes. Production rule 19, the *DummyNode*, is a condition node that always returns Success. Production 8 defines the right sub-tree of a PPA structure that sequentially combined preconditions and actions. Productions 9–11, define preconditions using condition nodes. Productions 12–14, define constraints using condition nodes. Production 15 calls the BT-based subtrees in which the primitive behaviors of the agent are defined. Productions 16–18 define static elements in the swarm environment, specifically a hub, and sites, all of which have fixed sizes. Production 18 defines movable objects, food and debris, in the environment. A *hub* is where agents originate, a *site* is a source of food for the agents, *food* is an object to be carred to the hub, and *debris* is an object to be removed from the hub. All execution nodes have self-explanatory names.

REFERENCES

R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, 1986.

D. Cli, P. Husbands, and I. Harvey. Evolving visually guided robots. In *From Animals to Animats 2. Proc. of the 2nd Intl. Conf. on Simulation of Adaptive Behavior*, pp. 374–383. MIT Press, 1993.

M. Colledanchise and P. Ögren. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Transactions on Robotics*, 33(2):372–389, 2017.

M. Colledanchise and P. Ögren. Behavior trees in robotics and al: An introduction. 2018.

M. Colledanchise, D. Almeida, and P. Ögren. Towards blended reactive planning and acting using behavior trees. In *2019 ICRA*, pp. 8839–8845. IEEE, 2019.

M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

M. Duarte, V. Costa, J. Gomes, T. Rodrigues, F. Silva, S. M. Oliveira, and A. L. Christensen. Evolution of collective behaviors for a real swarm of aquatic surface robots. *PloS One*, 11(3):e0151834, 2016.

Sondre A Engebråten, Jonas Moen, Oleg Yakimenko, and Kyrre Glette. Evolving a repertoire of controllers for a multi-function swarm. In *International Conference on the Applications of Evolutionary Computation*, pp. 734–749. Springer, 2018.

F. Fan, J. Xiong, M. Li, and G. Wang. On interpretability of artificial neural networks: A survey. *IEEE Transactions on Radiation and Plasma Medical Sciences*, 2021.

Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. Ponyge2: Grammatical evolution in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 1194–1201, 2017.

E. Ferrante, E. Duéñez-Guzmán, A. E. Turgut, and T. Wenseleers. Geswarm: Grammatical evolution for the automatic synthesis of collective behaviors in swarm robotics. In *Proc. of the 15th annual GECCO conference*, pp. 17–24. ACM, 2013.

R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

D. M. Gordon. *Ant encounters: interaction networks and colony behavior*, volume 1. Princeton University Press, 2010.

Eva Jablonka and Marion J Lamb. *Evolution in four dimensions, revised edition: Genetic, epigenetic, behavioral, and symbolic variation in the history of life*. MIT press, 2014.

D. Karaboga and B. Basturk. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *Journal of global optimization*, 39(3):459–471, 2007.

Jackie Kazil, David Masad, and Andrew Crooks. Utilizing python for agent-based modeling: The mesa framework. In Robert Thomson, Halil Bisgin, Christopher Dancy, Ayaz Hyder, and Muhammad Hussain (eds.), *Social, Cultural, and Behavioral Modeling*, pp. 308–317, Cham, 2020. Springer International Publishing. ISBN 978-3-030-61255-9.

C. A. Knoblock. Planning, executing, sensing, and replanning for information gathering. Technical report, University of Southern California, 1995.

L. König, S. Mostaghim, and H. Schmeck. Decentralized evolution of robotic behavior using finite state machines. *International Journal of Intelligent Computing and Cybernetics*, 2(4):695–723, 2009.

D. M. Kriesel, E. Cheung, M. Sitti, and H. Lipson. Beanbag robotics: Robotic swarms with 1-dof units. In *ANTS 2008*, pp. 267–274. Springer, 2008.

J. Kucking, A. Ligot, D. Bozhinoski, et al. Behavior trees as a control architecture in the automatic design of robot swarms. In *ANTS 2018*. IEEE, 2018.

J. Kuckling, Vincent Van P., and M. Birattari. Automatic modular design of behavior trees for robot swarms with communication capabilites. In *EvoApplications*, pp. 130–145, 2021.

Nick Lane. *The vital question: Energy, evolution, and the origins of complex life*. WW Norton & Company, 2015.

Wei-Po Lee. Evolving complex robot behaviors. *Information Sciences*, 121(1-2):1–25, 1999.

F. L. Lewis. Neural network control of robot manipulators. *IEEE Expert*, 11(3):64–75, 1996.

M. A. Lewis, A. H. Fagg, and A. Solidum. Genetic programming approach to the construction of a neural network for control of a walking robot. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE Intl. Conf. on*, pp. 2618–2623. IEEE, 1992.

V. J. Lumelsky and A. A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2(1):403–430, 1987.

Andrew L Nelson, Gregory J Barlow, and Lefteris Doitsidis. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57(4):345–370, 2009.

A. Neupane. Emergence of collective behaviors in hub-based colonies using grammatical evolution and behavior trees. Master's thesis, Brigham Young University. Department of Computer Science, 2019.

A. Neupane and M. A. Goodrich. Designing emergent swarm behaviors using behavior trees and grammatical evolution. In *Proc. of the 18th AAMAS conference*, pp. 2138–2140, 2019a.

A. Neupane and M. A. Goodrich. Learning swarm behaviors using grammatical evolution and behavior trees. In *IJCAI*, pp. 513–520, 2019b.

A. Neupane, M. A. Goodrich, and E. G. Mercer. Geese: grammatical evolution algorithm for evolution of swarm behaviors. In *Proc. of the 20th annual GECCO conference*, pp. 999–1006, 2018.

P. Ögren. Convergence analysis of hybrid control systems in the form of backward chained behavior trees. *IEEE Robotics and Automation Letters*, 5(4):6073–6080, 2020.

M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

S. E. Page. *Diversity and Complexity*. Princeton University Press, 2010.

P. Parashar, A. Naik, J. Hu, and H. I Christensen. Meta-modeling of assembly contingencies and planning for repair. *arXiv preprint arXiv:2103.07544*, 2021.

Pavel Petrovic. Evolving behavior coordination for mobile robots using distributed finite-state automata. In *Frontiers in evolutionary robotics*. InTech, 2008.

A. Pintér-Bartha, A. Sobe, and W. Elmenreich. Towards the light—comparing evolved neural network controllers and finite state machine controllers. In *Proc. of the 10th Workshop on WISES*, pp. 83–87. IEEE, 2012.

David Quammen. *The tangled tree: A radical new history of life*. Simon and Schuster, 2018.

C. I. Sprague, Ö. Özkahraman, A. Munafo, R. Marlow, et al. Improving the modularity of auv control systems using behaviour trees. In *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, pp. 1–6. IEEE, 2018.

Ivan Dale Steiner. *Group process and productivity*. Academic press, 1972.

Daniel Stonier and Michal Staniaszek. Behavior Tree implementation in Pyton, 12 2021. URL `https://github.com/splintered-reality/py_trees/`.

D. Sumpter and S. Pratt. A modelling framework for understanding social insect foraging. *Behavioral Ecology and Sociobiology*, 53(3):131–144, 2003.

R. S. Sutton. *Temporal credit assignment in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 1984.

Vito Trianni, Roderich Groß, Thomas H Labella, Erol Şahin, and Marco Dorigo. Evolving aggregation behaviors in a swarm of robots. In *European Conference on Artificial Life*, pp. 865–874. Springer, 2003.

A. Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pp. 429–528. Springer, 1996.