2019-02-01

Emergence of Collective Behaviors in Hub-Based Colonies using Grammatical Evolution and Behavior Trees

Aadesh Neupane
Brigham Young University

Emergence of Collective Behaviors in Hub-Based Colonies Using Grammatical Evolution and Behavior Trees

Aadesh Neupane

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

ABSTRACT

Emergence of Collective Behaviors in Hub-Based Colonies Using
Grammatical Evolution and Behavior Trees

Aadesh Neupane
Department of Computer Science, BYU
Master of Science

Animals such as bees, ants, birds, fish, and others are able to efficiently perform complex coordinated tasks like foraging, nest-selection, flocking and escaping predators without centralized control or coordination. These complex collective behaviors are the result of emergence. Conventionally, mimicking these collective behaviors with robots requires researchers to study actual behaviors, derive mathematical models, and implement these models as algorithms. Since the conventional approach is very time consuming and cumbersome, this thesis uses an emergence-based method for the efficient evolution of collective behaviors.

Our method, Grammatical Evolution algorithm for Evolution of Swarm bEhaviors (GEESE), is based on Grammatical Evolution (GE) and extends the literature on using genetic methods to generate collective behaviors for robot swarms. GEESE uses GE to evolve a primitive set of human-provided rules, represented in a BNF grammar, into productive individual behaviors represented by Behavior Tree (BT). We show that GEESE is generic enough, given an initial grammar, that it can be applied to evolve collective behaviors for multiple problems with just a minor change in objective function.

Our method is validated as follows: First, GEESE is compared with state-of-the-art genetic algorithms on the canonical Santa Fe Trail problem. Results show that GEESE outperforms the state-of-the-art by a) providing better solutions given sufficient population size while b) utilizing fewer evolutionary steps. Second, GEESE is used to evolve collective swarm behavior for a foraging task. Results show that the evolved foraging behavior using GEESE outperformed both hand-coded solutions as well as solutions generated by conventional Grammatical Evolution. Third, the behaviors evolved for single-source foraging task were able to perform well in a multiple-source foraging task, indicating a type of robustness. Finally, with a minor change to the objective function, the same BNF grammar used for foraging can be shown to evolve solutions to the nest-maintenance and the cooperative transport tasks.

Keywords: Swarms; Colonies; Bio-Inspiration; Grammatical Evolution

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Dr. Michael A. Goodrich for all the good ideas, discussion, feedback and motivation. I am grateful to be working under such an awesome researcher and a good human being. Dr. Goodrich has not just helped me in the technical side of research but also helped me to be a better person. I would also like to thank my committee Dr. Jacob Crandall and Dr. Casey Deccio for providing creative ideas and valuable feedback. I would also like to thank Dr. Eric G. Mercer for helping us with the design of initial BNF grammar. I am grateful for all my lab mates who provided motivation and words of encouragement in the time of need.

I am grateful to all my family members and friends who were by my side while I was busy pondering on difficult research questions. Last but not least, I would like to thank my friends from our start-up company Inspiring Lab.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Simple organisms have evolved over millions of years to collectively solve interesting and important problems. For example, bacteria interact with each other to move across cell surfaces efficiently by synthesizing large number of flagella [12]; slime molds move between complex locations using spatial memory [57]; ant colonies have evolved collective behaviors for foraging, nest defense, path planning and construction [24]; bees are able to select best sites among many good sites at their disposal [62]; and fish are able to avoid predators by organizing themselves into collective shapes that deter predation [32].

These organisms are able to solve these myriad problems with local interaction and without any central governing body. Often, each individual is very limited in the things that it can achieve whereas the swarms of those individuals can achieve great things. Because the collective behavior emerges from hundreds or thousands of local interactions, swarm solutions are often provide solutions that are resilient to agent drop-out and to environmental disturbances. The proper understanding of these collective behaviors has myriad applications in economic, computer networks, social behaviors, robotics, health care and many others. Because of their broad applicability, it is important to find general purpose algorithms that mimic collective behaviors.

Despite the benefits of modeling the algorithms based on these bio-swarms, only a few organisms have been explored for their collective behavior; see, for example, [45] which discusses how little we understand about the constructions methods of termites. One of the reasons for slow research in this field is the huge amount of time involved in understanding individual agent behavior and coming up with the mathematical model to describe both individual and collective behaviors [5]. Those behaviors were evolved as a result of survival strategy in a particular environment, taking hundreds of thousands years. If we are able to mimic this evolutionary process with artificial agents/robots then this will save researchers a lot of time and resources.

Researchers have created successful algorithms referred to as *spatial swarms*, meaning bio-inspired collectives that move or travel together more or less as a collective unit [13, 34, 42, 47, 69, 72, 76]. The properties of the swarms are investigated both with or without human interaction [3, 14, 22, 70].

Conventionally, mimicking these collective behaviors with robots requires researchers to study actual behaviors, derive mathematical models, and implement these models as algorithms. Since the conventional approach is very time consuming and cumbersome, we worked with an emergence-based method for the efficient evolution of collective behaviors. The proposed method provides an efficient approach by simplifying the problem into: (a) modeling the problem, (b) defining objective functions, and (c) specifying primitive behaviors. Importantly, this work was applied to *hub-based colonies* such as foraging and nest maintenance, contributing to contemporary efforts to extend the state-of-the-art beyond spatial swarms.

This thesis describes a framework combining evolutionary computing with Behavior Tree (BT) to generate swarm behaviors. Also, this work describes a distributed Embodied Evolution (dEE) algorithm called Grammatical Evolution algorithm for Evolution of Swarm bEhaviors (GEESE) that evolves desired collective behaviors from a set of primitive behaviors. The algorithm is online and distributed such that it can take advantage of a large number of swarm robots or agents. The algorithm evolves behaviors that are well suited for a given task instead of mimicking the bio-inspired collective behaviors. The algorithm adapts autonomously to the environment based on the local information and interaction between other robots or agents. Future work should extend the work to enable human interaction and readability of the evolved behaviors.

1.2 Research Questions

The goal of this thesis can be expressed in the following research question:
How can Grammatical Evolution (GE) be used to evolve effective swarm behaviors for different sets of tasks using the same Backus–Naur Form (BNF) grammar?
This overarching research question can be addressed by the answering the following specific research questions:

RQ1 What kind of structure does the BNF grammar need to have to express BT structures as well as swarm behaviors?

RQ2 How can GE algorithms be extended to work for multi-agent systems?

RQ3 Can BTs be used to represent swarm behaviors instead of FSMs?

RQ4 Will a single grammar be enough to evolve swarm behaviors for different tasks?

RQ5 Will evolved behaviors be better than hand-coded solutions?

1.3 Thesis Layout

The first part of the thesis will introduce the GEESE algorithm. The algorithm's implementation and experimental results were published in a scientific research paper, included in Chapter 2. Next, Chapter 3 addresses the research question posed above in a research paper that was recently submitted to a scientific conference. Chapter 4 provides the summary of this thesis and describes future work. Appendix 3.5 presents the programming framework used for the work in this thesis; the framework is generalizable and publicly available. Finally, Appendix B presents hand-coded behavior trees for several of the multi-agent problems addressed in this thesis.

Chapter 2

GEESE: Grammatical Evolution Algorithm for Evolution for Swarm Behaviors [1]

Abstract

Animals such as bees, ants, birds, fish, and others are able to perform complex coordinated tasks like foraging, nest-selection, flocking and escaping predators efficiently without centralized control or coordination. Conventionally, mimicking these behaviors with robots requires researchers to study actual behaviors, derive mathematical models, and implement these models as algorithms. We propose a distributed algorithm, Grammatical Evolution algorithm for Evolution of Swarm bEhaviors (GEESE), which uses genetic methods to generate collective behaviors for robot swarms. GEESE uses grammatical evolution to evolve a primitive set of human-provided rules into productive individual behaviors. The GEESE algorithm is evaluated in two different ways. First, GEESE is compared to state-of-the-art genetic algorithms on the canonical Santa Fe Trail problem. Results show that GEESE outperforms the state-of-the-art by (a) providing better solution quality given sufficient population size while (b) utilizing fewer evolutionary steps. Second, GEESE outperforms both a hand-coded and a Grammatical Evolution-generated solution on a collective swarm foraging task.

[1] This chapter is a reprint of the original GEESE [50] paper, which was published in the GECCO'18 conference. Note that a preliminary version of GEESE [51] was published as Extended Abstract at AAMAS'18. There are three authors, where Dr. Goodrich is my advisor and Dr. Mercer helped us to design BNF grammars. Most of the inspiration for this paper was based on the following two papers: *GESwarm* [20] and odNEAT [64].

2.1 Introduction

Simple organisms have evolved local interactions among individuals that produce useful collective behaviors: Bacteria interact with each other to move across cell surfaces efficiently by synthesizing a large number of flagella [12]; Ant colonies have evolved collective behaviors for foraging, nest defense, path planning and construction [24]; Bees are able to select best sites among many good sites at their disposal [62], and Fish are able to avoid predators by organizing themselves in collective shapes that deter predation [32].

Researchers have created successful algorithms that implement what we call *spatial swarms*, meaning bio-inspired collectives that move or travel together more or less as a collective unit [42, 47, 76]. Similarly, there has been significant research on designing bio-inspired or bio-mimetic distributed algorithms for hub-based colonies like honeybees, termites, and ants, particularly in optimization [16, 18, 30]. Many such algorithms are problem-specific, meaning that there is not a general solution for generating individual behaviors that produce desirable collective behaviors.

There are various forms of representing or controlling swarms including artificial neural networks, genetic programming structures, logic-based symbolic controllers, behavior-based controllers, and grammars. One approach to identifying individual behaviors that induce desirable collective behavior is to use Genetic Programming (GP), a type of Evolutionary Algorithm. GPs have been successfully applied to search ill-defined complex search spaces; for some search spaces, the time to find an acceptable solution is prohibitive [33].

Grammatical Evolution (GE) has been applied to problems for which GP can take too long. For example, in the Sante Fe Trail problem, the best solution produced by a GE is found more quickly and produces a superior solution compared to solutions produced by GP [33]. GEs work by restricting the search space by "seeding" the solution space using domain-specific knowledge. GEs have two main benefits over GPs. Firstly, GEs exploit prior knowledge, represented in the form of a grammar, which restricts the search space by only searching through valid grammars. Secondly, GEs exploit a greater distinction between a genotype and phenotype than more conventional GPs [52]. These benefits enable GEs to outperform conventional GPs in some problems.

When multiple agents are distributed in different spatial regions, the search for high-quality solutions can be accelerated if all the agents start in a different spatial location and interact with each other, sharing their knowledge of the search space accumulated so far. Distributed evaluation of fitness and searching different parts of the spatial domain suggest that a multi-agent GE may generate effective collective behaviors in swarms.

As agents interact with each other, each agent creates a temporary population from the genomes of its neighbors. Genetic operators like mutation and crossover are performed on the temporary population. We know of no distributed online GEs for swarms and colonies.

GEESE is evaluated on two problems: the Santa Fe Trail problem and a foraging problem. For the standard Santa Fe Trail problem, GEESE finds superior solutions in fewer generations compared to other variants of GE. For the foraging task, GEESE evolves collective behaviors that successfully accomplished foraging tasks and outperforms both a hand-coded and a GE-generated solution.

2.2 Background

There are many papers that "program" agents using genetic algorithms, but Yehonatan et al. [63] were the first to apply GP techniques to evolve Robocode players. They used genetic programming to produce a code in a tree-like structure. GEs also have been applied to robot and agent control. Burbidge et al. [4] used GE to generate a program that performed autonomous robot control.

The above papers used different forms of GE for autonomous control for a single agent, but the purpose of this paper is to evolve collective behaviors for multi-agent systems. O'Neill et al. [53] used particle swarm optimization and a social swarm algorithm to generate social programs. Each particle in the population was randomly initialized, fitness was calculated for each particle, and its velocity and location were updated using a specific update rule. Li

Chen et al. [7] combined a GE with a parallel GA to create an parallel evolutionary algorithm called GEGA. GEGA uses the genotype-to-phenotype mapping process of GE on the initial solution and the used the resulting phenotype as part of a population that is refined using a parallel GA.Ferrante et al. [20] developed a framework that could automatically synthesize collective behaviors for swarms using GE without using communication behaviors.

Cantu-Paz [6] identified three architectures for parallel genetic algorithms: master-slave, fine-grained, and multiple-population parallel GAs. Vacher et al. [74] described a multi-agent system guided by a multi-objective GA to find a balance point on the Pareto front. Stonedahl [66] described a multi-agent learning algorithm with a distributed GA to solve a bit-matching problem. Silva et. al. [64] developed "odNEAT", a distributed and decentralized neuroevolution algorithm for online learning in groups of autonomous robots that evolve both weights and network topology.

2.2.1 Grammatical Evolution

Grammatical Evolution (GE) is a context-free grammar-based GP paradigm that is capable of evolving programs or rules in many languages [52, 59]. 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. In GE, there is a central population of genomes where each genome is assigned a fitness or quality value. Only the portion of the population having higher fitness values are selected for genetic operations. We illustrate with an example.

BNF Grammar A BNF grammar is made up of the tuple N, T, P, S; where N is the set of all non-terminals symbols, T is the set of terminals, P is the set of productions that map N to T, and S is the initial start symbol and a member of N. When there are a number of rules that can be applied to the production of a non-terminal, a "—" (or) symbol separates the options. Note the difference in how we use "production" and "rule": a production is the set of possible things that can be produced for a single non-terminal, and a "rule" is one of the possible things produced (the right-hand-side of the production). The difference between terminal and non-terminal symbols is that non-terminal symbols are further expanded by a production rule whereas terminal symbols are not. A BNF grammar is used to translate genotype to phenotype.

The example BNF grammar that follows will be used in the Santa Fe Trail problem [36]. Details about Santa Fe Trail problem are explained in Section 2.4.1.

$$\langle \textit{code} \rangle ::= \langle \textit{code} \rangle \mid \langle \textit{progs} \rangle \tag{1}$$
$$\langle \textit{progs} \rangle ::= \langle \textit{condition} \rangle \mid \langle \textit{prog2} \rangle \mid \langle \textit{prog3} \rangle \mid \langle \textit{op} \rangle \tag{2}$$
$$\langle \textit{condition} \rangle ::= \texttt{if\_food\_ahead}(\langle \textit{progs} \rangle,\langle \textit{progs} \rangle) \tag{3}$$
$$\langle \textit{prog2} \rangle ::= \texttt{prog2}(\langle \textit{progs} \rangle,\langle \textit{progs} \rangle) \tag{4}$$
$$\langle \textit{prog3} \rangle ::= \texttt{prog3}(\langle \textit{progs} \rangle,\langle \textit{progs} \rangle,\langle \textit{progs} \rangle) \tag{5}$$
$$\langle \textit{op} \rangle ::= \texttt{left} \mid \texttt{right} \mid \texttt{move} \tag{6}$$

Koza gives an in-depth explanation of the above grammar [36].

Genome In GE, the genome defines how the left-derivation of a BNF grammar will proceed. GEs use genotypes encoded as binary or integer strings. A *codon* is a group of binary symbols, usually in a group of 4 or 8, chosen in such a way that there are enough bits per codon to be able to express both (a) the total number of productions and (b) the maximum number of right-hand-sides over each production. Consider a binary genome [001101000010001100100011] of length 24. Let the codon size be 4 for this example. The codon value for the five codon blocks is just the equivalent decimal values.

Mapping The mapping from genotype to phenotype is as follows: Let c denote the codon integer, let A denote the left-most non-terminal in the derivation, and let r_A denote the number of right-hand side rules associated with the production for A. GE maps from the production for the current non-terminal, A, to the right-hand side rule using the expression

$$RHSRule = c \% r_A \tag{2.1}$$

where % denotes the modulo operator. After a non-terminal is mapped to one of its right-hand side rules using Equation 2.1, the current codon is moved to the right in the binary string and the process resumes until no non-terminals remain. The resulting string of terminals is the *program* that the agent executes.

Phenotype The output from the mapping process is the phenotype. The phenotype represents a valid expansion of the BNF grammar. Continuing the example, below is a valid phenotype program obtained using the genome and grammar described earlier.

```
if_food_ahead(move, left)
```

The above program defines the behavior of an ant for the Santa Fe Trail problem. The explanation for the phenotype is, "If the food is just one step ahead, move forward; else turn left", for this program from the Santa Fe Trail grammar.

Given the genotype-to-phenotype mapping, we can describe how the canonical genetic operators operate on the genome. The genotype representation is a variable-length string. The mutation changes an integer to another random value, and one-point crossover swaps a section of the genetic code between parents.

2.2.2 Santa Fe Trail

The Santa Fe Trail problem [28] is a well-known problem used to benchmark new evolutionary algorithms. It is a "hard" problem due to evolutionary computing methods not solving it much more effectively than random search.

The agent's task is to find food in a predefined grid structure. The trail is embedded in a toroidally connected grid of 32×32 cells. The optimal food trail has 144 cells (89 containing food and 55 being gaps in the trail with no food). The objective of the Santa Fe Trail problem is to evolve a program that can navigate this trail, finding all the food. An agent can perform three moves: turn left, turn right, and move ahead.

There exists a standard solution to the Sante Fe Trail problem, one learned using GE, that serves as a baseline against which GP and GE solutions can be compared [36]. The baseline solution gathers all 89 food units within 543 moves.

2.3 GEESE

GEESE is loosely inspired by "odNEAT" [64], "Genetic Algorithm for Multi-Agent system" [74], and "GESwarm" [20]. GEESE provides an effective way to evolve programs where each individual agent/robot performs GE in a decentralized and distributed fashion similar to how "odNEAT" [64] performs neuroevolution. Since each GEESE agent encodes its own unique genetic instance and can run GE on its own (onboard), GEESE computation can be distributed.

GEESE is similar to GE in terms of initialization, genetic operators, and genotype-to-phenotype mapping. GEESE starts with a fixed number of agents initialized with a random string of integers (genotype). The genotype of an agent is also referred as an agent in GEESE; i.e, each agent has its own individual genotype. Each agent has an instance of **Algorithm 1** onboard.

Let A_j denote agent j, let $X = \{A_1, A_2, \ldots, A_M\}$ denote the set of M agents, let $\mathcal{M}_j$ denote the primary stack/memory of agent A_j, and let G_j denote the genotype of agent A_j. As displayed in Algorithm 1, each agent A_j is capable of performing three basic functions: *sense*, *act*, and *update* in a given environment.

Algorithm 1 Diversity Fitness

```
Require: $sense()$ //module
  for $A_j \in$ NEIGHBORHOOD$(A_j)$ do
    $\mathcal{M}_j \leftarrow \mathcal{M}_j \cup \{G_i\}$
  end for
  return $\mathcal{M}_j$
Require: $act()$ //module
  if $\mathcal{M}_j \neq \emptyset$ then
    $\mathcal{M}_j \leftarrow \mathcal{M}_i \cup \{G_j\}$
    $\mathcal{M}_j \leftarrow selection(\mathcal{M}_j)$
    $\mathcal{M}_j \leftarrow crossover(\mathcal{M}_j)$
    $\mathcal{M}_j \leftarrow \mathcal{M}_j \cup mutation(\mathcal{M}_j)$
  end if
  return $G^* \arg\max\{\text{FITNESS}(G_k) : G_k \in \mathcal{M}_j\}$
Require: $update()$ //module
  if FITNESS$(G_j) <$ FITNESS$(G^*)$ then
    $G_j \leftarrow G^*$
  end if
Require: agents $\leftarrow$ list(agent,M)
  for $agent$ in $agents$ do
    $agent.sense()$
    $agent.act()$
    $agent.update()$
  end for
```

2.3.1 Sense

During the *sense step*, agent A_j uses input from its sensors to get information about the environment. The sense step is general, meaning that depending on the application an agent can sense multiple things about the world. General to all applications is the sensing of neighbors. If agent A_j senses other agents A_i nearby, denoted $\textsc{Neighborhood}(A_j)$, agent A_j requests each agent $A_i \in \textsc{Neighborhood}(A_j)$ to share its genotype G_i. The agent then temporarily stores the genotypes of nearby agents in $\mathcal{M}_j$.

Consider two methods for determining agent A_j's neighbors. First, use a Euclidean distance threshold, where all agents within a given distance are considered neighbors. Second, randomly sample from all agents in the population regardless of Euclidean distance. For this method, the sensing capability of the agents in X is controlled by the $INTERACTION\_PROB$ parameter. A greater parameter value means that there is a higher probability of agents being in each other's neighborhoods.

2.3.2 Act

During the *act step*, agent A_j checks its memory $\mathcal{M}_j$ where it stores all the genotypes received from agents in its neighborhood. If its memory M_j is empty, then it doesn't perform any action; otherwise, it performs a series of operations. First, it adds its own genotype to the memory. Second, a *selection* operator is performed on its memory to get parents. Selection samples from memory a subset of genotypes to be used to form a new population; the paragraph below describes selection methods. Third, the *crossover* operator is applied to the parents to add children to the population; parents are discarded from the population after crossover. The set of children is mutated using a *mutation* operator, fitness is evaluated for the set of children, and the mutated genotypes are added to the population. The highest performing child, G^*, is returned.

We explored four variations of selection operators: tournament, truncation, NSGA-II and Pareto tournament [19]. Each selection operator returns the best individuals from the $\mathcal{M}_j$, which are termed "parents". We conducted experiments with these operators and subjectively chose the *tournament operator* for its efficiency.

2.3.3 Update

During the *update step*, an agent checks whether the best genotype returned by the *act* step is superior to the current genotype. Agent A_j will replace genotype G_j with G^* if the fitness of G^* exceeds the fitness of G_j.

2.3.4 Differences between GEESE and Conventional GE

The advantage of GEESE over standard GE is that each agent is capable of applying genetic operators on its own. Using GEESE, each agent is able to compute GE onboard without centralized storage of genome population; i.e. GEESE computation is distributed and performed online. This enables each agent to search the evolutionary fitness landscape starting from a different location in the landscape. Instead of evaluating the whole population at each generation, GEESE evaluates locally, increasing the chances of average individuals to reach the next generation. This enables GEESE to maintain genetic diversity and slow down convergence.

To illustrate this increase genetic diversity, consider a GEESE population with nine agents and consider how three agents, A_1, A_2, and A_3, might update their genomes. For simplicity, assume that agents $\{1, 2, 3\}$ move randomly and perform GEESE in a sequential order as illustrated in Figure 2.1. Each agent in the blue circles, exchanges information with neighbors in the pink circles and perform *sense*, *act* and *update* methods of **Algorithm 1**. Since each agent performs the genetic operations locally, i.e. the genetic operations are performed using only genetic information from neighboring agents, the genetic diversity between groupings is likely to stay distinct for several generations. Also, if the agent doesn't find any neighbors, it doesn't perform any genetic operations and holds on to its genome.

This makes it possible that individual agents who initially have average fitness, relative to other agents, to have a chance to take part in genetic operations.

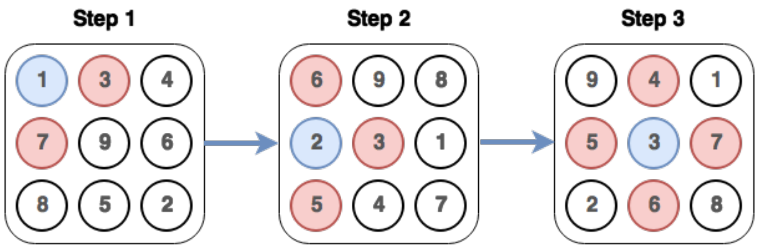

Figure 2.1: Three conceptual evolution steps in GEESE

The effect of preserving genetic diversity is illustrated in Figure 2.2, which represents a conceptual fitness landscape for the nine agents from the previous example. If a standard GE algorithm is applied, only the top performing individual or individuals is picked to perform genetic operations among the global population. For this fitness landscape, agent A_4 and A_7 would be picked and GE would be stuck at a local minimum. Since standard GE ignores other individuals, there is a higher chance of loss of diversity early in the search, resulting in slower convergence or a higher probability of converging to a suboptimal solution.

By contrast, in GEESE the genetic operations are applied to local neighborhoods, allowing some average-performing individuals to survive to the next generation. For this fitness landscape, agent A_8 and A_3 will also take part in genetic operation even though agent A_4 and A_7 are the top performers. Thus, GEESE inhibits premature convergence and increases genetic diversity with the use of local genetic operators. This pattern of avoiding premature commitment to a local "hill" in the fitness landscape is well-known and is utilized, for example, in beam-search [58].

GEESE defines 'generation' slightly differently than GE. A single generation of GEESE refers to the spontaneous execution of each agent's methods (sense, act, and update). Since the act and update method is only activated when the agent senses other neighbors, in each generation only a few agents will be genetically active.

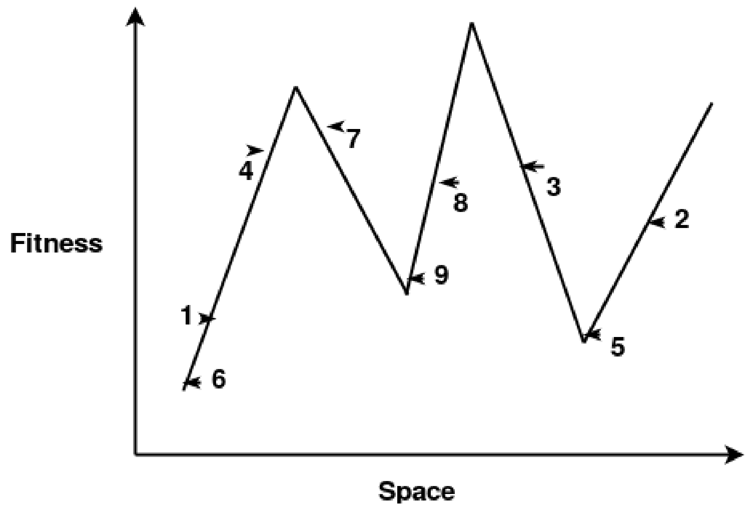

Figure 2.2: Conceptual fitness landscape with nine agents.

2.4 Experiments

Section 2.4.1 compares GEESE against GE on the Santa Fe Trail problem [28]. Section 2.4.2 then demonstrates the use of GEESE to evolve individuals rules to enable collective foraging. All the experiments reported in this paper are carried out using PonyGE2 [19] and GEESE code is merged into PonyGE2. We experimented with 50, 100, 200, 400, and 1000 runs for each experiment reported below. Since lengthy runs didn't reveal significant differences, results are presented for 50 runs.

2.4.1 Santa Fe Trail

To evaluate GEESE, we define *fitness* to be the total number of food units collected by the agents on the trail. Fitness reaches its maximum value when all the food units are collected. Also, we define *minimum steps* to be the minimum number of steps the agent needs to take in order to collect all the food units, i.e. 89. The BNF grammar used for this experiment was described in Section 2.2.1.

Santa Fe Trail using Standard Fitness function

50 evolutionary runs were conducted for both conventional GE and for GEESE. Parameters used in the simulations are shown in Table 2.1. Two parameter values from Table 2.1 need discussion. First, GE has a population of genomes, denoted by *Genetic Population Size*, which is a global collection of genomes on which the genetic operators act. GEESE doesn't have a shared population of genomes, but rather *Number of Agents*, which defines a unique number of agents that form neighborhoods with probability *Agent Interaction Prob.* Because neighborhoods are probabilistic, the neighbor relation is asymmetric. Agents receive genomic information from their neighbors to create a temporary population on which the genetic operators act.

Second, the *Maximum Codon Int* is the maximum allowed integer that can be produced by the codon, c, in Equation 2.1.

Parameters	**GE**	**GEESE**	**Novelty GE**	**Novelty GEESE**
Genetic Population Size	100	N/A	100	N/A
Number of Agents	N/A	100	N/A	100
Agent Interaction Prob	N/A	0.85	N/A	0.85
Maximum Generation	50	50	50	50
Mutation Probability	0.01	0.01	0.01	0.01
Crossover Probability	0.9	0.9	0.9	0.9
Maximum Codon Int	1000	1000	1000	1000

Table 2.1: GE/GEESE parameters used for the Santa Fe Trail problem.

Figure 2.3 demonstrates that GEESE converges to a solution faster than standard GE. The solid line is the mean fitness. The shaded region represents one standard deviation across the 50 trials.

GEESE required fewer generations with a smaller effective population size to solve the Santa Fe Trail in comparison with standard GE. The *hit rate* is the ratio of the total number of successful programs which found all 89 food units to the total number of experiment runs. The *hit rate* for Standard GE with 50 runs is just 6% whereas the hit rate from GEESE is 57%. This shows that GEESE, a decentralized and on-line GE algorithm, outperforms standard GE in the Santa Fe Trail problem.

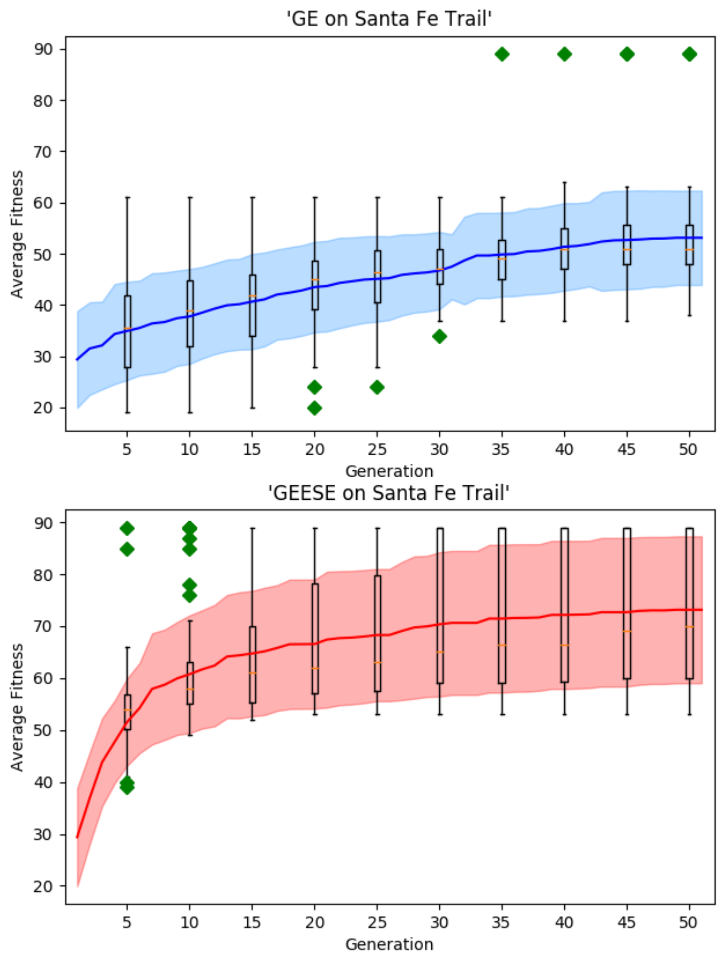

Figure 2.3: GEESE converges quicker than GE.

Santa Fe Trail using Novelty Search

Lehman et. al. [41] introduce novelty search techniques and Urbano et al. [73] used Novelty Search (NS) techniques to improve the performance of GE. Objective-based fitness is replaced by fitness functions that favor novelty in the genetic population. The idea is not to select the fittest individuals for reproduction but rather those with the most novel behaviors. Novel individuals are rewarded, thus favoring exploration of different phenotypical behaviors regardless of their fitness. The idea is that by exploring the behavior space without any goal besides novelty, ultimately an individual with the desired behavior will be found.

NS requires the definition of distances between behavior descriptors [73]. Those descriptors may be specific to a task or suited for a class of tasks. The descriptors are normal vectors that capture behavior information along the whole evaluation or simply sampled at particular instants. Given a behavior function and a distance metric, the novelty score of an individual is computed as the average distance from its k-nearest neighbors in the population

$$\rho(x) = \frac{1}{k \sum_{y \in \textsc{Neighborhood}(x)} \textsc{Dist}(x, y)}$$

Urbano et al. [73] used three behavior descriptors for the Santa Fe Trail problem: the amount of food eaten, food eaten sequence, and step sequence. We will refer to these behavior descriptors as novelty1, novelty2, and novelty3, respectively. These behaviors descriptors are implemented exactly as described in the paper.

50 evolutionary runs were conducted for both GE with NS using the parameters detailed in Table 2.1. It is evident from Figure 2.4 that GEESE converges to the solution faster than standard GE with NS.

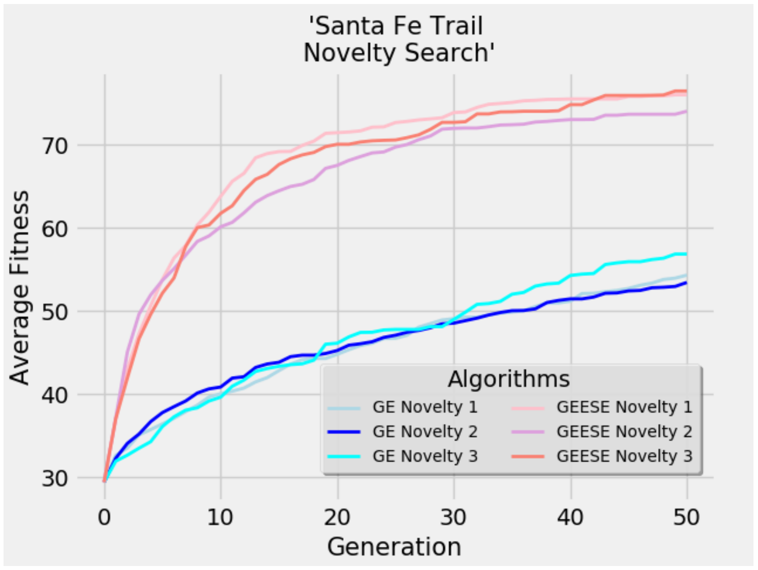

Figure 2.4: GEESE with NS converges quicker than GE with NS.

One of the programs evolved by GEESE using novelty search is shown in Figure 2.5. This program was able to complete the trail in fewer steps than any other known solution to the Santa Fe Trail problem, including those given by Urbano [73].

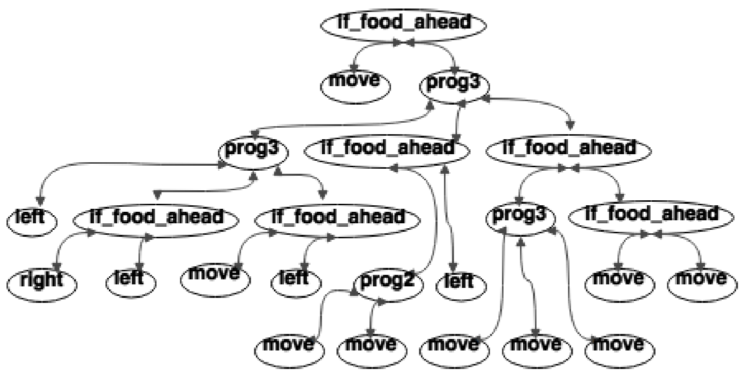

Figure 2.5: The evolved program that solved the Santa Fe Trail problem is just 324 steps.

Additionally, GEESE with NS has a higher hit rate than standard NS. The *hit rate* for standard NS with 50 runs is just 26% whereas the hit rate from GEESE with NS is 58%. Using the concepts from NS and combining it with GEESE, GEESE outperformed all known solutions to the Santa Fe Trail problem by solving it in minimum number of steps. Table 2.2 shows relevant performance metrics over many different algorithms. λ-LGP is a variant of GP that outputs sequence of instructions instead of the tree-like structure of general GP using mutation and replacement genetic operations. λ-LGP outperforms GEESE in solution quality, but no computation time is given and λ-LGP is not a distributed algorithm. Since, GEESE is not tailored to solve only Santa Fe trail problem it has slightly lower success rate than other tailored algorithms.

Sensitivity

Two parameters had a significant impact on GEESE performance: (a) Interaction Probability (IP) and, (b) number of agents. A high value of interaction probability correlates to a higher chance of agents interacting frequently with one another enabling them to share genetic information between them. As shown in Figure 2.6, when the IP is close to 1, GEESE

Algorithms	**Mean Fitness**	**Success Rate**	**Minimum Steps**
Koza GP [36]	N/A	N/A	543
Cartesian GP [48]	N/A	0.93	N/A
MuACOsm [8]	N/A	N/A	394
λ-LGP [65]	89	1	N/A
GE [26]	80.1	0.63	N/A
GE (Repair) [71]	75.02	0.32	N/A
Grammatical Swarm [53]	80.18	0.58	N/A
Attribute Grammar [31]	N/A	0.85	N/A
GE (Novelty) [73]	77.88	0.41	331
GE (Constituent) [21]	N/A	0.9	337
GEESE	**84.1**	**0.6**	**324**

Table 2.2: Comparison with state-of-the-art methods for Santa Fe Trail. Values not presented in the original work are marked as "N/A"

performs worse than when IP is lower. The reason is that with IP near 1, GEESE degenerates to standard GE. Figure 2.6 shows performance for IP $\in \{0.2, 0.4, ...0.99\}$.

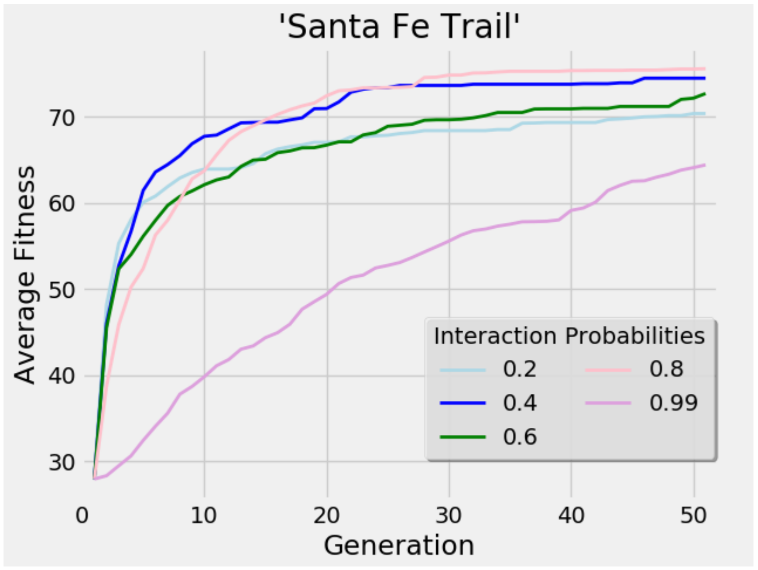

Figure 2.6: Average fitness for varying interaction probabilities.

In addition to IP, performance varied as a function of the number of agents. An increase in the number of agents meant each agent would start in a different location of the search space; i.e. a large section of search space will be explored during initialization. Figure 2.7 illustrates that the increase in the number of agents enables GEESE to reach the solution in fewer generations.

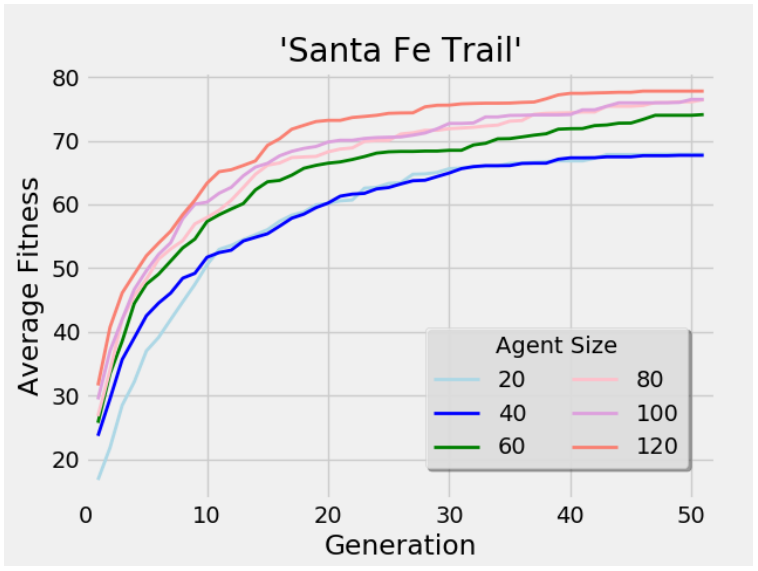

Figure 2.7: Average fitness with varying number of agents.

Summary

The performance of GEESE was superior to other variants of GE as seen in Table 2.1. It was able to solve the Santa Fe trail problem in fewer generations and using a smaller set of agents. Additionally, one evolved program collected all the food in the Santa Fe Trail problem in fewer steps than any other solution derived using a GE.

2.4.2 Swarm Behavior

This section applies GEESE to discover agent behaviors that enable a swarm to perform a foraging task.

Problem Description

Figure 2.8 illustrates a foraging scenario known as the *center place food foraging* problem [68]. The agent's task is to collect food from a *source* region in the environment and bring the food to a *hub* region in the environment. A *source* has following properties: (a) it has a fixed number of food units available, (b) a single agent can take only one food unit per visit, and

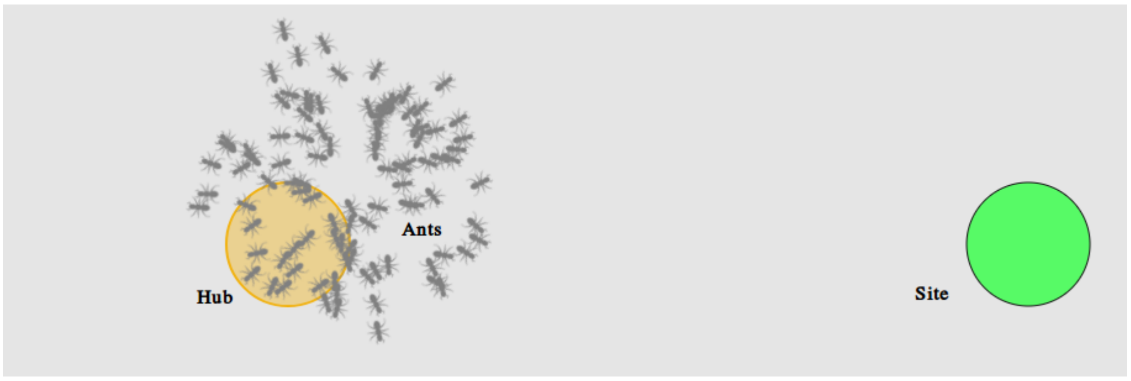

Figure 2.8: A foraging environment with one hundred agents, a hub, and a single food source.

(c) the source location is anywhere within a pre-defined bounded area except for within a fixed distance from the hub.

A *hub* acts as a nest to the agents. Initially, all the agents are located inside the hub. Agents carry the food units from source to hub where the food is stored. Agents do not have prior information regarding the source location. Agents carry as many food units back to the hub as possible during a fixed time frame.

Agent Behavior

Recall that GEESE creates a program from a user-specified grammar. Thus, the first step for applying GEESE to the foraging problem is to specify the grammar. Section 2.4.2 presents and describes the grammar.

Grammar

The grammar used for this experiment is:

$$\langle start \rangle ::= \langle ruleset \rangle \tag{1}$$
$$\langle ruleset \rangle ::= \langle rule \rangle \mid \langle rule \rangle \ \langle ruleset \rangle \tag{2}$$
$$\langle rule \rangle ::= \langle state \rangle \ \langle pc \rangle \ \langle transition \rangle \tag{3}$$
$$\langle pc \rangle ::= \langle bool \rangle \ \langle bool \rangle \ \langle bool \rangle \ \langle bool \rangle \ \langle bool \rangle \ \langle bool \rangle \ \langle bool \rangle \tag{4}$$
$$\langle transition \rangle ::= \langle change\_state \rangle \mid \langle change\_mem \rangle \tag{5}$$
$$\langle change\_state \rangle ::= \langle prob \rangle \ \langle state \rangle \tag{6}$$

$$\langle \textit{change\_mem} \rangle ::= \langle \textit{prob} \rangle \ \langle \textit{id} \rangle \ \langle \textit{bool} \rangle \tag{7}$$

$$\langle \textit{state} \rangle ::= \text{randwalk} \mid \text{tosource} \mid \text{tonest} \mid \text{dropcues} \mid \tag{8}$$

$$\text{pickcues} \mid \text{sendsignals} \mid \text{recsignals}$$

$$\langle \textit{bool} \rangle ::= \text{false} \mid \text{true} \mid \text{don't\_care} \tag{9}$$

$$\langle \textit{id} \rangle ::= \text{dropfood} \mid \text{wantfood} \tag{10}$$

$$\langle \textit{prob} \rangle ::= 0.1 \mid 0.2 \mid 0.5 \mid 0.7 \mid 1.0 \tag{11}$$

$$\langle \textit{bool} \rangle ::= \text{true} \mid \text{false} \tag{12}$$

Each non-terminal can be expanded either to groups of non-terminal plus terminal symbols or to groups of terminal symbols. A valid string is produced only when all non-terminals have been expanded. Once GEESE has generated a valid string, the agents *sense*, *act*, and *update*.

Productions (1) and (2) expand the start non-terminal into a set of productions and rules. The productions produced by this grammar encode a probabilistic state machine with one-bit of internal memory. The agent uses this state machine to interact with the environment. Production (3) defines a rule as having three parts: *states*, *preconditions*, and *transitions*.

States Agent behaviors are determined by the state of the agent. Each state generates a *low-level behavior*, which is an activity that an agent can execute in the environment. The low-level behaviors are defined by the terminal symbols of Production (8). Four behaviors are communicative: $dropcues, pickcues, sendsignals$, and $recsignals$. Three behaviors are non-communicative: $randwalk, tonest$, and $tosource$, which move the agent around. There are two boolean internal states stored in agents memory: *wantfood* and *dropfood*. These values are checked by precondition check: P_{drop_food} and P_{want_food} described in the next section.

- $B_{randwalk}$ Move in random direction.
- B_{tonest} Move towards the nest.

- $B_{tosource}$ Move to the source.
- $B_{dropcues}$ Drop pheromone cues in the environment before it moves. The cue contains the direction of a source.
- $B_{pickcues}$ If cues are found in the environment, pick up the cue, read the information from that object, and change internal memory to reflect the knowledge from the cue.
- $B_{sendsignals}$ Broadcast information in a fixed radius around it. The signal is information about a discovered source.
- $B_{recsignals}$ If a signal is found, accept the information from the signal and change internal memory by adding the knowledge acquired from the signal.

Preconditions Productions (4) and (9) expand $< pc >$ into a *precondition* string of boolean bits with length 7. The seven bits in the precondition string correspond to: P_{has_food}, P_{on_nest}, P_{on_source} , P_{drop_food}, P_{want_food}, $P_{on\_signals}$ and P_{on_cues}. Validating the preconditions are done during *sense* step. The set of preconditions are "and"-ed together, meaning all preconditions must be satisfied.

Transitions Productions (5)-(7) specify probabilistic *transitions* between states. Each transition is associated with a probability value p_i. Provided that all preconditions are met, the transition is executed with probability p_i. Production (6) defines a behavior transition with a certain probability to the specified next state. Production (7) defines the transition of internal memory; there are two internal states, *dropfood* and *wantfood*, as defined in production (10). Production (7) has three arguments: the probability of changing, the name of the internal state variable to change, and internal state variable's new value. The change in internal state changes the evaluation of the precondition check.

Evolutionary Setup

Fifty evolutionary runs are executed. Each evolutionary run lasts 50 generations and involves 100 agents. A single-point crossover with probability 0.9 and a mutation probability of 0.01 is used. A generational-type of replacement is used. Tournament selection chooses individuals used for crossover.

In the experiments, the size of the food source depletes as the agents consume food. The agent capacity of sensing food in the environment is directly proportional to food size. So, using "the total time required to collect all food" as a fitness metric, in this case, is not feasible because some small "scraps" of depleted food may never be found. Thus *Fitness* is defined as the total number of food units collected during a fixed time period. As discussed in Section 2.4.1, algorithm performance for a given problem is dependent on the number of agents. Using the same fixed time for experiments with a varying number of agents will give inaccurate fitness metric; i.e. for more agents less time should be allocated when compared to fewer agents. Careful evaluation of the hand-coded solution using varying numbers of agents indicated that 284 fixed time steps produces reliable fitness evaluation results for 100 agents.

Results

We created a hand-coded benchmark for comparison. The benchmark consisted a set of 13 rules from the grammar 2.4.2. The hand-coded program was able to collect 77 units of food in an average of 284 time steps. We also ran standard GE.

50 evolutionary runs for both standard GE and GEESE were performed. Using standard GE, the evolved programs were able to locate the food source in the environment and bring back the food to the hub. On average, 56 units of food were collected by the agents using standard GE, which is fewer than the hand-coded benchmark. Moreover, the evolved programs lacked communication behaviors, even though the grammar was capable of expressing communication. Interaction and communication enable ants and bees them to solve complex problems [24].

GEESE evolved programs that were able to collect food efficiently by making use of communication behaviors. The evolved program was more efficient than the hand-coded program; one evolved program had only 8 rules in contrast to the 13 rules in the hand-coded program. The evolved program on average collected 83 units of food in 284 time steps which is higher than the benchmark value. The evolved program contained communication behaviors.

One of the benefits of using GE for the evolution of swarm behaviors is that evolved behaviors are expressed as a human-readable program. Two of the rules in the evolved program deserve mention. The first rule says if the agent is in *randwalk* or *tosource* state and if it satisfies all the precondition then it has 0.7 chance of transitioning to *recsignals* state. Simply put, if the agent is wandering around or heading to the source then it occasionally transitions to listening for signals; see Appendix. The second rule is obvious, namely when the agent arrives at the *hub* and drops its food, it returns to the *source.* Although the grammar involved both signals (e.g., communication) and cues (e.g., pheromones), only signaling behaviors evolved; a single communication behavior was enough to efficiently solve the foraging problem.

2.5 Future Work

GEESE showed promising results on the Santa Fe Trail problem, and it should be tested on other GE problems. In addition, GEESE should be applied to developing agent behaviors for other colony-based tasks like construction, cleaning, and defense. Other future work should explore whether GEESE would still produce effective rules when foraging in a wide range of complex environments.

2.6 Summary

This paper presented the GEESE algorithm, a grammatical evolution algorithm for a multi-agent system. Results demonstrated the effectiveness of GEESE on the Santa Fe Trail problem, outperforming the state of the art in terms of minimum steps to solve the problem.

Additionally, GEESE was used to evolve individual behaviors that lead to successful colony-level foraging, outperforming behaviors evolved by conventional grammatical evolution as well as hand-coded individual behaviors. Finally, results illustrated that the agent behaviors could be interpreted by humans.

Chapter 3

Designing Emergent Swarm Behaviors Using Behavior Trees and Grammatical Evolution [1]

Abstract

It is difficult to design collective artificial swarm behaviors that generalize through diverse sets of environments. Evolution has given collectives like ants, bees, and termites the ability to perform diverse behaviors efficiently without any centralized control. This paper introduces an algorithm and grammar that evolve problem-specific collective behaviors by specifying problem-specific fitness functions. An agent grammar is introduced that enables the GEESE multi-agent grammatical evolution algorithm [50] to evolve Behavior Tree (BT) programs. Given human-provided, problem-specific fitness-functions, the learned BT programs encode individual agent behaviors that produce desired swarm behaviors. We verify the robustness properties of the framework by providing empirical evidence on four different problems: single-source foraging, multiple-source foraging, collective transport, and nest maintenance. The evolved behaviors outperform hand-coded solutions for each task.

[1]This chapter is a reprint of the paper, which was submitted in the conference. The paper is under review.

3.1 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 (see, for example, [25, 61, 70]).

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 [45]. 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 [5]. Mimicking an evolutionary process with artificial agents may yield useful collective behaviors in a reasonable time.

Conventional approaches for evolving swarm behaviors used Finite State Machines (FSM) with or without neuro-evolutionary algorithms [20, 35, 50, 55, 56]. When the system is complex and the number of states is huge, a hierarchical finite state machine (HFSM) offers benefits [2, 75]. Unfortunately, HFSMs must trade-off between reactivity and modularity [10]. Also, behaviors encoded in HFSMs can be hard to debug and extend [44]. Behaviour Trees (BTs), which are useful in game design, overcome some HFSM limitations [27].

BTs have recently been used to evolve behaviors for robot swarms. Jones et al. [29] used genetic evolution algorithm to evolve a BT for a Kilobot foraging task. The evolved BT included ten primitive behaviors and performed the task in simulation and reality.

Distributed grammatical evolution coupled with BTs might be adequate for generating swarm behaviors. This paper presents a framework combining a distributed evolutionary algorithm called GEESE [50] with BTs to generate swarm behaviors. The framework is similar to Simon et al. [29], but decentralized Grammatical Evolution is used in-place of vanilla genetic programming. There are two important differences between the use of GEESE in this paper compared to Neupane et al. prior work: First, the grammar that generated genotypes was redesigned to allow BT programs to be the evolutionary phenotype. Second, a fitness function was designed to promote not only task-specific success but also diversity and curiosity.

We identified twenty-eight primitive individual behaviors designed to mimic behaviors frequently seen in the swarm literature. We then design a BNF grammar that embeds the primitive behaviors as BT nodes, and claim that the grammar is general enough to solve many collective spatial allocation tasks. To test the claim, we evaluate four canonical swarm problems: *single-source* and *multiple-source foraging*, *cooperative-transport*, and *nest maintenance*. For each problem, the performance of the evolved swarm behaviors are compared to the results from hand-coded counterparts.

3.2 Related Work

Evolutionary robotics (ER) is useful for generating autonomous behaviors. Early work applied neural-network-based evolving control architectures to visually guiding robots [9]. Lewis et al. [43] applied staged evolution of a complex motor pattern generator for the control of a walking robot. Stephan et. al. Doncieux et al. [15] aggregated achievements of ER and claimed that ER's agent-centered paradigm and behavior-based selection process allows challenging phenomena to modeled and analyzed by statistics-based processes.

Evolving swarm behaviors was first described in [38], which showed that individuals don't need to possess complex capabilities for effective swarm behaviors. Kriesel [37] developed a generic framework for the evolution of swarm behaviors. Kriesel's framework contained distributed evolution software for efficient distributed evolution computing, a swarm simulation engine with real-time physics, and a topology-evolving neural network.

Duartel et al. [17] 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.

Node	**Succeeds**	**Fails**	**Running**
Sequence	If all children succeed	If one child fails	If one child returns running
Selector	If one child succeeds	If all children fail	If one child returns running
Action	Task completion	Task impossible to complete	Task being computed
Condition	If true	If false	Never

Table 3.1: Control Flow details for BT nodes.

Many ER approaches use Neural Networks (NNs) for evolving a robot controller. 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. [20] used GE to build a framework to evolve foraging behaviors than can be traced back to individual-level rules.

Neupane et. al. [50] developed a multi-agent variant of Grammatical Evolution (GE) to evolve swarm behaviors. Their approach was able to perform 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.

Representing swarm behaviors with FSMs gets troublesome when the number of states increases. Hierarchical Finite State Machines (HFSMs) and Probabilistic Finite State Machine are often used to overcome these limitations. BT representations are equivalent to Control Hybrid Dynamical Systems and HFSMs [46], and BTs promote increased readability, maintainability, and code reuse [11].

Scheper [60] used a BT in DelFly drone to perform a window search and fly-through task. The evolved BTs performed well in both simulation and in the real world. Jones et al. [29] used genetic evolution to evolve BT to perform a foraging task. Jonas et al. [40] used BTs with *AutoMoDe* to perform foraging and aggregation.

3.3 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* [11]. Each node in the tree can be associated as parent or child nodes. The root node in a BT is the one without parents and all other nodes have one parent. The 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**, with 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 only in 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 [67]. A variant of the GE algorithm called *GEESE* [49] 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. A detailed discussion on the evolution of a BT via GEESE is presented in Section 3.5.1.

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 our work. A memory module known as the *Blackboard* holds relevant BT data. We use a simple blackboard with a global scope, but a formalism that doesn't share data between tree instances. Table 3.1 summarizes what each BT node type can do; for more information, see [11].

3.4 Swarm Grammar

This section describes the swarm grammar used with GEESE, and identifies important elements in the tasks to be performed.

Grammatical Evolution (GE) is a context-free grammar-based genetic program paradigm that is capable of evolving programs or rules in many languages [52, 59]. 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. In GE, there is a central population of genomes where each genome is assigned a fitness or quality value. Only the portion of the population having higher fitness values are selected for genetic operations.

This paper uses a specific GE algorithm called **GEESE** [50], which was designed for distributed multi-agent systems. Since the BNF grammar guides the genotype-to-phenotype mapping process, proper design of the BNF grammar is critical in GE. A contribution of this paper is a BNF grammar that allows BT programs to be created for a variety of swarm problems.

3.4.1 Grammar Structure and Rules

The BNF grammar is shown below. *Every swarm experiment in this paper use this grammar to evolve task-specific behaviors.* The BNF grammar is the critical GE element that maps the genotype of the agents to a phenotype, encoded as a valid BT program. 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.

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

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

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

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

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

$\langle \textit{condition} \rangle$

$$\langle \textit{condition} \rangle ::= \text{NeighbourObjects} \mid \tag{6}$$

$$\text{IsDropable\_}\langle \textit{sobjects} \rangle \mid \text{NeighbourObjects\_}\langle \textit{objects} \rangle \mid$$

$$\text{NeighbourObjects\_}\langle \textit{objects} \rangle\text{\_invert} \mid$$

$$\text{IsVisitedBefore\_}\langle \textit{sobjects} \rangle \mid \text{IsCarrying\_}\langle \textit{dbjects} \rangle \mid$$

$$\text{IsVisitedBefore\_}\langle \textit{sobjects} \rangle\text{\_invert} \mid$$

$$\text{IsCarrying\_}\langle \textit{dbjects} \rangle\text{\_invert} \mid \text{IsInPartialAttached\_}\langle \textit{dbjects} \rangle \mid$$

$$\text{IsInPartialAttached\_}\langle \textit{dbjects} \rangle\text{\_invert} \mid$$

$$\langle \textit{action} \rangle ::= \text{MoveTowards\_}\langle \textit{sobjects} \rangle \mid \text{Explore} \mid \tag{7}$$

$$\text{CompositeSingleCarry\_}\langle \textit{dobjects} \rangle \mid$$

$$\text{CompositeDrop\_}\langle \textit{dobjects} \rangle \mid \text{MoveAway\_}\langle \textit{sobjects} \rangle \mid$$

$$\text{CompositeMultipleCarry\_}\langle \textit{dobjects} \rangle \mid$$

$$\text{CompositeDropPartial\_}\langle \textit{dobjects} \rangle \mid$$

$$\text{CompositeDropCue\_}\langle \textit{sobjects} \rangle \mid$$

$$\text{CompositePickCue\_Cue} \mid$$

$$\text{CompositeSendSignal\_}\langle \textit{sobjects} \rangle \mid$$

$$\text{CompositeReceiveSignal\_Signal}$$

$$\langle \textit{sobjects} \rangle ::= \text{Hub} \mid \text{Sites} \mid \text{Obstacles} \tag{8}$$

$$\langle \textit{dobjects} \rangle ::= \text{Food} \mid \text{Debris} \tag{9}$$

$$\langle \textit{cobjects} \rangle ::= \text{Signal} \mid \text{Cue} \tag{10}$$

$$\langle \textit{objects} \rangle ::= \langle \textit{sobjects} \rangle \mid \langle \textit{dobjects} \rangle \tag{11}$$

The right-hand side of the 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. Examples include *Move*, *DoNotMove*, *IsVisitedBefore*, etc. Preliminary results showed appropriate swarm behaviors rarely evolved because finding useful combinations of primitive behaviors required a very big search space.

There were subjectively obvious combinations of primitive behaviors that we used to reduce the search space. The grammar above adopts a clustering approach, reducing the total behaviors to sixteen (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 *sequence* control structure from BT. Similarly, other high-level behaviors were combined using subjective choices of primitive-behaviors.

Production rule 8 defines the *general static elements* in the swarm environment, specifically a hub, sites, and obstacles; see subsection 3.4.2. Production rule 9 defines the *dynamic objects* in the environment, specifically food and movable debris.

3.4.2 Grammar Literals

We now define a few important elements in swarm experiments: **hub**, **source**, **cue**, **signal**, **debris**, and **food**. These elements correspond to the literals in the grammar from the previous subsection.

A *hub* acts as a nest/home to the agents. Initially, all the agents are located inside the hub. The radius and location of the hub may vary in the experiments. Modeled after ant and bee colonies, a *hub* is the central place where most of the local communication and interaction between agents occurs.

A *source* or *site* is a region in the environment which contains a fixed number of food units, which determines its quality. The radius and location of the source/site may vary in the experiments and there could be more than one sources in a particular experiment.

Cues and *signals* are the communication mechanisms in the swarms. A *cue* is a stationary object placed in the environment by an agent that contains information about the states of the environment. Other agents must be present physically at the location of the cue to "read" the information. One example of the cue-based communication is the pheromones deposited by many ants species while foraging. A *signal* is modeled as a mobile object, connected the agent, that broadcasts information about the states of the environment. Other agents can "read" the information from the signal when they are within the signal radius.

Debris are the scattered pieces of waste in the environment. These objects obstruct the trail of the agents and agents want this debris to be far away from their usual routes and other important places like the *hub* and *sites*.

3.5 Framework

The framework "Swarm" [49] is a Python based Agent-Based Modeling framework. This framework encodes a variant of Grammatical Evolution (GE) algorithm called GEESE [50]. The swarm framework enables three capabilities. First, the framework allows a user to quickly create, evolve, and test swarm-based behaviors using built-in core components (spatial grids, agent scheduler, evolution algorithms, and BTs) or customized implementation. Second, the framework allows behaviors to be visualized using a browser-based interface. Third, the framework provides data analysis tools.

The swarm framework consists of five major **modules**: *model*, *agent*, *objects*, *space*, and *time*. The **model** module holds the model-level attributes (spatial space), manages the agents and objects, and schedules. The **model** module has a special function called *step*, which sends signals to all the elements in the environment. Through steps, the model module defines the environment in which the tasks are performed. The **agent** module holds the agent-level attributes (sensors, actuators) and a *step* function. The **objects** module holds the definitions of all environmental objects like *Hub*, *Source*, *Food*, *Debris*, *Obstacles* and others. The **space** module defines a spatial structure using a dynamic grid instance with an easy interface to get information about the objects in the space.

The **time** module defines the scheduler, which controls the order of the agents' activation. At each step of the model, the agents are activated based on the scheduler. Then, each agent activates their BT, changing them internally, interacting with one other, and/or the environment. Thus, the *step* function coordinates the simulation across all major models (model, agent, and time).

3.5.1 Agents

Agent properties are similar to the agents in previous instances of the GEESE algorithm [50] and included three basic functions: *sense*, *act*, and *update*. Furthermore, each agent is capable of performing genetic operations on its own. In this paper, *sense* and *update* has been encoded as BT elements whereas *act* function is implemented as a *step* function. The step function holds the step trigger for an agent's BT program. So, the agent's main components are GEESE for grammatical evolution and a BT module to act in the environment.

Every agent is initialized with a fixed random string of integers; this random string is the agent's *genotype*. Since each agent has all required methods to compute genetic operations, each agent converts the *genotype* into a *phenotype* using the *Genotype-Phenotype mapping* process. This mapping process takes the BNF grammar and genotype as input and uses the mathematical expression from [50]to convert the genotype into a valid BT program (phenotype).

3.5.2 Simulator

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 for a GEESE multi-agent system is partially observable because the states of other agents are not observable in distributed learning, which adds more complexity to the problem.

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

3.5.3 Behavior Sampling

After running a sufficient number of epochs, 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 (3.1) below. 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 agents were homogeneous. Second, in *top agents* sampling, the agent phenotype was sorted based on the fitness values; then, a fixed number of top agents phenotype were extracted. Results indicated that, because of homogeneity, *best agent* sampling performed worse than *top agents*, so *top agents* results are presented.

3.6 Fitness

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

When two GEESE agents are in proximity to each other at the hub or near each other elsewhere in the world, they pass their current genome to each other. Each GEESE 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 3.2), the agent performs genetic operations independently. 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 within an epoch or can experience few or no generations.

Parameters	**GEESE**
Number of Genomes Required to Trigger Genetic Operations	70
Parent-Selection	Fitness + truncation
Mutation Probability	0.01
Crossover	variable_onepoint
Crossover Probability	0.9
Genome-Selection	Diversity
Maximum Codon Int	1000
Agent Size	100

Table 3.2: GEESE parameters used for the swarm experiments.

GEESE applies four genetic operators: parent-selection, cross-over, mutation, and genome-selection. We discuss each of these operations below. Table 3.2 shows the parameters used in the different genetic operations and is included so that others can replicate the paper's results. Other parameters were chosen directly from [50].

3.6.1 Parent-Selection, Crossover, Mutation

The **parent-selection** operator returns the most fit individuals from the population, which are termed "parents". After parents are selected, the **crossover** operator is applied to the parents to add children to the population; parents are discarded from the population after *crossover*. The set of children is then mutated using a *mutation* operator. Following mutation, a single genome is retained by the agent; the process for deciding which genome to retain is described in the next subsection.

Parent Fitness

The credit assignment problem for swarm behavior design is a hard problem precisely because (a) the collective behavior 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 epochs, 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 form of bootstrapping, prospective fitness, rewards agents for persisting in activities that may be useful.

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

$$A_t = \beta(A_{t-1}) + (O_t + E_t + P_t). \tag{3.1}$$

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, epochs, and generations can operate on different time scales, the fitness of an agent needs to have some memory to enable it to account for its part in the credit assignment problem [1]. The $t-1$ term in Equation (3.1) represents the agent fitness in the previous generation and β is the generational discount factor.

Task-Specific Objectives

The task or goal that the user wants the swarms to accomplish needs to be specified. The goal is defined in terms of a task-specific *objective function*. The objective function embodies the intent of the user for the swarms to accomplish. The objective function guides the evolution of individual agent behaviors.

The user 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 document, 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 $O \in \{F, N, T\}$.

Multi- and Single-Source Foraging Denote foraging fitness by F where $F : \{\text{Phenotypes}\} \rightarrow \mathbb{R}$. F represents the user's goal for the swarms to collect the maximum amount of food and the set {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 user's goal for the swarms to transport a heavy object from source location, located anywhere in the environment, to 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 user'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 $M(\textbf{phenotype}) = |N_D|$ 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 for all the task that involves spatial information.

Prospective Objective

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 **P**. In each task in this paper, moving stuff (food, debris, objects of interest) contribute 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 three different control-flow nodes and sixteen types of execution nodes, organized recursively. If the width is a behavior tree is N, then there are $3 * \frac{16!}{N! \times (16-N)!}$ ways to build the behavior tree. 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, preliminary experiments showed that each fitness element (task-specific, exploration, prospective) was necessary to produce good swarm behaviors. Moreover, the diversity function described in the next section was also necessary to produce acceptable behaviors across the different task types.

3.6.2 Genome Selection

When agents have enough genomes, the fitness of each genome is computed using the task, exploration, and prospective fitnesses described above. The most fit 50% of the agent's genomes are then selected to be parents. Crossover is performed to create a new population of genomes, after which the parents are discarded. Mutation is then applied to enlarge the population. And then a single genome is selected based on diversity and the agent follows the corresponding BT program phenotype.

Note that it is impossible for the agent to know for sure which of its genomes are the most fit. The only way to test a genome is to use the corresponding BT program phenotype 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.

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 [25, 61]. The primary purpose of promoting diversity in this paper is to contribute to what is called the diversity of "types and kinds" [54], 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.

Denote the diversity heuristic by D where $D : \{\text{Phenotypes}\} \rightarrow \mathbb{R}$. The **diversity** function takes a BT as input and extracts all the nodes from the tree. Recall that for the BNF grammar used in this report, the nodes can have only (a) "Sequence" and "Selector" controls and (b) primitive 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 swarm grammar.

3.7 Experiments and Results

This section describes the evolved collective behaviors for different swarm tasks and presents quantitative results for the four 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 to 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, average results are shown in a graph with a solid green line surrounded by a red region that represents ± one standard deviation. All the values in the y-axis are scaled using natural logarithm. 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. The leftmost portion shows a section of one of the learned behavior tree and the rightmost portion shows a section of the hand-coded behavior tree for the particular task.

3.7.1 Foraging

In the *single-source foraging* problem, the swarm's task is to collect food from a *source* region in the environment and bring the food to the *hub* [68]. A *source* can be located anywhere within a pre-defined bounded area except for within a fixed distance from the hub and has a fixed number of food units available. A single agent can take only one food unit per visit. Agents carry the food units from source to hub, where the food is stored. Agents do not have prior information regarding the source location. Agents can make several trips between source and hub.

When there are multiple *sources* of food in the environment, the foraging problem is called *multi-source foraging*. The multiple sources can have different sizes and quantities of

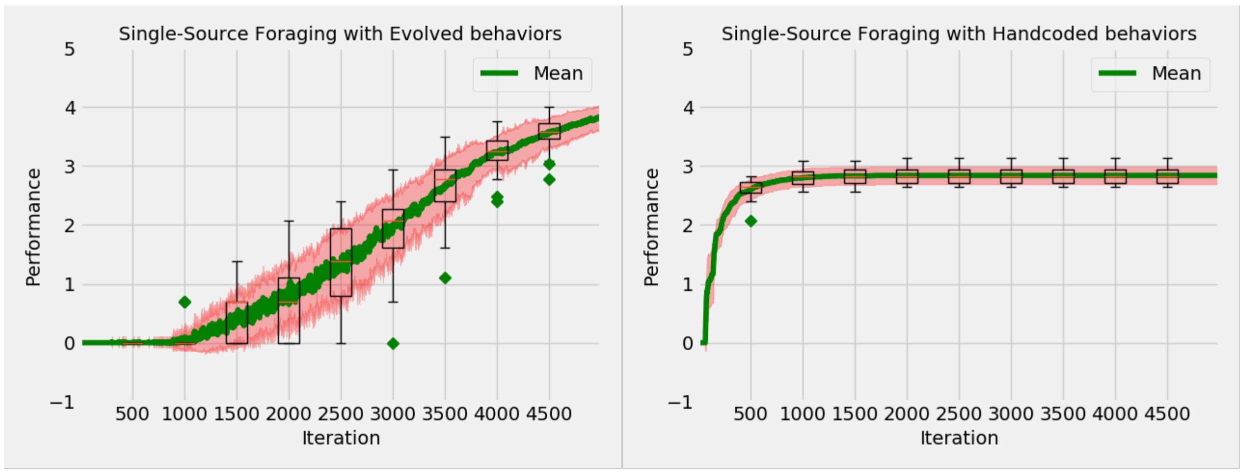

Figure 3.1: Empirical results on single source foraging problem.

food. In multi-source foraging, agents in the swarm need to exploit the multiple food source locations to maximize the food collected at the hub.

To show that the evolved behaviors are robust across environments, we apply behaviors evolved on the single-source foraging problem to the multi-source foraging problem. In each foraging problem, the objective for the swarm is the same: to collect the maximum amount of food in the hub. The only difference between the foraging problems is the number of the sources and where they are at in the world. When behaviors evolved in one environment performing well in another, qualitatively different environment implies robustness.

Single Source Foraging

Figure 3.1 shows the average performance of swarm for Single-Source Foraging. The left portion shows the learned behaviors and the right portion shows the results for a hand-coded BT solution. The evolved behaviors clearly perform better than the hand-coded behaviors.

It takes about 3000 iterations before agents start gathering food in the hub. The reason behind the slow start is that the agents need to explore the environment first to find the location of the source. Since there is just a single source it takes agents some time to find the source and communicate the information of the source. In contrast, the hand-coded behavior was able to collect food right before 500 iterations. After analyzing the BT for both behaviors in figure 3.1, we observe that the exploration node is independent of other nodes in hand-coded BT whereas in evolved BT, the explore node is dependent on another composite node. The independence of explore node from other nodes allows the hand-coded BT to explore the environment much faster than the evolved BT.

Subjective observations, to be quantified in future work, indicate that the diversity of agents in the evolved agents is one reason for their success; 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, there is cyclic pattern of visiting hub and other objects of interest.

Multi-Source Foraging

Figure 3.2 shows how well behaviors evolved for single-source foraging work on the multi-source foraging problem. Two observations are apparent. First, since there are multiple sources, the time to discover one of the sources is significantly lower than in single source problem. Second, even though the problem is different, the single-source learners can succeed on the multi-source problem, though one would anticipate higher performance if the agents could specialize in the multi-source foraging task. A slightly lower performance of the evolved behavior in the multi-source foraging problem is due to the variability in size of the food source in the environment.

3.7.2 Cooperative Transport

Cooperative transport arises when a group of agent works together to carry a large object and form a consensus on a travel direction [39]. Since a single agent doesn't have the resources to carry the large object, it needs to cooperate with others to move the object from a start location to a goal location. Without loss of generality, the object is treated as food and the

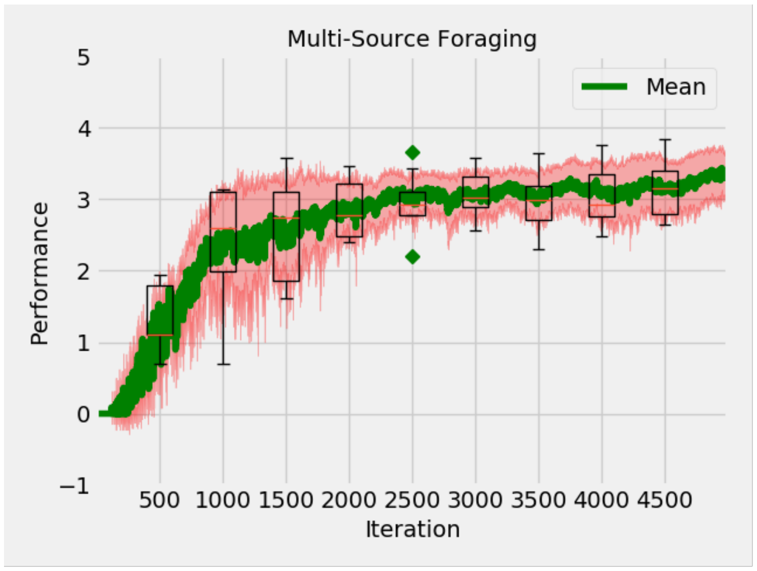

Figure 3.2: Empirical results on multiple source foraging problem. The behaviors evolved for single-source foraging problem was used to run this experiment.

goal location is the hub. Results are shown for a world in which each agent can carry 10 food units, but when each food object weighs more than 10 units.

Figure 3.3 show the performance of the evolved behaviors for cooperative transport. The figure has box-and-whiskers elements, but the quartiles and 90% variation bound all lie on the same line. Only outliers appear. Almost every experiment succeed but one experiment run failed. Thus, all outliers yield a score of zero.

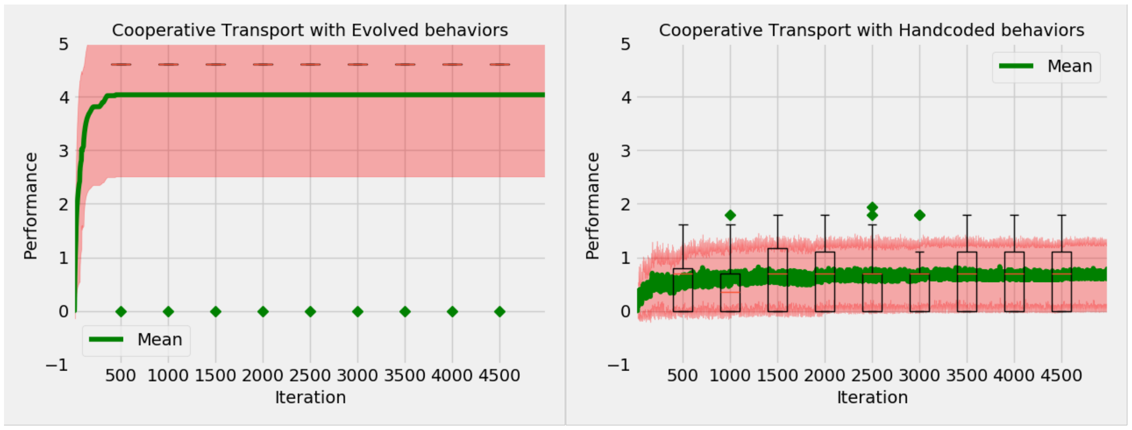

Figure 3.3: Empirical results on cooperative transport problem.

Comparing evolved behaviors (left) with hand-coded behaviors (right) in Figure 3.3. As with single-source foraging, subjective observations indicate that the homogeneous nature of the hand-coded population interfered with the swarm's ability to succeed.

3.7.3 Nest Maintenance

The nest maintenance task is to remove debris near the hub, which is a well-known behavior for ant colonies [23]. The debris was distributed around the hub and the agent task is to move the debris away from the hub.

Figure 3.4 shows the performance of the evolved behaviors in the nest-maintenance task. With just 1000 iterations, the swarm was successful in removing all the debris from the hub. Towards the final iterations, almost all the experiment run converges to the solution.

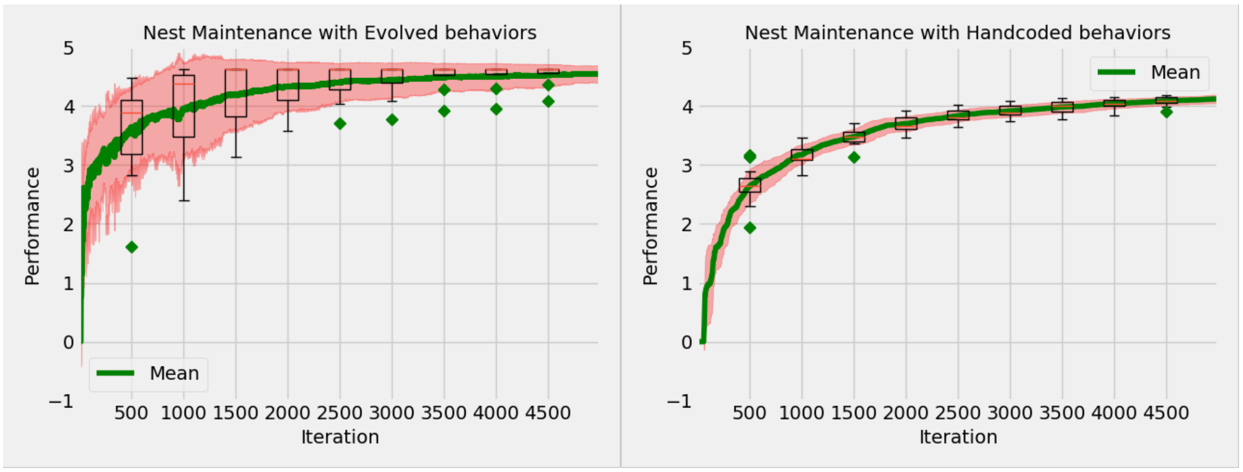

Figure 3.4: Empirical results on nest-maintenance task.

Comparing behaviors for the evolved (left) to the right (hand-coded) behaviors in Figure 3.4 again shows that the evolved behaviors performed better than the hand-coded behaviors. One of the causes of the bad performance of hand-coded behavior is, again, due to homogeneous agents.

3.8 Future Work

The framework 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. The framework has been tested only on spatial colony tasks, and it would be interesting to test the framework for classification, regression and NLP related problems. Finally, the effect of behavioral diversity and the diversity heuristic on the quality and resilience of swarm behavior needs to be studied, using, for example Page's taxonomy of diversity [54].

3.9 Conclusions

Results show that a recursively defined BT-based grammar, built from common agent behaviors, 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 reasonable time. When agents evolved for single-source foraging were put into a world with multiple sources (of different sizes and locations), the agents were still able to succeed, indicating a type of robustness.

Two things indicate that the diversity of the evolved behaviors was essential to their success. First, evolved behaviors outperformed homogeneous hand-coded behaviors. Second, when a swarm was constructed by using a collection of the top-performing type of evolved agent, the swarm fails to succeed in most of the tasks. However, when a diverse set of evolved agents are used in the tests, the swarm succeeds in the task. The evolved BT are readable such that looking at BT graph provides us the information about agent decisions.

Chapter 4

Summary and Future Work

4.1 Summary

In this thesis, we developed a framework that embodied Agent-Based Modeling (ABM), Grammatical Evolution algorithm for Evolution of Swarm bEhaviors (GEESE), and Behavior Trees (BTs). The effectiveness of GEESE was first demonstrated on the Santa Fe Trail problem, outperforming the state of the art in terms of minimum steps to solve the problem. Additionally, GEESE was used to evolve individual behaviors that lead to successful colony-level foraging, outperforming behaviors evolved by conventional grammatical evolution as well as hand-coded individual behaviors.

The thesis provided evidence in support of the following claims: 1) GEESE produces high-quality collective behaviors more effectively than centralized GEs, at least for some problems; 2) behaviors learned via GEESE can be reused across a set of similar problems (a type of robustness); and 3) the same grammar can be tuned for dissimilar collective behavior (a breadth claim).

Results show that a recursively defined BT-based grammar, built from common agent behaviors, 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 were added to the fitness function to find solutions in reasonable time. When agents evolved for single-source foraging were put into a world with multiple sources (of different sizes and locations), the agents were still able to succeed, indicating a type of robustness.

Two things indicate that the diversity of the evolved behaviors was essential to their success. First, evolved behaviors outperformed homogeneous hand-coded behaviors. Second, when a swarm was constructed by using a collection of the top-performing type of evolved agent, the swarm fails to succeed in most of the tasks. However, when a diverse set of evolved agents are used in the tests, the swarm succeeds in the task. The evolved BTs were readable, meaning that looking at a BT graph provided information about agent decisions.

This work has three main contributions, namely: a general purpose swarm framework with GE and BT capabilities; a carefully designed BNF grammar, which can be used to evolve a diverse set of behaviors; and a set of bootstrapped fitness functions inspired by the concepts of diversity. Results provide evidence in support of the claim that the framework successfully addressed the research questions presented in Section 1.2.

4.2 Future Work

The GEESE algorithm was able to evolve behaviors for different multi-agent tasks that performed well in simulations, but the algorithm should be applied to physical robots. Physical robots present important problems that are difficult to simulate, like collisions, sensors failures, and actuator failures. This future work should explore the probabilistic modeling of agent's sensing and acting capabilities. Moreover, future work show explore the properties of interference with respect to agent size, obstacle density, and communication failures.

Future work should also explore how the grammar can be modified to evolve other swarm tasks. For example, insect colonies are known to perform construction tasks, to perform defense-related activities, to farm and hunt, and to raid other colonies. Other animal behaviors might also be amenable to grammatical evolution, like pack hunting, joining or leaving a herd, and migration.

The framework has been tested only on spatial colony tasks, and it would be interesting to test the framework for decision-making tasks such as classification, regression, more general consensus decision-making, and possibly even NLP-related problems.

Finally, the effect of behavioral diversity and the diversity heuristic on the quality and resilience of swarm behavior needs to be studied, using, for example, Page's taxonomy of diversity [54].

Appendix A

Swarm Framework

The swarm framework enables three capabilities. First, the framework allows a user to quickly create, evolve, and test swarm-based behaviors using built-in core components (spatial grids, agent scheduler, evolution algorithms, and BTs) or customized implementation. Second, the framework allows behaviors to be visualized using a browser-based interface. Third, the framework provides data analysis tools that allow collective behaviors to be analyzed.

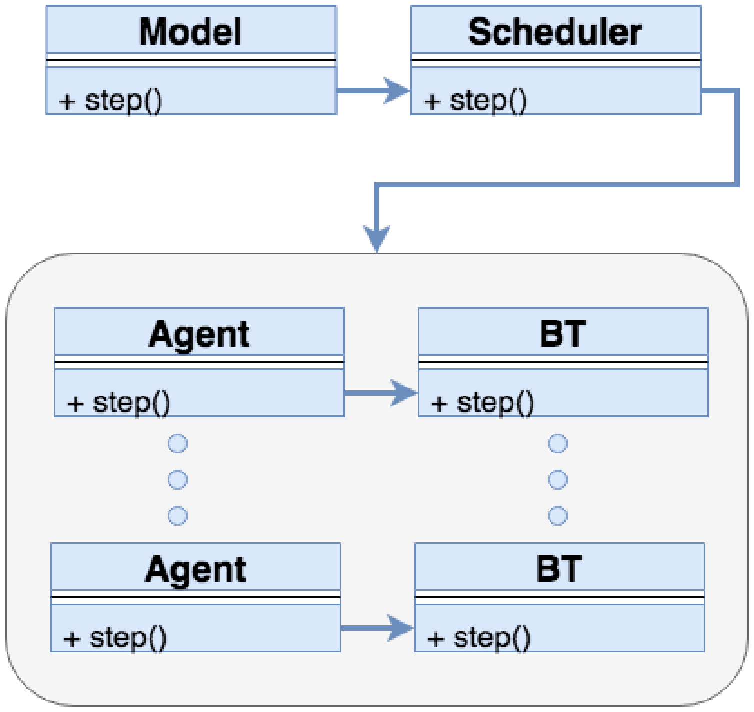

Figure A.1: Control flow diagram of the framework. This figure describes a simplified version of the control flow between different modules in the framework.

Figure A.1 shows the control flow between the different modules in the framework. The length of each experiment run, denoted as an epoch, is defined as the number of calls of the *step* function of the *model* module. The scheduler step function is called for each step function call of the model module. The scheduler modules determine how to activate all the agents in the environment. The scheduler activates the step function of the agent module, which in-turn activates the BT program of each agent.

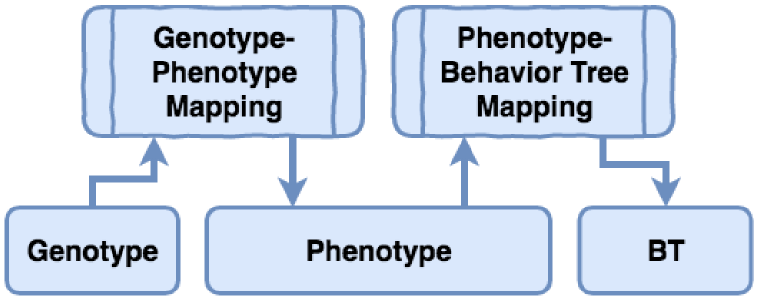

Figure A.2: Five major components of an agent in swarm framework. Genotype is the array of numbers. A function maps the genotype to phenotype and then the phenotype is mapped to a behavior tree.

The valid programs in this framework are also valid XML documents. Encoding phenotypes as XML documents has two purposes: first, it is human readable, and second, it can be easily parsed into a tree structure. The phenotype then can be used by a *Phenotype to Behavior Tree* mapping process to create an executable Behavior Tree. The mapping process between Phenotype and Behavior Tree involves a recursive function to convert all the XML tags and their associated attributes into the control flow nodes and execution nodes. The conversion process should convert the XML document (phenotype) into a valid Behavior Tree instance. Then the BT instance can be executed by passing ticks through the directed tree. The XML tags in the BNF grammar are not shown to make it readable. For the grammar with XML tags, please refer to the project source[1].

Figure A.3 describes the agent initialization process in the framework and Figure A.4 describes the schematic diagram of the framework. The initialization block on the top-left of Figure A.4 represents the process from Figure A.3.

[1]`https://github.com/aadeshnpn/swarm/blob/master/grammars/bt.bnf`

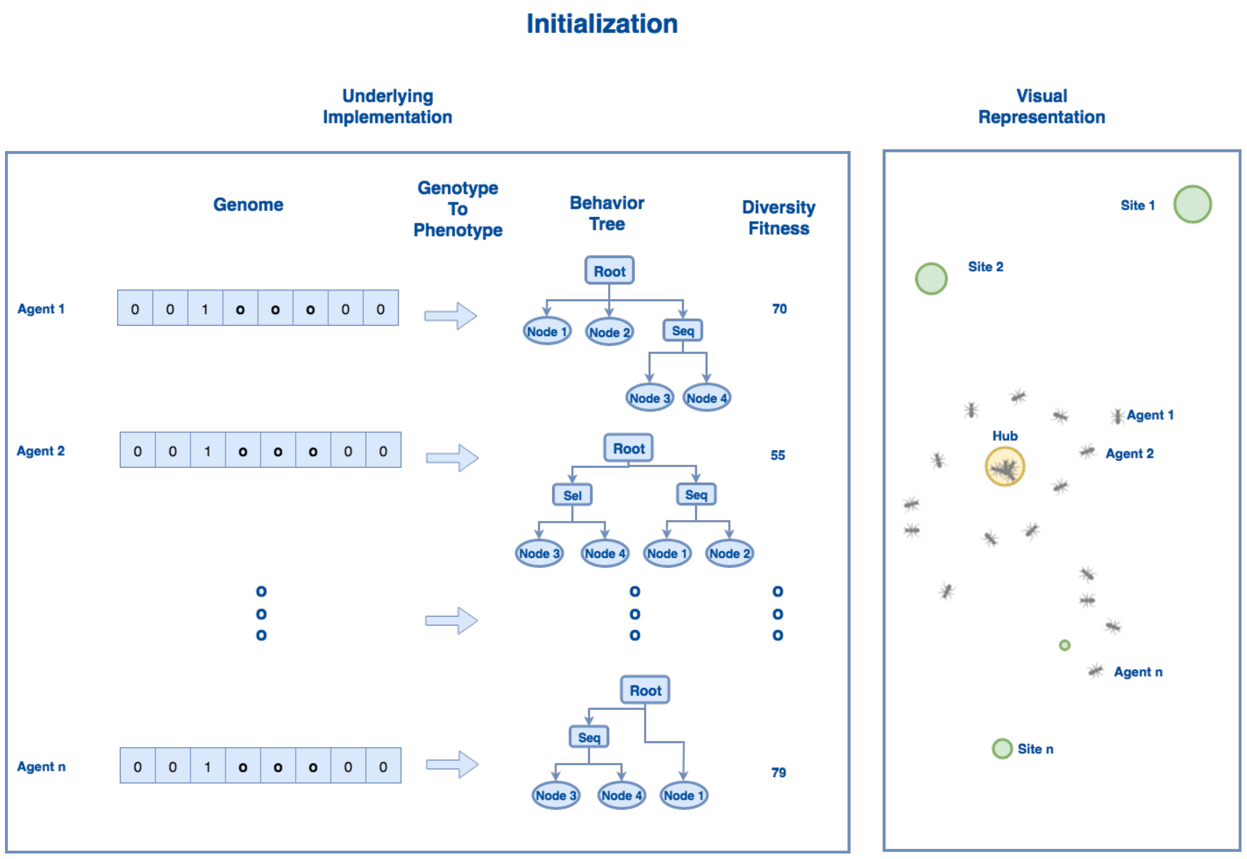

Figure A.3: Initialization process overview in the swarm framework.

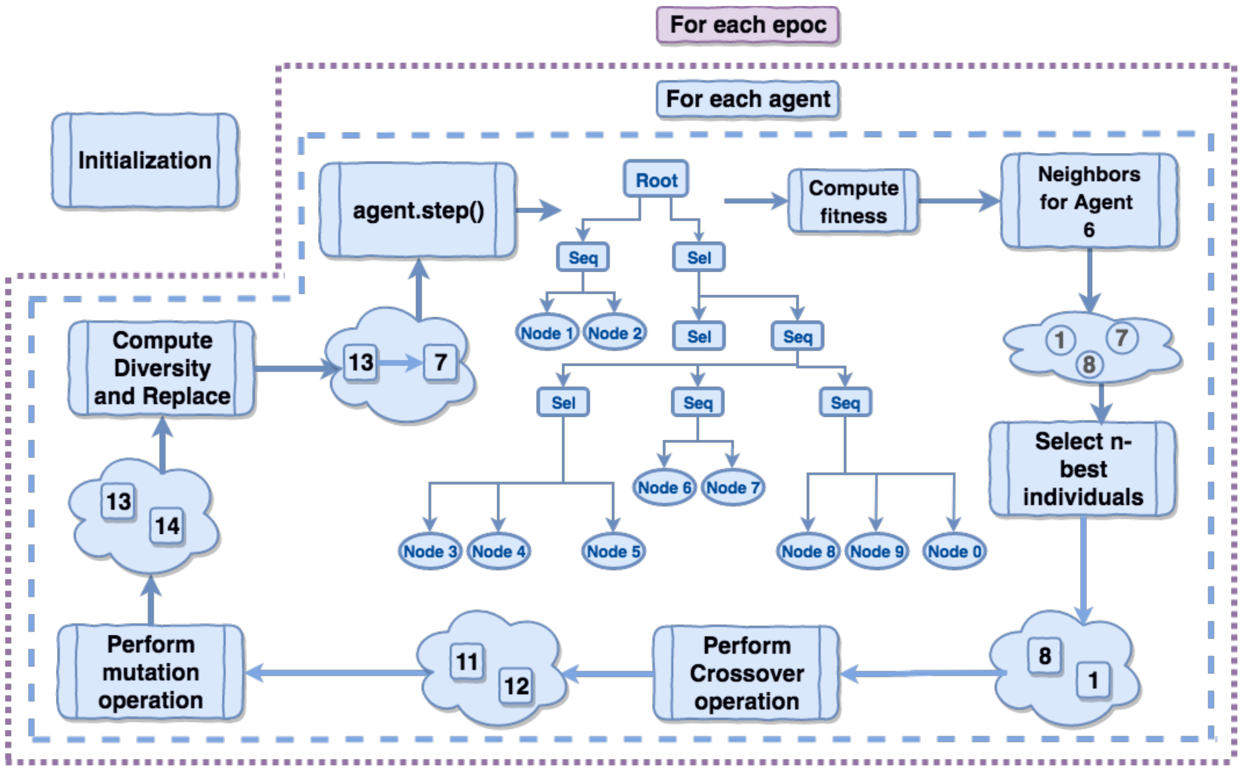

Figure A.4: Schematic diagram of the framework.

Appendix B

Hand-Coded Behavior Trees

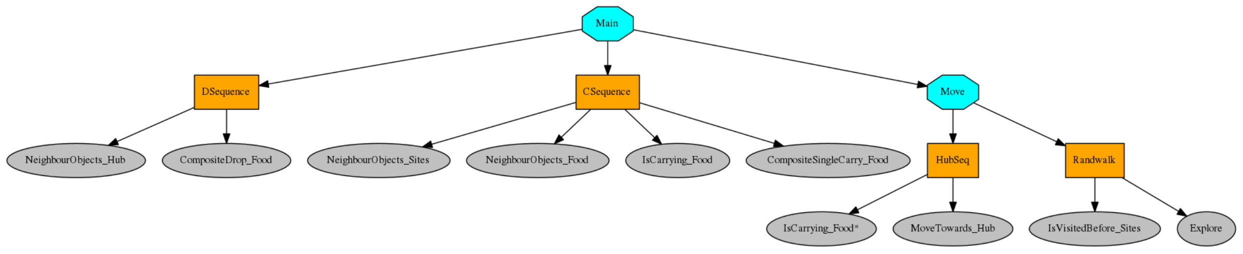

Figure B.1: Hand-coded Behavior Tree for single-source foraging task.

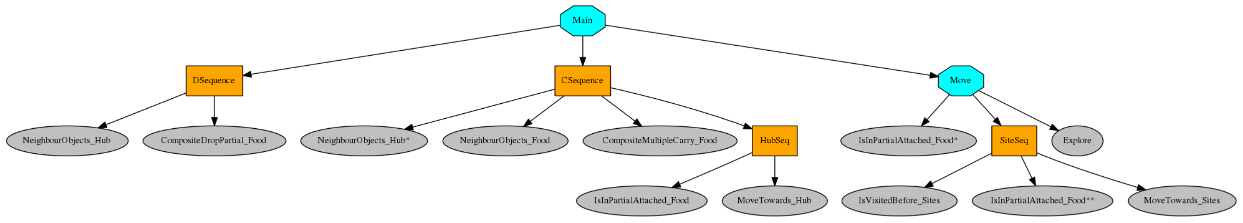

Figure B.2: Hand-coded Behavior Tree for cooperative transport task.

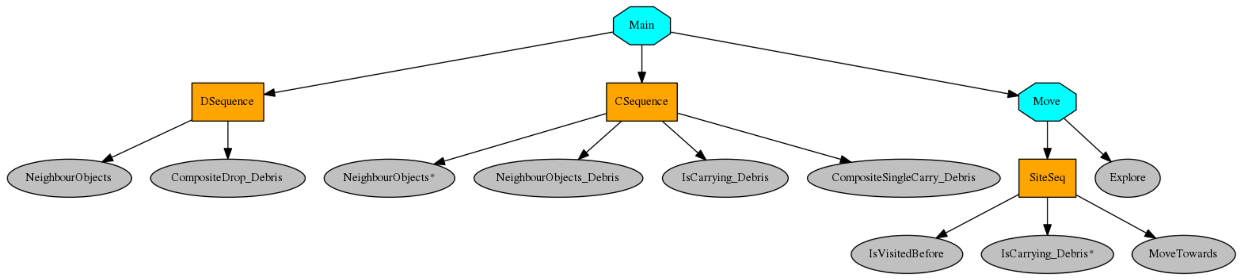

Figure B.3: Hand-coded Behavior Tree for nest maintenance task.

Acronyms

ABM	Agent-Based Modeling. 51
BNF	Backus–Naur Form. ii, 3, 52
BT	Behavior Tree. ii, 2, 3, 52
dEE	distributed Embodied Evolution. 2
GE	Grammatical Evolution. ii, 3, 51, 52
GEESE	Grammatical Evolution algorithm for Evolution of Swarm bEhaviors. ii, 2–4, 51

References

[1] Adrian K Agogino and Kagan Tumer. Unifying temporal and structural credit assignment problems. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 980–987. IEEE Computer Society, 2004.

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

[3] Daniel S Brown, Michael A Goodrich, Shin-Young Jung, and Sean Kerman. Two invariants of human-swarm interaction. *Journal of Human-Robot Interaction*, 5(1):1–31, 2016.

[4] Robert Burbidge and Myra S Wilson. Vector-valued function estimation by grammatical evolution for autonomous robot control. *Information Sciences*, 258:182–199, 2014.

[5] Scott Camazine, Jean-Louis Deneubourg, Nigel R Franks, James Sneyd, Eric Bonabeau, and Guy Theraula. *Self-organization in biological systems*, volume 7. Princeton University Press, 2003.

[6] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.

[7] Li Chen, Chih-Hung Tan, Shuh-Ji Kao, and Tai-Sheng Wang. Improvement of remote monitoring on water quality in a subtropical reservoir by incorporating grammatical evolution with parallel genetic algorithms into satellite imagery. *Water Research*, 42 (1-2):296–306, 2008.

[8] Daniil Chivilikhin and Vladimir Ulyantsev. Muacosm: A new mutation-based ant colony optimization algorithm for learning finite-state machines. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pages 511–518. ACM, 2013.

[9] Dave Cli, Philip Husbands, and Inman Harvey. Evolving visually guided robots. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 374–383. MIT Press/Bradford Books, 1993.

[10] Michele Colledanchise and Petter 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.

[11] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and Al: An Introduction*. CRC Press, 2018.

[12] Matthew F Copeland and Douglas B Weibel. Bacterial swarming: A model system for studying dynamic self-assembly. *Soft matter*, 5(6):1174–1187, 2009.

[13] Iain D Couzin, Jens Krause, Richard James, Graeme D Ruxton, and Nigel R Franks. Collective memory and spatial sorting in animal groups. *Journal of theoretical biology*, 218(1):1–11, 2002.

[14] Jacob W Crandall, Nathan Anderson, Chace Ashcraft, John Grosh, Jonah Henderson, Joshua McClellan, Aadesh Neupane, and Michael A Goodrich. Human-swarm interaction as shared control: Achieving flexible fault-tolerant systems. In *International Conference on Engineering Psychology and Cognitive Ergonomics*, pages 266–284. Springer, 2017.

[15] Stephane Doncieux, Nicolas Bredeche, Jean-Baptiste Mouret, and Agoston E Gusz Eiben. Evolutionary robotics: What, why, and where to. *Frontiers in Robotics and AI*, 2:4, 2015.

[16] Marco Dorigo, Gianni Di Caro, and Luca M Gambardella. Ant algorithms for discrete optimization. *Artificial life*, 5(2):137–172, 1999.

[17] Miguel Duarte, Vasco Costa, Jorge Gomes, Tiago Rodrigues, Fernando Silva, Sancho Moura Oliveira, and Anders Lyhne Christensen. Evolution of collective behaviors for a real swarm of aquatic surface robots. *PloS One*, 11(3):e0151834, 2016.

[18] Russell Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43. IEEE, 1995.

[19] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Michael O'Neill, and Erik Hemberg. Ponyge2: Grammatical evolution in python. *CoRR*, abs/1703.08535, 2017. URL `http://arxiv.org/abs/1703.08535`.

[20] Eliseo Ferrante, Edgar Duéñez-Guzmán, Ali Emre Turgut, and Tom Wenseleers. GESwarm: Grammatical evolution for the automatic synthesis of collective behaviors in swarm robotics. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pages 17–24. ACM, 2013.

[21] Loukas Georgiou and William J Teahan. Constituent grammatical evolution. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1261, 2011.

[22] Michael A Goodrich, Sean Kerman, and Shin-Young Jun. On leadership and influence in human-swarm interaction. In *AAAI Fall Symposium: Human Control of Bioinspired Swarms*, 2012.

[23] Deborah M Gordon. The dynamics of the daily round of the harvester ant colony (pogonomyrmex barbatus). *Animal Behaviour*, 34(5):1402–1419, 1986.

[24] Deborah M Gordon. *Ants at work: How an insect society is organized.* Simon and Schuster, 1999.

[25] Deborah M Gordon. *Ant encounters: Interaction networks and colony behavior.* Princeton University Press, 2010.

[26] Jonatan Hugosson, Erik Hemberg, Anthony Brabazon, and Michael O'Neill. Genotype representations in grammatical evolution. *Applied Soft Computing*, 10(1):36–43, 2010.

[27] D Isla. Handling complexity in the Halo 2 AI, 2005. *URL: http://www. gamasutra. com/gdc2005/features/20050311/isla _01. shtml [21.1. 2010]*, 2010.

[28] David Jefferson, R Collins, C Cooper, M Dyer, M Flowers, R Korf, C Taylor, and A Wang. *The genesys/tracker system.* Reading, MA: Addison-Wesley, 1992.

[29] Simon Jones, Matthew Studley, Sabine Hauert, and Alan Winfield. Evolving behaviour trees for swarm robotics. In *Distributed Autonomous Robotic Systems*, pages 487–501. Springer, 2018.

[30] Dervis Karaboga. An idea based on honey bee swarm for numerical optimization. Technical report, Erciyes University, Engineering Faculty, Computer Engineering Department, 2005.

[31] Muhammad Rezaul Karim and Conor Ryan. Sensitive ants are sensible ants. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, pages 775–782. ACM, 2012.

[32] Yael Katz, Kolbjørn Tunstrøm, Christos C Ioannou, Cristián Huepe, and Iain D Couzin. Inferring the structure and dynamics of interactions in schooling fish. *Proceedings of the National Academy of Sciences*, 108(46):18720–18725, 2011.

[33] Spyros A Kazarlis, AG Bakirtzis, and Vassilios Petridis. A genetic algorithm solution to the unit commitment problem. *IEEE Transactions on Power Systems*, 11(1):83–92, 1996.

[34] Sean Kerman, Daniel Brown, and Michael A Goodrich. Supporting human interaction with robust robot swarms. In *5th International Symposium on Resilient Control Systems (ISRCS)*, pages 197–202. IEEE, 2012.

[35] Lukas König, Sanaz Mostaghim, and Hartmut Schmeck. Decentralized evolution of robotic behavior using finite state machines. *International Journal of Intelligent Computing and Cybernetics*, 2(4):695–723, 2009.

[36] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[37] David Kriesel. Evolutionary synthesis of collective behavior. In *CollSec*, 2010.

[38] David MM Kriesel, Eugene Cheung, Metin Sitti, and Hod Lipson. Beanbag robotics: Robotic swarms with 1-dof units. In *International Conference on Ant Colony Optimization and Swarm Intelligence*, pages 267–274. Springer, 2008.

[39] C Ronald Kube and Eric Bonabeau. Cooperative transport by ants and robots. *Robotics and Autonomous Systems*, 30(1-2):85–101, 2000.

[40] Jonas Kuckling, Antoine Ligot, Darko Bozhinoski, and Mauro Birattari. Behavior trees as a control architecture in the automatic modular design of robot swarms. In *International Conference on Swarm Intelligence*, pages 30–43. Springer, 2018.

[41] Joel Lehman and Kenneth O Stanley. Efficiently evolving programs through the search for novelty. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, pages 837–844. ACM, 2010.

[42] Naomi Ehrich Leonard and Edward Fiorelli. Virtual leaders, artificial potentials and coordinated control of groups. In *Proceedings of the 40th IEEE Conference on Decision and Control*, volume 3, pages 2968–2973. IEEE, 2001.

[43] M Anthony Lewis, Andrew H Fagg, and Alan Solidum. Genetic programming approach to the construction of a neural network for control of a walking robot. In *IEEE International Conference on Robotics and Automation*, pages 2618–2623. IEEE, 1992.

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

[45] Lisa Margonelli. *Underbug: An Obsessive Tale of Termites and Technology.* Scientific American, 2018.

[46] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. Towards a unified behavior trees framework for robot control. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5420–5427. IEEE, 2014.

[47] Mehran Mesbahi and Magnus Egerstedt. *Graph theoretic methods in multiagent networks.* Princeton University Press, 2010.

[48] Julian F Miller and Peter Thomson. Cartesian genetic programming. In *European Conf. on Genetic Programming*, pages 121–132. Springer, 2000.

[49] Aadesh Neupane. Swarm. `https://github.com/aadeshnpn/swarm.git`, 2018.

[50] Aadesh Neupane, Michael A Goodrich, and Eric G Mercer. Geese: grammatical evolution algorithm for evolution of swarm behaviors. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 999–1006. ACM, 2018.

[51] Aadesh Neupane, Michael A Goodrich, and Eric G Mercer. Geese: Grammatical evolution algorithm for evolution of swarm behaviors. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2025–2027. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[52] Michael O'Neil and Conor Ryan. Grammatical evolution. In *Grammatical Evolution*, pages 33–47. Springer, 2003.

[53] Michael O'Neill and Anthony Brabazon. Grammatical swarm: The generation of programs by social programming. *Natural Computing*, 5(4):443–462, 2006.

[54] Scott E Page. *Diversity and complexity.* Princeton University Press, 2010.

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

[56] Agnes Pintér-Bartha, Anita Sobe, and Wilfried Elmenreich. Towards the light—comparing evolved neural network controllers and finite state machine controllers. In *Proceedings of the Tenth Workshop on Intelligent Solutions in Embedded Systems (WISES)*, pages 83–87. IEEE, 2012.

[57] Chris R Reid, Tanya Latty, Audrey Dussutour, and Madeleine Beekman. Slime mold uses an externalized spatial “memory” to navigate in complex environments. *Proceedings of the National Academy of Sciences*, 109(43):17490–17494, 2012.

[58] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[59] Conor Ryan, JJ Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*, pages 83–96. Springer, 1998.

[60] KYW Scheper. Behaviour trees for evolutionary robotics: Reducing the reality gap. Masters Thesis, 2014.

[61] Thomas D Seeley. *The wisdom of the hive: The social physiology of honey bee colonies*. Harvard University Press, 2009.

[62] Thomas D Seeley and Susannah C Buhrman. Nest-site selection in honey bees: How well do swarms implement the “ best-of-n” decision rule? *Behavioral Ecology and Sociobiology*, 49(5):416–427, 2001.

[63] Yehonatan Shichel, Eran Ziserman, and Moshe Sipper. Gp-robocode: Using genetic programming to evolve robocode players. In *European Conference on Genetic Programming*, pages 143–154. Springer, 2005.

[64] Fernando Silva, Paulo Urbano, Sancho Oliveira, and Anders Lyhne Christensen. odNeat: An algorithm for distributed online, onboard evolution of robot behaviours. *Artificial Life*, 13:251–258, 2012.

[65] Léo FDP Sotto, Vinícius V de Melo, and Márcio P Basgalupp. An improved λ-linear genetic programming evaluated in solving the santa fe ant trail problem. In *Proc. of the 31st Annual ACM Symp. on Applied Computing*, pages 103–108. ACM, 2016.

[66] Forrest Stonedahl, William M Rand, and Uri Wilensky. Multi-agent learning with a distributed genetic algorithm. *Proceedings of Autonomous Agents and Multi-Agents Systems (AAMAS) ALAMAS + ALAg Workshop*, 2008.

[67] Daniel Stonier. Py_trees. `https://github.com/stonier/py_trees.git`, 2018.

[68] John H Franks Sudd, Nigel R John H Sudd, and Nigel R Franks. The behavioural ecology of ants. Technical report, University of Hull, 1987.

[69] P Sujit, Joel George, and Randy Beard. Multiple UAV task allocation using particle swarm optimization. In *AIAA Guidance, Navigation and Control Conference and Exhibit*, page 6837, 2008.

[70] David JT Sumpter. *Collective animal behavior*. Princeton University Press, 2010.

[71] John Mark Swafford, Erik Hemberg, Michael O'Neill, Miguel Nicolau, and Anthony Brabazon. A non-destructive grammar modification approach to modularity in grammatical evolution. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 1411–1418. ACM, 2011.

[72] Katia Sycara, Anandeep Pannu, M Willamson, Dajun Zeng, and Keith Decker. Distributed intelligent agents. *IEEE Expert*, 11(6):36–46, 1996.

[73] Paulo Urbano and Loukas Georgiou. Improving grammatical evolution in santa fe trail using novelty search. In *European Conference on Artificial Life*, pages 917–924, 2013.

[74] J-P Vacher, Thierry Galinho, Franck Lesage, and Alain Cardon. Genetic algorithms in a multi-agent system. In *Proceedings., IEEE International Joint Symposia on Intelligence and Systems*, pages 17–26. IEEE, 1998.

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

[76] Csaba Virágh, Gábor Vásárhelyi, Norbert Tarcai, Tamás Szörényi, Gergő Somorjai, Tamás Nepusz, and Tamás Vicsek. Flocking algorithm for autonomous flying robots. *Bioinspiration & Biomimetics*, 9(2):025012, 2014.