GEESE: Grammatical Evolution Algorithm for Evolution of Swarm Behaviors

Aadesh Neupane
Brigham Young University
Provo, Utah
aadeshnpn@byu.edu

Michael A. Goodrich
Brigham Young University
Provo, Utah
mike@cs.byu.edu

Eric G. Mercer
Brigham Young University
Provo, Utah
egm@cs.byu.edu

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.

CCS CONCEPTS

• **Computing methodologies** → **Genetic algorithms**; *Multi-agent systems*; • **Computer systems organization** → *Evolutionary robotics*;

KEYWORDS

Swarms, Grammatical Evolution

ACM Reference Format:
Aadesh Neupane, Michael A. Goodrich, and Eric G. Mercer. 2018. GEESE: Grammatical Evolution Algorithm for Evolution of Swarm Behaviors. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205619

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 [5]; Ant colonies have evolved collective behaviors for foraging, nest defense, path planning and construction [11]; Bees are able to select best sites among many good sites at their disposal [27], and Fish are able to avoid predators by organizing themselves in collective shapes that deter predation [16].

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 [20, 21, 36]. 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 [6, 7, 14]. 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 [17].

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 [17]. 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 [23]. 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.

GECCO '18, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5618-3/18/07…$15.00
https://doi.org/10.1145/3205455.3205619

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 BACKGROUND

There are many papers that "program" agents using genetic algorithms, but Yehonatan et al. [28] 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. [1] 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. [24] 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. [3] 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. [9] developed a framework that could automatically synthesize collective behaviors for swarms using GE without using communication behaviors.

Cantu-Paz [2] identified three architectures for parallel genetic algorithms: master-slave, fine-grained, and multiple-population parallel GAs. Vacher et al. [35] described a multi-agent system guided by a multi-objective GA to find a balance point on the Pareto front. Stonedahl [31] described a multi-agent learning algorithm with a distributed GA to solve a bit-matching problem. Silva et. al. [29] developed "odNEAT", a distributed and decentralized neuroevolution algorithm for online learning in groups of autonomous robots that evolve both weights and network topology.

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 [23, 26]. 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 [18]. Details about Santa Fe Trail problem are explained in Section 4.1.

$\langle code \rangle$ ::= $\langle code \rangle$ | $\langle progs \rangle$ (1)

$\langle progs \rangle$ ::= $\langle condition \rangle$ | $\langle prog2 \rangle$ | $\langle prog3 \rangle$ | $\langle op \rangle$ (2)

$\langle condition \rangle$ ::= `if_food_ahead`($\langle progs \rangle$,$\langle progs \rangle$) (3)

$\langle prog2 \rangle$ ::= `prog2`($\langle progs \rangle$,$\langle progs \rangle$) (4)

$\langle prog3 \rangle$ ::= `prog3`($\langle progs \rangle$,$\langle progs \rangle$,$\langle progs \rangle$) (5)

$\langle op \rangle$ ::= `left` | `right` | `move` (6)

Koza gives an in-depth explanation of the above grammar [18].

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

$$RHSRule = c\%r_A \quad (1)$$

where % denotes the modulo operator. After a non-terminal is mapped to one of its right-hand side rules using Equation 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 Santa Fe Trail

The Santa Fe Trail problem [13] 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 [18]. The baseline solution gathers all 89 food units within 543 moves.

3 GEESE

GEESE is loosely inspired by "odNEAT" [29], "Genetic Algorithm for Multi-Agent system" [35], and "GESwarm" [9]. 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" [29] 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.

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 NEIGHBORHOOD(A_j), agent A_j requests each agent $A_i \in$ 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.

Algorithm 1 GEESE

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 ← list(agent,M)
  for *agent* in *agents* **do**
    agent.sense()
    agent.act()
    agent.update()
  end for

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 [8]. 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.

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.

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 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.

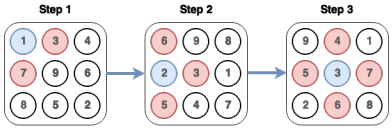

Figure 1: Three conceptual evolution steps in GEESE

The effect of preserving genetic diversity is illustrated in Figure 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 [25].

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.

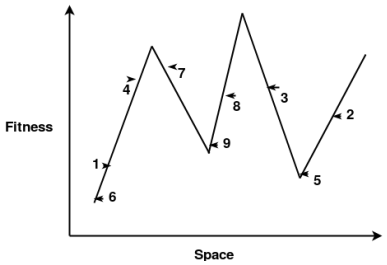

Figure 2: Conceptual fitness landscape with nine agents.

4 EXPERIMENTS

Section 4.1 compares GEESE against GE on the Santa Fe Trail problem [13]. Section 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 [8] 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.

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 2.1.

4.1.1 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 1. Two parameter values from Table 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 1.

Figure 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.

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

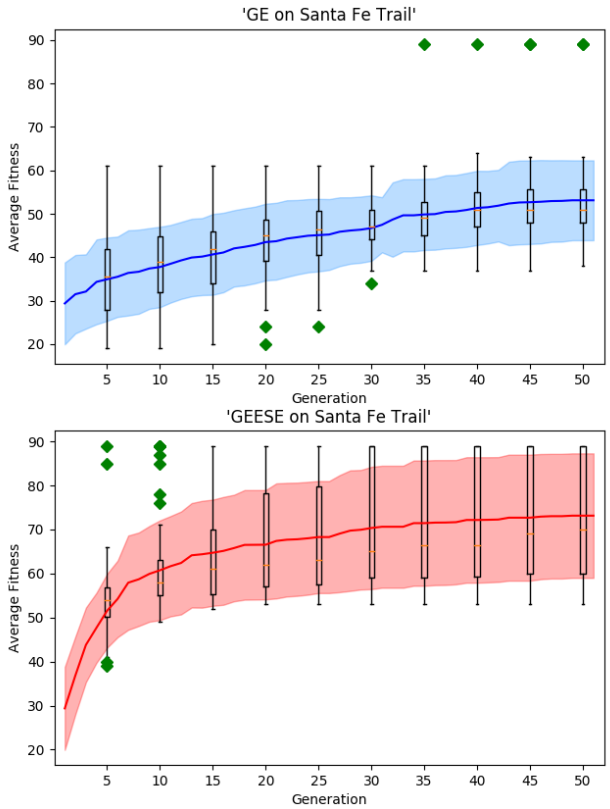

Figure 3: GEESE converges quicker than GE.

4.1.2 Santa Fe Trail using Novelty Search. Lehman et. al. [19] introduce novelty search techniques and Urbano et al. [34] 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 [34]. 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

$$\rho(x) = \frac{1}{k \sum_{y \in \text{NEIGHBORHOOD}(x)} \text{DIST}(x, y)}$$

Urbano et al. [34] 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 1. It is evident from Figure 4 that GEESE converges to the solution faster than standard GE with NS.

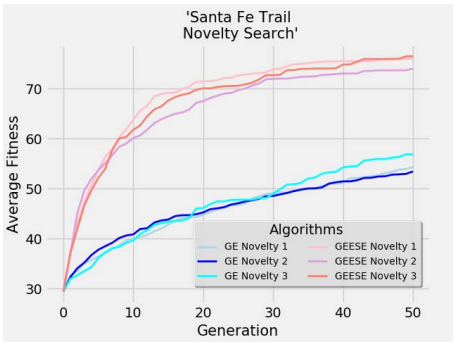

Figure 4: GEESE with NS converges quicker than GE with NS.

One of the programs evolved by GEESE using novelty search is shown in Figure 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 [34].

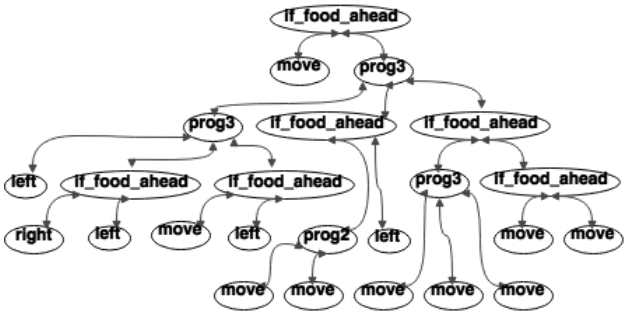

Figure 5: The evolved program that solved the Santa Fe Trail problem is just 324 steps.

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 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.

Algorithms	Mean Fitness	Success Rate	Minimum Steps
Koza GP [18]	N/A	N/A	543
Cartesian GP [22]	N/A	0.93	N/A
MuACOsm [4]	N/A	N/A	394
λ-LGP [30]	89	1	N/A
GE [12]	80.1	0.63	N/A
GE (Repair) [33]	75.02	0.32	N/A
Grammatical Swarm [24]	80.18	0.58	N/A
Attribute Grammar [15]	N/A	0.85	N/A
GE (Novelty) [34]	77.88	0.41	331
GE (Constituent) [10]	N/A	0.9	337
GEESE	**84.1**	**0.6**	**324**

Table 2: Comparison with state-of-the-art methods for Santa Fe Trail. Values not presented in the original work are marked as "N/A"

4.1.3 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 6, when the IP is close to 1, GEESE performs worse than when IP is lower. The reason is that with IP near 1, GEESE degenerates to standard GE. Figure 6 shows performance for IP $\in \{0.2, 0.4, ...0.99\}$.

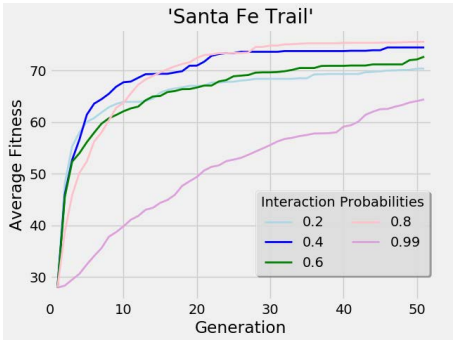

Figure 6: Average fitness for varying interaction probabilities.

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 7 illustrates that the increase in the number of agents enables GEESE to reach the solution in fewer generations.

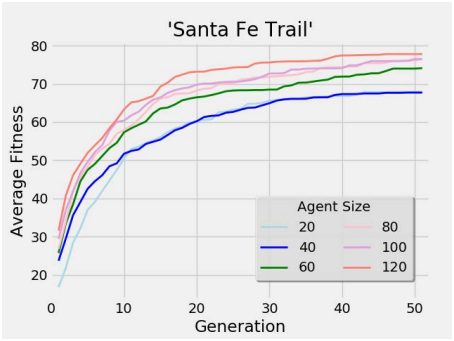

Figure 7: Average fitness with varying number of agents.

4.1.4 Summary. The performance of GEESE was superior to other variants of GE as seen in Table 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.

4.2 Swarm Behavior

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

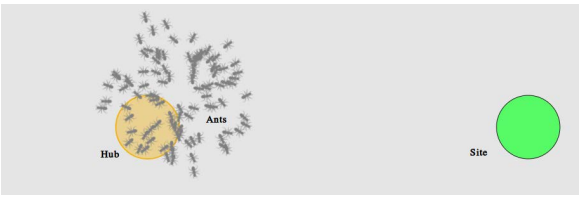

Figure 8: A foraging environment with one hundred agents, a hub, and a single food source.

4.2.1 Problem Description. Figure 8 illustrates a foraging scenario known as the *center place food foraging* problem [32]. 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 (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.

4.2.2 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 4.2.3 presents and describes the grammar.

4.2.3 Grammar. The grammar used for this experiment is:

$$\langle start \rangle ::= \langle ruleset \rangle \quad (1)$$

⟨*ruleset*⟩ ::= ⟨*rule*⟩ | ⟨*rule*⟩ ⟨*ruleset*⟩ (2)
⟨*rule*⟩ ::= ⟨*state*⟩ ⟨*pc*⟩ ⟨*transition*⟩ (3)
⟨*pc*⟩ ::= ⟨*bool*⟩ ⟨*bool*⟩ ⟨*bool*⟩ ⟨*bool*⟩ ⟨*bool*⟩ ⟨*bool*⟩ ⟨*bool*⟩ (4)
⟨*transition*⟩ ::= ⟨*change_state*⟩ | ⟨*change_mem*⟩ (5)
⟨*change_state*⟩ ::= ⟨*prob*⟩ ⟨*state*⟩ (6)
⟨*change_mem*⟩ ::= ⟨*prob*⟩ ⟨*id*⟩ ⟨*bool*⟩ (7)
⟨*state*⟩ ::= randwalk | tosource | tonest | dropcues | pickcues | sendsignals | recsignals (8)
⟨*bool*⟩ ::= false | true | don't_care (9)
⟨*id*⟩ ::= dropfood | wantfood (10)
⟨*prob*⟩ ::= 0.1 | 0.2 | 0.5 | 0.7 | 1.0 (11)
⟨*bool*⟩ ::= true | false (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.

4.2.4 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 4.1.3, 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.

4.2.5 Results. We created a hand-coded benchmark for comparison. The benchmark consisted a set of 13 rules from the grammar 4.2.3. 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 [11].

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.

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.

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.

ACKNOWLEDGMENTS

This work was supported by ONR grant number N000141613025. All findings, results, and opinions are the responsibility of the authors and do not necessarily reflect the viewpoint of the funding agency.

REFERENCES

[1] Robert Burbidge and Myra S Wilson. 2014. Vector-valued function estimation by grammatical evolution for autonomous robot control. Information Sciences 258 (2014), 182–199.
[2] Erick Cantú-Paz. 1998. A survey of parallel genetic algorithms. Calculateurs paralleles, reseaux et systems repartis 10, 2 (1998), 141–171.
[3] Li Chen, Chih-Hung Tan, Shuh-Ji Kao, and Tai-Sheng Wang. 2008. 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 (2008), 296–306.
[4] Daniil Chivilikhin and Vladimir Ulyantsev. 2013. MuACOsm: a new mutation-based ant colony optimization algorithm for learning finite-state machines. In Proc. of the 15th annual conf. on Genetic and evolutionary computation. ACM, 511–518.
[5] Matthew F Copeland and Douglas B Weibel. 2009. Bacterial swarming: a model system for studying dynamic self-assembly. Soft matter 5, 6 (2009), 1174–1187.
[6] Marco Dorigo, Gianni Di Caro, and Luca M Gambardella. 1999. Ant algorithms for discrete optimization. Artificial life 5, 2 (1999), 137–172.
[7] Russell Eberhart and James Kennedy. 1995. A new optimizer using particle swarm theory. In Micro Machine and Human Science, 1995. MHS'95., Proc. of the Sixth International Symp. on. IEEE, 39–43.
[8] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Michael O'Neill, and Erik Hemberg. 2017. PonyGE2: Grammatical Evolution in Python. CoRR abs/1703.08535 (2017). http://arxiv.org/abs/1703.08535
[9] Eliseo Ferrante, Edgar Duéñez-Guzmán, Ali Emre Turgut, and Tom Wenseleers. 2013. GESwarm: Grammatical evolution for the automatic synthesis of collective behaviors in swarm robotics. In Proc. of the 15th annual conf. on Genetic and evolutionary computation. ACM, 17–24.
[10] Loukas Georgiou and William J Teahan. 2011. Constituent grammatical evolution. In IJCAI Proc. -International Joint Conf. on Artificial Intelligence, Vol. 22. 1261.
[11] Deborah M Gordon. 1999. Ants at work: how an insect society is organized. Simon and Schuster.
[12] Jonatan Hugosson, Erik Hemberg, Anthony Brabazon, and Michael OâĂŹNeill. 2010. Genotype representations in grammatical evolution. Applied Soft Computing 10, 1 (2010), 36–43.
[13] David Jefferson, R Collins, C Cooper, M Dyer, M Flowers, R Korf, C Taylor, and A Wang. 1992. The genesys/tracker system. Reading, MA: Addison-Wesley.
[14] Dervis Karaboga. 2005. An idea based on honey bee swarm for numerical optimization. Technical Report. Technical report-tr06, Erciyes university, engineering faculty, computer engineering department.
[15] Muhammad Rezaul Karim and Conor Ryan. 2012. Sensitive ants are sensible ants. In Proc. of the 14th annual conf. on Genetic and evolutionary computation. ACM, 775–782.
[16] Yael Katz, Kolbjørn Tunstrøm, Christos C Ioannou, Cristián Huepe, and Iain D Couzin. 2011. Inferring the structure and dynamics of interactions in schooling fish. Proc. of the National Academy of Sciences 108, 46 (2011), 18720–18725.
[17] Spyros A Kazarlis, AG Bakirtzis, and Vassilios Petridis. 1996. A genetic algorithm solution to the unit commitment problem. IEEE trans on power systems 11, 1 (1996), 83–92.
[18] John R Koza. 1992. Genetic programming: on the programming of computers by means of natural selection. Vol. 1. MIT press.
[19] Joel Lehman and Kenneth O Stanley. 2010. Efficiently evolving programs through the search for novelty. In Proc. of the 12th annual conf. on Genetic and evolutionary computation. ACM, 837–844.
[20] Naomi Ehrich Leonard and Edward Fiorelli. 2001. Virtual leaders, artificial potentials and coordinated control of groups. In Decision and Control, 2001. Proc. of the 40th IEEE Conf. on, Vol. 3. IEEE, 2968–2973.
[21] Mehran Mesbahi and Magnus Egerstedt. 2010. Graph theoretic methods in multiagent networks. Princeton University Press.
[22] Julian F Miller and Peter Thomson. 2000. Cartesian genetic programming. In European Conf. on Genetic Programming. Springer, 121–132.
[23] Michael OâĂŹNeil and Conor Ryan. 2003. Grammatical evolution. In Grammatical Evolution. Springer, 33–47.
[24] Michael OâĂŹNeill and Anthony Brabazon. 2006. Grammatical swarm: The generation of programs by social programming. Natural Computing 5, 4 (2006), 443–462.
[25] Stuart Russell and Peter Norvig. 2003. Artificial Intelligence: A Modern Approach. (2003).
[26] Conor Ryan, JJ Collins, and Michael O Neill. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In European Conf. on Genetic Programming. Springer, 83–96.
[27] Thomas D Seeley and Susannah C Buhrman. 2001. Nest-site selection in honey bees: how well do swarms implement the" best-of-N" decision rule? Behavioral Ecology and Sociobiology 49, 5 (2001), 416–427.
[28] Yehonatan Shichel, Eran Ziserman, and Moshe Sipper. 2005. GP-robocode: Using genetic programming to evolve robocode players. In European Conf. on Genetic Programming. Springer, 143–154.
[29] Fernando Silva, Paulo Urbano, Sancho Oliveira, and Anders Lyhne Christensen. 2012. odNEAT: An algorithm for distributed online, onboard evolution of robot behaviours. Artificial Life 13 (2012), 251–258.
[30] Léo FDP Sotto, Vinícius V de Melo, and Márcio P Basgalupp. 2016. 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. ACM, 103–108.
[31] Forrest Stonedahl, William M Rand, and Uri Wilensky. 2008. Multi-agent learning with a distributed genetic algorithm. (2008).
[32] John H Franks Sudd, Nigel R John H Sudd, and Nigel R Franks. 1987. The behavioural ecology of ants. Technical Report.
[33] John Mark Swafford, Erik Hemberg, Michael O'Neill, Miguel Nicolau, and Anthony Brabazon. 2011. A non-destructive grammar modification approach to modularity in grammatical evolution. In Proc. of the 13th annual conf. on Genetic and evolutionary computation. ACM, 1411–1418.
[34] Paulo Urbano and Loukas Georgiou. 2013. Improving Grammatical Evolution in Santa Fe Trail using Novelty Search.. In ECAL. 917–924.
[35] J-P Vacher, Thierry Galinho, Franck Lesage, and Alain Cardon. 1998. Genetic algorithms in a multi-agent system. In Intelligence and Systems, 1998. Proc. ., IEEE International Joint Symposia on. IEEE, 17–26.
[36] Csaba Virágh, Gábor Vásárhelyi, Norbert Tarcai, Tamás Szörényi, Gergő Somorjai, Tamás Nepusz, and Tamás Vicsek. 2014. Flocking algorithm for autonomous flying robots. Bioinspiration & biomimetics 9, 2 (2014), 025012.