Learning Swarm Behaviors using Grammatical Evolution and Behavior Trees

Aadesh Neupane and **Michael Goodrich**
Brigham Young University, Provo, UT
aadeshnpn@byu.edu, mike@cs.byu.edu

Abstract

Algorithms used in networking, operation research and optimization can be created using bio-inspired swarm behaviors, but it is difficult to mimic swarm behaviors that generalize through diverse environments. State-machine-based artificial collective behaviors evolved by standard Grammatical Evolution (GE) provide promise for general swarm behaviors but may not scale to large problems. This paper introduces an algorithm that evolves problem-specific swarm behaviors by combining multi-agent grammatical evolution and Behavior Trees (BTs). We present a BT-based BNF grammar, supported by different fitness function types, which overcomes some of the limitations in using GEs to evolve swarm behavior. Given human-provided, problem-specific fitness-functions, the learned BT programs encode individual agent behaviors that produce desired swarm behaviors. We empirically verify the algorithm's effectiveness on three different problems: single-source foraging, collective transport, and nest maintenance. Agent diversity is key for the evolved behaviors to outperform hand-coded solutions in each task.

1 Introduction

Bio-inspired collectives like honeybee, ant, and termite colonies provide elegant distributed solutions to complex collective problems like finding food sources, selecting a new site, and allocating tasks. Effective collective behaviors emerge from biological swarms through local interactions [Gordon, 2010; Seeley, 2009; Sumpter, 2010].

Despite the potential benefits of bio-inspired algorithms, only a few organisms have been explored for their collective behavior; for example, very little is understood about the construction methods of termites [Margonelli, 2018]. One reason for slow research is the effort involved in understanding individual agent behavior and creating mathematical models to describe both individual and collective behaviors. Mimicking an evolutionary process with artificial agents may yield useful collective behaviors in a reasonable time.

Conventional approaches for evolving swarms behaviors used Finite State Machines (FSM) with or without neuro-evolutionary algorithms [Petrovic, 2008; Pintér-Bartha *et al.*, 2012; König *et al.*, 2009; Neupane *et al.*, 2018a]. When the system is complex and the number of states is huge, a hierarchical finite state machine (HFSM) offers benefits [Brooks, 1986; Valmari, 1996].

Unfortunately, HFSMs must trade-off between reactivity and modularity [Colledanchise and Ögren, 2017]. Also, behaviors encoded in HFSMs can be hard to debug and extend [Lim, 2009]. Behaviour Trees (BTs), which are useful in game design, overcome some HFSM limitations [Isla, 2005]. BTs have recently been used to evolve behaviors for robot swarms. For example, [Jones *et al.*, 2018] used genetic evolution algorithm to evolve a BT for a Kilobot foraging task.

This paper presents an algorithm that extends a distributed evolutionary algorithm called GEESE [Neupane *et al.*, 2018b] with BTs to generate swarm behaviors. GEESE is similar to the work in [Jones *et al.*, 2018], but decentralized GE is used in-place of genetic programming. There are two important differences between the use of GEESE in this paper compared to [Neupane *et al.*, 2018b]: First, the grammar that generated genotypes was changed to allow BT programs to be the evolutionary phenotype. Second, three fitness function types were designed to promote not only task-specific success but also diversity with useful learning through a bootstrapping process. The novelty of this work includes the grammar and fitness function types, enabling GE to evolve effective swarm BT programs. More specifically:

- Twenty-eight primitive individual behaviors were designed to mimic behaviors frequently seen in the swarm literature. A BNF grammar was designed that embeds the primitive behaviors as BT nodes, and experiments showed that the grammar was general enough to solve multiple collective spatial allocation tasks.
- Behavioral diversity among agents enabled high performance in all the swarm problems.
- The three fitness functions constrained the search space so that the GEESE algorithm efficiently found collective behaviors that solved the task.

2 Related Work

Evolutionary robotics (ER) is useful for generating autonomous behaviors. Early work applied neural-network-based evolving control architectures to visually guiding

robots [Cli *et al.*, 1993]. [Lewis *et al.*, 1992] applied staged evolution of a complex motor pattern generator for the control of a walking robot. [Doncieux *et al.*, 2015] aggregated achievements of ER and claimed that ER's agent-centered paradigm and behavior-based selection process allows challenging phenomena to be modeled and analyzed by statistics-based processes.

Evolving swarm behaviors was first described in [Kriesel *et al.*, 2008], which showed that individuals don't need to possess complex capabilities for effective swarm behaviors. [Duarte *et al.*, 2016] demonstrated an evolved neural net-based controller in a real and uncontrolled environment for homing, dispersion, clustering, and monitoring with ten aquatic surface robots. Key properties of swarm intelligence-based control were demonstrated, namely scalability, flexibility, and robustness.

Many ER approaches use neural networks (NNs) to evolve robot controllers. However, NN models are hard to reverse engineer and are not transparent, meaning that it is difficult to figure out why the algorithm choose a certain action during execution. A viable alternative to NN models is Genetic Programs (GPs), particularly Grammatical Evolution (GE). [Ferrante *et al.*, 2013] used GE to evolve foraging behaviors that than can be traced back to individual-level rules.

[Neupane *et al.*, 2018b] built a distributed multi-agent variant of GE to evolve swarm behaviors. Their approach performed better in a canonical GE task called the Sante Fe Trail problem, and successfully evolved foraging behaviors that outperformed a hand-coded solution and other GP-based solutions. The evolved behaviors were represented as a FSM. [Neupane and Goodrich, 2019] described a proof-of-concept about using grammatical evolution and BT to evolve behaviors for foraging task.

Representing swarm behaviors with FSMs gets troublesome when the number of states increases. HFSMs and Probabilistic FSMs are often used to overcome these limitations [Brooks, 1986]. BT representations are equivalent to Control Hybrid Dynamical Systems and HFSMs [Marzinotto *et al.*, 2014], and BTs promote increased readability, maintainability, and code reuse [Colledanchise and Ögren, 2018].

[Scheper, 2014] used a BT to evolve a program that enabled a DelFly drone to perform a window search and fly-through task. The evolved BTs performed well in both simulation and in the real world. [Kucking *et al.*, 2018] used BTs with *AutoMoDe* to perform foraging and aggregation.

3 Behavior Trees Overview

BTs are composed of a small set of simple components but they can give rise to very rich structures. A BT is a directed rooted tree where the internal nodes are called *control flow nodes* and leaf nodes are called *execution nodes*. Each tree node is either a parent or child node. The root node in a BT is the one without parents and all other nodes have one parent. All control flow nodes have at least one child.

The execution in BTs starts from the root node by generating *signals* or *control flows*, often called *ticks*, at a particular frequency. Signals and control flows are sent from a control flow node to its children. After receiving a tick, the nodes can be in only one of the following three states: *running*, *success*, or *failure*. *Running* indicates that processing for that node is ongoing, *success* indicates that the node has achieved its objective, and anything else is a *failure*.

The execution nodes in this paper are based on the primitive behaviors of bio-swarms. We use a python-based BT implementation. A variant of the GE algorithm called *GEESE* is used to convert a colony-specific grammar, written to produce BT programs, into the phenotype of the agents. The phenotype is an executable BT program.

In many BT formulations, there are four categories of control flow nodes: *Sequence*, *Selector*, *Parallel*, and *Decorator*; and there are two categories of execution nodes: *Action* and *Condition*. The *parallel* control node is not used in this paper. A memory module known as the *Blackboard* holds relevant BT data. We use a dictionary data structure as the Blackboard for the agents to store information. Each BT has its unique blackboard, and data sharing is forbidden. For more information, see [Colledanchise and Ögren, 2018].

4 GEESE-BT

GE is a context-free grammar-based genetic program paradigm that is capable of evolving programs or rules in many languages [Ryan *et al.*, 1998]. GE adopts a population of genotypes represented as binary strings, which are transformed into functional phenotype programs through a genotype-to-phenotype transformation. The transformation uses a BNF grammar, which specifies the language of the produced solutions.

We extend a specific distributed, multi-agent GE algorithm called *GEESE* in three ways: we present a BNF grammar capable of expressing BT for multiple swarm tasks, we present a way to address the credit-assignment problem through non-episodic rewards, and we show the importance of three distinct fitness function types. The BNF grammar guides the genotype-to-phenotype mapping process.

4.1 Swarm Grammar

This section describes the swarm grammar used with GEESE. The grammar uses a set of elements that are somewhat general for a set of spatial swarm tasks. The BNF grammar, which is used for every spatial swarm result shown in this paper, is shown below. The phenotype created from the mapping process using this grammar is a BT controller. The BT program is used by the agents to act in the environment. The grammar incorporates individual agent rules that can produce a valid BT, which induce desirable swarm behaviors.

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

$\langle sequence \rangle$ ::= $\langle execution \rangle$ | $\langle s \rangle \langle s \rangle$ | $\langle sequence \rangle$ $\langle s \rangle$ (2)

$\langle selector \rangle$ ::= $\langle execution \rangle$ | $\langle s \rangle \langle s \rangle$ | $\langle selector \rangle$ $\langle s \rangle$ (3)

$\langle execution \rangle$ ::= $\langle conditions \rangle$ $\langle action \rangle$ (4)

$\langle conditions \rangle$ ::= $\langle condition \rangle$ $\langle conditions \rangle$ | $\langle condition \rangle$ (5)

$\langle condition \rangle$ ::= NeighbourObjects | IsDropable_$\langle sobjects \rangle$ | (6)
NeighbourObjects_$\langle objects \rangle$ | IsVisitedBefore_$\langle sobjects \rangle$ |
NeighbourObjects_$\langle objects \rangle$_invert | IsCarrying_$\langle dbjects \rangle$ |
IsVisitedBefore_$\langle sobjects \rangle$_invert |
IsCarrying_$\langle dbjects \rangle$_invert | IsInPartialAttached_$\langle dbjects \rangle$
| IsInPartialAttached_$\langle dbjects \rangle$_invert

⟨*action*⟩ ::= MoveTowards_⟨*sobjects*⟩ | Explore | CompositeSingleCarry_⟨*dobjects*⟩ | CompositeDrop_⟨*dobjects*⟩ | MoveAway_⟨*sobjects*⟩ | CompositeMultipleCarry_⟨*dobjects*⟩ | CompositeDropPartial_⟨*dobjects*⟩ | CompositeDropCue_⟨*sobjects*⟩ | CompositePickCue_Cue | CompositeSendSignal_⟨*sobjects*⟩ | CompositeReceiveSignal_Signal (7)

⟨*sobjects*⟩ ::= Hub | Sites | Obstacles (8)

⟨*dobjects*⟩ ::= Food | Debris (9)

⟨*cobjects*⟩ ::= Signal | Cue (10)

⟨*objects*⟩ ::= ⟨*sobjects*⟩ | ⟨*dobjects*⟩ (11)

The right-hand side of production rule 1 defines the *sequence* and *selector* BT control structures. Production rules 2 and 3 recursively expand the nodes. Production rules 4 and 5 recursively define execution nodes, *actions* and *conditions*. Thus, the first five production rules in the grammar define the way the control nodes of the BT can be constructed. The production rules were devised so that the grammar can express many different BT shapes of arbitrary depth. More specifically, the first five production rules should be sufficient to generate any valid structure of BTs that do not use a *parallel* control structure.

Initial grammar designs placed the twenty-eight primitive behaviors directly in production rules 6 and 7. Primitive behaviors are atomic behaviors that represent the lowest level of behavior granularity, and are designed specifically to enable spatial swarm behaviors. Examples include *Move*, *DoNotMove*, *IsVisitedBefore*, etc. These primitive behaviors are specific to spatial tasks. Note the *invert* literal at the end of the some primitive behaviors is a *decorator* BT control node.

Preliminary results showed that appropriate swarm behaviors rarely evolved because finding useful combinations of primitive behaviors required a very big search space. Thus, we introduced a set of subjectively obvious combinations of spatially useful primitive behaviors to reduce the search space. The grammar adopts a clustering approach, reducing the total behaviors to twenty-one (rules 6 and 7). High-level behaviors include *MoveTowards*, *CompositeDrop*, etc. The *MoveTowards* behavior is the combination of *GoTo*, *Towards*, and *Move* primitive behaviors. These primitive behaviors were combined using the *sequence* BT control structure. The, other high-level behaviors were similarly subjective combinations of primitive-behaviors.

Production rule 8 defines the *general static elements* in the swarm environment, specifically a hub, sites, and obstacles. Production rule 9 defines the *dynamic objects* in the environment, specifically food and movable debris.

4.2 Non-Episodic Credit Assignment

In many previous GE and GP problems, an episodic learning environment is used. Episodic learning allows an agent to learn a good policy by focusing on long-term rewards. Every-time the agent reaches a terminal state or reaches a time-bound, the environment resets to an initial state. This type of episodic environment is widely used in reinforcement learning problems and genetic computation to test the algorithms/agents. The episodic learning environment allows visiting various parts of the state space in different episodes.

In general, an episodic learning environment is useful for testing single agent systems, but for the multi-agent system many complications arise. A big complication is that an agent not only needs to take environment states into account to choose the best action but also the states of other agents. In addition, the environment is partially observable because the states of other agents are not observable in distributed learning, which adds more complexity to the problem. Without experiencing any feedback from the environment about its controller, the agents don't have a reliable mechanism to assign credit for the collective success of the swarm. This hampers the GE's capability to converge to a useful solution.

Because pure episodic learning is not compatible with fully distributed multi-agent systems, the learning in our algorithm is not based on an episodic variant. Thus, there is no notion of terminal states and replay. Rather, the algorithm runs for a particular number of time steps, which are defined before the start of the simulation.

When two GEESE agents are in proximity to each other, either at the hub or near each other elsewhere in the environment, they pass their current genome to each other. Each agent in the swarm collects genomes from every other agent with whom it interacts, and once an agent has collected a sufficient number of genomes (see Table 1), the agent performs genetic operations independently. Since the evolution depends on agents having a high probability of being in proximity with each other, the density of agents in the environment is a critical albeit implicit criterion for successful learning.

The most fit 50% of the genomes in the agent's repository are then selected to be parents. Crossover is performed to create a new population of genomes. Mutation is then applied and a new population is formed by combining the mutated and unmutated individuals. The highest performing parents are included in the new population. A single genome is then selected based on diversity fitness, and the agent follows the corresponding BT program phenotype. In every step of simulation, each agent evaluates its fitness based on how well it performs in the environment using its BT controller. Three distinct class of fitness function, each necessary given the non-episodic learning and the large search space, are used to evaluate the agent.

A *generation* begins with an agent holding only a single genome and ends when that agent has selected a new genome using genetic operations. Since whether or not an agent has had enough encounters with others depends on the probability that agents meet in the world, each agent can experience multiple generations, or can experience few or no generations.

5 Fitness Functions Types

Three distinct fitness function types guide the evolutionary process: Type I (Diversity), Type II (Task-specific), and Type III (Bootstrap). Type I fitness acts as a surrogate when Type II and III function aren't salient, especially early in learning, and applies only to the genomes created between genetic operations. Type III fitness guides genomes to regions of the search space where Type II fitness can be used. Type II and III fitness evaluate agent success.

Parameters	GEESE
Number of Genomes Required to Trigger Genetic Operations	10
Parent-Selection	Fitness + truncation
Elite-size	1
Mutation Probability	0.01
Crossover	variable_onepoint
Crossover Probability	0.9
Genome-Selection	Diversity
Maximum Codon Int	1000
Number of Agents	100
Behavior Sample	0.1

Table 1: GEESE parameters used for the swarm experiments.

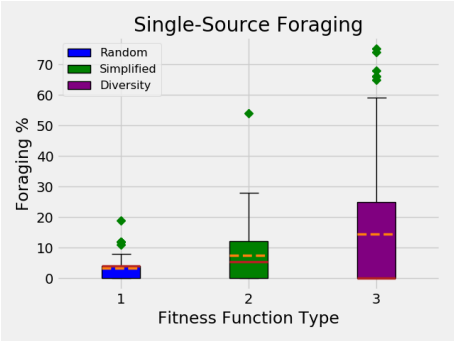

Figure 1: Genetic fitness functions comparison

5.1 Diversity Fitness

In non-episodic and multi-agent learning, it is very difficult for an agent to know which of its genomes are the most fit. The only way to test a genome is to use the corresponding BT controller and move around in the environment, but it is not possible to do this in any kind of efficient way because of the credit assignment problem. Nevertheless, a single genome must be selected so that the agent can act. This paper uses a heuristic measure of diversity to select a genome. We call this heuristic *Diversity Fitness*.

Let $D : \{\text{Phenotypes}\} \rightarrow \mathbb{R}$ denote diversity fitness. The **diversity** function takes a BT as input and extracts all the nodes from the tree. Recall that the BNF grammar produces only (a) "Sequence" and "Selector" BT controls and (b) primitive and higher-level agent behaviors. The extracted nodes from the tree are stored in a dictionary structure as control nodes and behavior nodes; the number of such nodes is also stored. The diversity fitness is then the total number of unique behavior nodes divided by the total behaviors defined in the grammar.

The relative importance of diversity fitness was measured against two other heuristics: *random fitness* and *simplified fitness*. *Random fitness* values are assigned randomly to the agent's phenotype. *Simplified fitness* values are computed based on the presence of three primitive behaviors: *Explore*, *CompositeSingleCarry* and *CompositeDrop*. These behaviors are the workhorse for swarms tasks. If all three behaviors are present, it assigns 100%, 66% for any two, 33% for one and 0% for everything else. Experiment results for a foraging task, see Figure 1, show that the *Diversity Fitness* is considerably better than *Random* and *Simplified*.

Promoting diversity among agents in a collective serves two purposes. First, promoting diversity allows a more thorough exploration of the set of possible solutions. Second, promoting diversity contributes to the resilience of the swarm as a whole; such diversity has been identified as key for tuning the number of agents committed to specific tasks in biological colonies [Gordon, 2010; Seeley, 2009]. The primary purpose of promoting diversity in this paper is to contribute to what is called the diversity of "types and kinds" [Page, 2010], enabling swarm resilience and efficient collective behavior, with greater exploration a side-benefit.

The diversity of a population of agents is computed at their phenotype level since it is the phenotype/program that actually interacts with the environment. As the evolutionary algorithm is trying to combine different type of behaviors within a BT, the diversity function ensures the agents with unique BT program phenotypes have higher chances of survival.

5.2 Task-specific and Bootstrap Function

This section describes how the phenotypical behaviors evolved by GEESE are evaluated for evolutionary fitness.

Agents act within the environment using the evolved BT as a controller. At each iteration, each agent evaluates and revises the quality of its BT based on the two level of fitness function discussed in this section: bootstrap and task-specific.

The credit assignment problem for swarm behavior design is a hard problem because (a) swarm success emerges from the collective actions of individual agents and (b) the reward to an individual is determined by the reward to the group. This credit assignment problem is exacerbated by different time scales used in the algorithm: the number of iterations, the number of generations, how long tasks take to execute, and how long it takes for information to flow between agents.

Preliminary experiments showed that task-specific fitness functions were not sufficient to induce desirable learned behavior in reasonable time. This paper uses two forms of "bootstrapping" to help boost learning. One form of bootstrapping, exploration, rewards exploration to different spatial regions. The other bootstrapping form, prospective fitness, rewards agents for trying potentially useful activities.

Whether or not a genome is considered fit enough to be a parent is determined by three elements: Task Fitness, Exploration Fitness, and Prospective Fitness. Let O, E, and P denote task fitness, exploration fitness, and prospective fitness, respectively. Task fitness, O, is defined by a task-specific objective function. The next subsubsection describes the three task-specific fitness functions used in this paper. Exploration fitness, denoted by E, and prospective fitness, denoted by P, are described in the subsequent subsubsections.

Let $\mathbf{A}_t$ denote the fitness of the agent A_t. The overall agent fitness of a genome to be a parent at time step t is given by

$$A_t = \beta(A_{t-1}) + (O_t + E_t + P_t). \quad (1)$$

Recall that a generation begins when an agent has a single genome and ends when the agent collects enough genomes from interactions with neighbors to produce a new genome

through genetic operations. Because task completion and generations can operate on different time scales for different agents, the fitness of an agent needs to have some memory to enable it to account for its part in the credit assignment problem [Agogino and Tumer, 2004]. The $t-1$ term in Equation (1) represents the agent fitness in the previous generation and β is the generational discount factor.

Task-Specific Objectives

The designer must specify the task or goal the swarms should accomplish. The goal is defined in terms of a task-specific *objective function*. The objective function embodies the intent of the designer for the swarms to accomplish. The objective function guides the evolution of individual agent behaviors.

Designers can define an objective function for the experiments they want the swarms to perform given that the function is (quickly) computable using the information available in the environment. For the experiments reported in this paper, three different task-specific objective functions are defined: *Foraging*, *Nest-Maintenance*, and *Cooperative Transport*. Denote these objective functions by F, N, and T, respectively; thus the objective function O is selected by a human operator from the set $O \in \{F, N, T\}$.

Single-Source Foraging

Denote foraging fitness by F where $F : \{\text{Phenotypes}\} \rightarrow \mathbb{R}$. F represents the designer's goal for the swarms to collect the maximum amount of food, and the set $\{\text{Phenotypes}\}$ represents all possible BT program phenotypes. More food is preferred to less food. Since the food could be in any part of the environment, the agents need to explore the environment, find the source of food, and transport food to the hub. Let $F(\text{phenotypes}) = |H_F|$, where H_F is food "items" collected to hub. Foraging fitness is set cardinality, i.e., total number of food items collected and transported from environment to hub.

Cooperative Transport

Denote cooperative transport fitness by T where $T : \{\text{Phenotypes}\} \rightarrow \mathbb{R}$. T represents the designer's goal for the swarms to transport a heavy object from a source location, located anywhere in the environment, to a destination. No agent is capable of moving the object alone. The agents need to explore the environment, find the heavy object, and coordinate with other agents to transport it to a desired destination. Let D_O be the set which contains all the objects collected and moved to a desired destination. Let $T(\text{phenotypes}) = |D_O|$, which is the number of heavy objects collected and moved to the desired destination.

Nest Maintenance

Denote the nest maintenance function by M where $M : \{\text{Phenotypes}\} \rightarrow \mathbb{R}$. M represents the designer's goal for the swarms to clear the area around the hub from debris. The debris objects are distributed around the periphery of the hub blocking the way for any exploring agents who might wish to return to the hub. In nest maintenance, agents need to explore near the hub, find the debris, and then move the debris to a place outside a boundary. Let N_D be the set which contains all the debris collected outside the boundary. Then

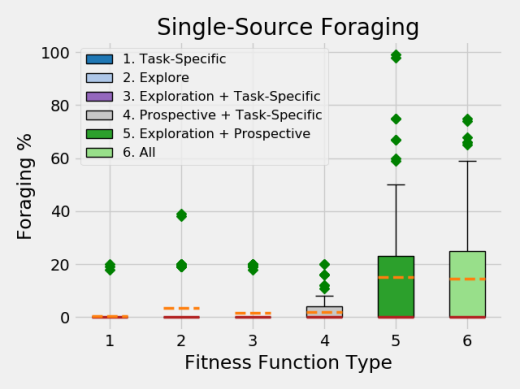

Figure 2: Comparison of different level of fitness functions

$M(phenotype) = |N_D|$ is the total number of debris objects moved outside the boundary.

Exploration Fitness

Exploration fitness is a form of bootstrapping that rewards those agents that explore the world. Denote the exploration fitness function by E where $E : \{\text{Locations}\} \rightarrow \mathbb{R}$. At each time step, each agent stores the location it is in. If the current location has not previously been visited by an agent, then the agent appends the current location into its memory. Exploration fitness is the total numbers of locations the agent has visited. The exploration function is computed exactly the same way each swarm task.

Prospective Fitness

Prospective fitness is a form of bootstrapping that rewards agents that persist in tasks that have a "prospect" of contributing to the collective good. Denote the prospective fitness by $\mathbf{P}$. In each task in this paper, moving things (food, debris, objects of interest) contributes to collective good. Thus P is set to the number of items that an agent is carrying.

Necessity of Task, Exploration, Prospective

The grammar allows BT to create two different control-flow nodes and twenty-one types of execution nodes, organized recursively. If the width is a behavior tree is N, then there are $2 * \frac{21!}{N! \times (21-N)!}$ ways to build the behavior tree at a particular depth. Thus the search space is a combinatoric problem where there are myriad ways to construct the BT.

Because of the size of the possible BT programs and the difficulty of the credit assignment problem, experiments showed that each fitness element (task-specific, exploration, prospective) was necessary to produce good swarm behaviors. Figure 2 compares the performance of the different combination of Type II and III fitness function. It is clear from Figure 2 that task-specific fitness function performed worse without the assistance from the bootstrap functions. Results are shown only for the foraging task, which might suggest that exploration and prospective fitness functions are sufficient to produce good performance. Results for other tasks show that task-specific, exploration, and prospective fitness are all necessary.

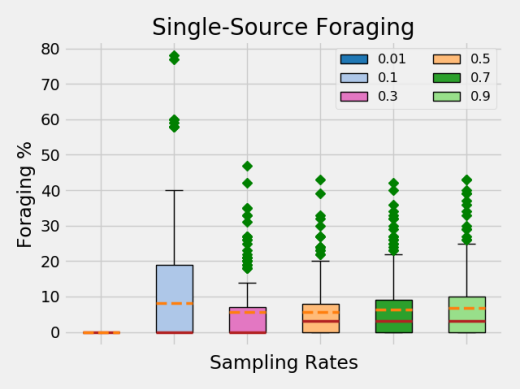

Figure 3: Performance with different phenotype sampling

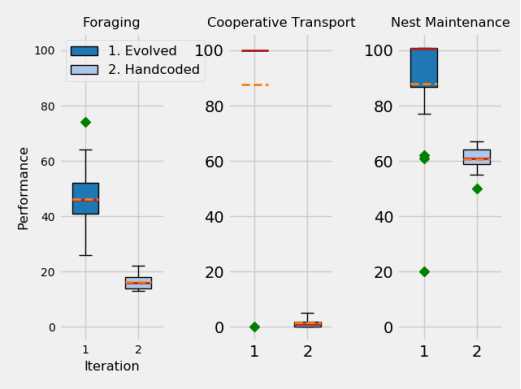

Figure 4: Empirical results on all swarm experiments.

6 Experiments and Results

This section describes the evolved collective behaviors for different swarm tasks and presents empirical results for the three tasks described in the introduction. Because of the stochastic nature of genetic evolution, the set of swarm behaviors can consist of different BT program phenotypes for each experimental run. The evolved behaviors that successfully perform the task are transferred to the test environment. Debris, sites, and objects are randomly placed in the world. Because the agent phenotypes are affected by the randomness in mutation and because of the randomness in the world, the evolved behaviors are evaluated for fifty separate runs. For each swarm task, a box-plot is superimposed at regular time intervals with the box indicating IQR (Q3-Q1), the whiskers with Q3 + 1.5*IQR upper bound and Q1 - 1.5*IQR lower bound, and outliers as rhombi.

6.1 Behavior Sampling

After running a pre-selected number of iterations, the swarm has a collection of agents, some of which are fit and some of which may not be fit. *Behavior sampling* is the process of choosing which evolved behaviors from the evolution process are used in a swarm when the swarm is placed in a test environment.

Two different methods of behavior sampling were considered. Both approaches use the fitness of agents as defined in Equation (1). First, in *best agent* sampling, only the highest performing agent's phenotype was picked from all the agents that took part in the evolution; in *best agent* sampling, all the test agents were homogeneous. Second, in *top agents* sampling, the agents' phenotypes were sorted based on the fitness values; then, a fixed number of top phenotypes were extracted. Results indicated that, because of homogeneity, *best agent* sampling performed worse than *top agents*, so *top agents* results are presented. Figure 3 show the performance of different sample size. The *best agent* sampling method performs worse whereas the sampling with 0.1% *top agents* performed best. The relative performance of sampling size between 0.5% - 0.9% show the resilience behaviors of the swarm as the performance remains roughly the same though more, less fit phenotypes are introduced in the world.

6.2 Swarm Tasks

Single Source Foraging

The leftmost graph of Figure 4 shows the swarm performance for Single-Source Foraging between evolved behaviors and Handcoded behaviors. The evolved behaviors clearly perform better than the hand-coded behaviors. Figure 3 suggests that the diversity of agents is one reason for the success of the learned behaviors; all the agents have the same behavior for the hand-coded solution. Another reason for the inferior performance of hand-coded behavior is that it is hard for humans to construct a BT tree with a cyclic nature as BT are directed trees. For foraging and other swarm tasks, there is cyclic pattern of visiting hub and other objects of interest.

Figure 5 shows one of the evolved BT for the foraging task. Hexagonal blue colored nodes represent selector controls, rectangular orange nodes represent sequence controls, and elliptical grey leaf nodes represent the execution nodes. A sequence control node returns "Success" when all of its children return "Success" else "Failure", while a Selector control node returns "Success" when at least one of its children return "Success" else "Failure". For details on the control flow in BT, see [Colledanchise and Ögren, 2018].

A BT is a directed acyclic graph (DAG) so it's fairly easy to analyze and understand the control flow. The tick/signal flows through the root node and uses depth-first search to traverse the nodes. In the remainder of this section, we first describe the control flow and return status traversal with the BT show in Figure 5. We then give an intuitive description about a specific scenario when an agent uses the BT to interact with the environment.

In the beginning, the control flows to "RootSelector" which is a selector node; for this selector node to be a "Success" either of its two children needs to return "Success" else "RootSelector" will return "Failure". The first child of the "RootSelector" has just one child node, "Selector17". "Selector17" has two children "Sequence16" and "Sequence56" and both of them are sequence control nodes. If "Sequence16" returns "Success" then the "Sequence56" need not be executed since a selector node only requires one of its children to return "Success". Then the return status will flow back to "Selector17" and to the root node. Thus it completes one full execution of the BT.

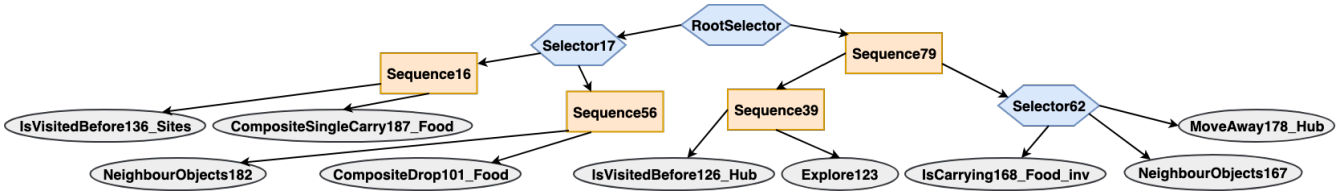

Figure 5: One of the evolved BTs for the foraging task

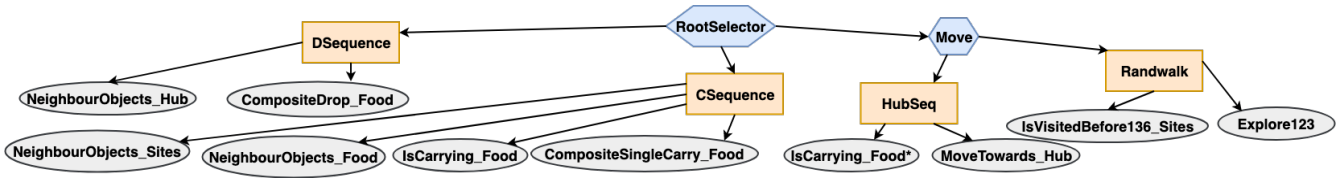

Figure 6: Hand-coded behavior tree for the foraging task

If "Sequence16" returns "Failure", then control flow is transferred to the "Sequence56" node. If "Sequence56" returns "Success", then the return status follows similar path to the root node completing one full execution. However, if both "Sequence16" and "Sequence56" return "Failure" then the "Selector17" is a failure which forces the root node to transfer the control flow to the right-hand block "Sequence79". "Sequence79" has two children: "Sequence39" and "Selector62". For "Sequence79" to return "Success" both its children should return "Success". We can reason about the return status from "Sequence39" and "Selector62" as done before.

Intuitively, the "Sequence16" block implements a 'Food carry phase', "Sequence56" block implements a 'Food drop phase', "Sequence39" implements an 'Explore phase', and "Selector62" implements a 'Traverse from hub to site' phase. In other words, when the agent is at a site and finds food, it carries the food [Sequence16]. If the agent has food and there are some objects near it, then it drops the food [Sequence56]. If the agent is not near the site and isn't carrying food, then it will explore the environment [Sequence39]. When the agent is not carrying food and finds an object, then it will move away from the hub [Selector62].

For contrast, Figure 6 represents the (less successful) hand-coded behavior for the foraging task. Recall that every agent in the experiments with hand-coded behaviors used the same BT. In the interest of space, the hand-coded BT is not described, but it can be interpreted similarly to the evolved BT illustrated above. Importantly, the hand-coded BT was designed to mimic foraging behavior of agents in nature.

Note that the BT in Figure 5 is just one of the BTs evolved by the algorithm. Note also that the BT subjectively does not match the ideal behavior of an individual in nature – the agent's behavior is imperfect. Importantly, the diversity experiment shown in Figure 3 shows that when there is a diverse set of those imperfect BTs used by a large number of agents, the performance obtained by the diverse-and-imperfect agents is superior to the homogeneous agent hand-coded behaviors designed to resemble the ideal bio-swarms. Subjective observations, to be quantified in future work, indicate that the evolved BT is readable: an expert can analyze its structure and modify few nodes to make it more efficient for a specific problem.

Cooperative Transport & Nest Maintenance

The middle and rightmost graphs of Figure 4 shows swarm performance for Cooperative Transport and Nest Maintenance, respectively. The evolved behaviors perform better than the hand-coded behaviors in each tasks. As with single-source foraging, the homogeneous nature of the hand-coded population interfered with the swarm's ability to succeed.

7 Conclusions and Future Work

A recursively defined BT-based grammar, built for spatial swarm tasks, can be used by the GEESE algorithm to evolve solutions to multiple swarm tasks. Because of the difficulty of solving the credit assignment problem, bootstrapping methods must be added to the fitness function to find solutions in a reasonable time. Moreover, results show that the diversity of the evolved behaviors was essential to their success.

The algorithm was able to evolve behaviors for different task and perform well in the simulation but should be applied to physical robots. Future work should also explore how the grammar can be modified to evolve other swarm tasks. Additionally, future work should explore probabilistic modeling of agent's sensing and acting capabilities.

Acknowledgments

The work in this paper was partially supported by a grant from the US Office of Naval Research. All opinions, findings, and results are the responsibility of the authors and not the sponsoring organization.

References

[Agogino and Tumer, 2004] A. K. Agogino and K. Tumer. Unifying temporal and structural credit assignment problems. In *Proc. 3rd Intl. Joint Conf. on Autonomous Agents and Multiagent Systems-Volume 2*, pages 980–987. IEEE Computer Society, 2004.

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

[Cli *et al.*, 1993] D. Cli, P. Husbands, and I. Harvey. Evolving visually guided robots. In H.L. Roitblatt J.A. Meyer and S.W. Wilson, editors, *From Animals to Animats 2. Proc. of the 2nd Intl. Conf. on Simulation of Adaptive Behavior*, pages 374–383. MIT Press, 1993.

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

[Colledanchise and Ögren, 2018] M. Colledanchise and P. Ögren. Behavior trees in robotics and AI: An introduction. *arXiv preprint arXiv:1709.00084*, 2018.

[Doncieux *et al.*, 2015] S. Doncieux, N. Bredeche, J.-B. Mouret, and A. E. G. Eiben. Evolutionary robotics: What, why, and where to. *Frontiers in Robotics and AI*, 2:4, 2015.

[Duarte *et al.*, 2016] 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.

[Ferrante *et al.*, 2013] 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. 15th annual conf. on Genetic and Evolutionary Computation*, pages 17–24. ACM, 2013.

[Gordon, 2010] D. M. Gordon. *Ant Encounters: Interaction Networks and Colony Behavior*. Princeton University Press, 2010.

[Isla, 2005] D. Isla. Handling complexity in the Halo 2 AI. *http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml [21.1. 2010]*, 2005.

[Jones *et al.*, 2018] S. Jones, M. Studley, S. Hauert, and A. Winfield. Evolving behaviour trees for swarm robotics. In *Distributed Autonomous Robotic Systems*, pages 487–501. Springer, 2018.

[König *et al.*, 2009] L. König, S. Mostaghim, and H. Schmeck. Decentralized evolution of robotic behavior using finite state machines. *Intl. Jnl. of Intelligent Computing and Cybernetics*, 2(4):695–723, 2009.

[Kriesel *et al.*, 2008] D. M. M. Kriesel, E. Cheung, M. Sitti, and H. Lipson. Beanbag robotics: Robotic swarms with 1-DOF units. In *Intl. Conf. on Ant Colony Optimization and Swarm Intelligence*, pages 267–274. Springer, 2008.

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

[Lewis *et al.*, 1992] 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*, pages 2618–2623. IEEE, 1992.

[Lim, 2009] C.-U. Lim. An AI player for DEFCON: An evolutionary approach using behavior trees. *Imperial College, London*, 2009.

[Margonelli, 2018] L. Margonelli. *Underbug: An Obsessive Tale of Termites and Technology*. Scientific American/Farrar, Straus and Giroux, 2018.

[Marzinotto *et al.*, 2014] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren. Towards a unified behavior trees framework for robot control. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 5420–5427. IEEE, 2014.

[Neupane and Goodrich, 2019] Aadesh Neupane and Michael A. Goodrich. Designing emergent swarm behaviors using behavior trees and grammatical evolution. In *Proceedings of the 18th Intl. Conf. on Autonomous Agents and Multiagent Systems*, pages 2138–2140, 2019.

[Neupane *et al.*, 2018a] A. Neupane, M. A. Goodrich, and E. G. Mercer. Geese: Grammatical evolution algorithm for evolution of swarm behaviors. In *Proceedings of the 17th Intl. Conf. on Autonomous Agents and Multiagent Systems*, pages 2025–2027, 2018.

[Neupane *et al.*, 2018b] A. Neupane, M. A. Goodrich, and E. G. Mercer. GEESE: Grammatical evolution algorithm for evolution of swarm behaviors. In *Proc. of the Genetic and Evolutionary Computation Conf.*, pages 999–1006. ACM, 2018.

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

[Petrovic, 2008] P. Petrovic. Evolving behavior coordination for mobile robots using distributed finite-state automata. In *Frontiers in Evolutionary Robotics*. InTech, 2008.

[Pintér-Bartha *et al.*, 2012] 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 Tenth Workshop on Intelligent Solutions in Embedded Systems*, pages 83–87. IEEE, 2012.

[Ryan *et al.*, 1998] C. Ryan, J. J. Collins, and M. O. Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conf. on Genetic Programming*, pages 83–96. Springer, 1998.

[Scheper, 2014] K. Y. W. Scheper. Behaviour trees for evolutionary robotics: Reducing the reality gap. 2014.

[Seeley, 2009] T. D. Seeley. *The Wisdom of the Hive: The Social Physiology of Honey Bee Colonies*. Harvard University Press, 2009.

[Sumpter, 2010] D. J. T. Sumpter. *Collective Animal Behavior*. Princeton University Press, 2010.

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