April 2018

Reusable Software Components for Multi-Robot Foraging

Tanuj Sane
Worcester Polytechnic Institute

Vanshaj Chowdhary
Worcester Polytechnic Institute

Reusable Software Components for Multi-Robot Foraging

Worcester Polytechnic Institute

A Major Qualifying Project
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science

By:

Joseph Wilder

Tanuj Sane

Vanshaj Chowdhary

Abstract

Swarm intelligence is a rapidly growing area of robotics research that has the potential to reshape traditional approaches in many different fields, including military, agriculture, and medicine. Swarm robotics involves the coordination of large groups of simple robots to accomplish complex, novel behaviors in a decentralized way. However, a lack of widely available development platforms for swarm applications has hindered progress by forcing researchers to recreate previous efforts. In addition, the lack of common platforms makes it impossible to compare existing approaches, which in turn hinders the conception of effective swarm engineering practices. The goal of this MQP is to provide a framework for swarm developers to more easily realize their own projects. The focus of this project is on identifying, programming, and evaluating the common behaviors that compose complex tasks such as foraging. Foraging refers to the food harvesting behaviors of social insects such as ants and bees. In robotics, foraging captures general scenarios such as search-and-rescue and construction material collection, and as such it constitutes a basic benchmark to test a wide variety of algorithms. The final deliverable is a library of software components for 3 different types of foraging behaviors previously studied in the swarm robotics literature. The software components we developed can be easily reused and extended by other developers to realize other foraging algorithms.

Acknowledgements

Our Faculty Advisors

Professor Carlo Pinciroli
Professor George Heineman

NEST Lab

Results in this paper were obtained in part using a high-performance computing system acquired through NSF MRI grant DMS-1337943 to WPI.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

The overriding motivation behind this project is to create modular software libraries that enable future developers to use the Buzz programming language in novel ways. In order to appreciate the benefits of this approach, it is necessary to first understand the basics of swarm robotics and the Buzz language itself.

1.1 Swarm Robotics

1.1.1 Definition

Swarm robotics research is a growing field in which researchers seek to implement collective behavior in a decentralized manner over a group of robots in an environment. Swarm robotics is derived from swarm intelligence research, which seeks to develop algorithms inspired by natural swarm behaviors for such team-based activities as foraging.

1.1.2 Design Considerations

Considerations such as complexity, scalability, and robustness are often important aspects of robotic swarm design. Much like the swarms observed in nature-- bees, for example -- the individual robots in a swarm are relatively simple and unable to accomplish much on their own. To achieve most goals, these simple robots must cooperate with others in the swarm. By leveraging the combined capabilities of all members of a swarm, these simple robots are able to manifest a swarm intelligence that allows them to execute more complex behaviors.

As mentioned above, robustness is likewise a critical concern in the design of robot swarms. Much like a school of fish or a colony of ants, a robot swarm must be able to withstand the loss of some of its constituents. If a single robot experiences a failure, it should not jeopardize the overall performance of the rest of the swarm. In practice, accomplishing the goal of robustness in robot swarms often means employing a decentralized design methodology. Under this model, the success of the swarm does not depend on a single, highly capable robot to direct others in the swarm. Often, robots in a swarm all have the same capabilities and rely on peer-to-peer communication to propagate information across the entire swarm. This approach improves robustness by removing a single point of failure, thus making the swarm more fault-tolerant.

A similarly important consideration in swarm design is scalability, which refers to the ability of a swarm to maintain consistent performance regardless of how large the swarm itself becomes. A swarm that accomplishes a task on a small scale with relatively few robots should generally be able to achieve the same goal in a larger environment with more robots. A related concept is a system's flexibility, which captures how a swarm is able to cope with different or changing environments. Ideally, a swarm that is flexible will be able to perform consistently in environments with a myriad of different challenges, obstacles, and hazards.

These basic aspects of swarm performance are important to consider in the design of any swarm robotics implementation, such as the one proposed in this project. They also form a vocabulary that allows for the comparison of different strategies and implementations during the design phase of this project, as well providing a convenient metric to use in the evaluation and improvement of the finished system.

1.1.3 The Foraging Problem

The foraging problem describes a scenario where a swarm of robots forages for “food” and returns it to a “nest”. This mimics behavior seen in swarms of actual organisms - for instance, ants - that exhibit some degree of group intelligence. This behavior is not only a common in swarm robotics, but is also essential for other swarm behaviors such as construction.

As one of the most researched scenarios in swarm robotics, there is a wide array of proposed foraging algorithms. Some of these are very environment-specific, requiring particular sensing capabilities and allowing robots to manipulate the environment. Other assumptions are sometimes made about the presence of obstacles and the location or distribution of the “food”. Many scenarios contain a single, centrally-located nest, while others assume the use of multiple nests.

Where possible, the algorithms proposed in this project attempt to minimize assumptions about the hardware capabilities of the robot in the swarm or the exact parameters of the environment. The precise terms of the foraging problem can change depending on the environment and the capabilities of the involved robots, there are generalized approaches that make for efficient foraging.

1.2 Project Motivation

As a relatively new field, swarm robotics research lacks the widely-used frameworks and toolsets seen in other domains. As a result, developers often must build their own solutions themselves. This is an impediment to progress because it compels researchers to solve problems that may have been solved by others previously. Designing complex swarm scenarios can become difficult due to the high overhead required for each project.

This project aims to address this problem for developers of the Buzz programming language by creating a library of foraging behaviors. Despite the stated focus on foraging, the idea is to develop a library that is useful in many different contexts beyond foraging as well. Since foraging is among the most common swarm behaviors encountered in the literature, it makes some sense to choose foraging algorithms as the basis for a hypothetical library. However, many other common swarm scenarios share similarities with the foraging problem, which gives a foraging-centric library additional utility.

A key component of this project is that it includes multiple foraging algorithms with different approaches. Since these algorithms perform differently depending on the circumstances (for example, high or low food density), users might want to use a specific algorithm to suit the environment. Having multiple algorithms gives developers the ability to choose or even combine several different approaches to foraging.

On a different level, individual sub-behaviors in the library can be combined to create novel scenarios. The modular design proposed in this project allows developers to combine not only different foraging algorithms, but also the individual behaviors that comprise those algorithms. This empowers swarm researchers to use the library as a basis for their own projects, and saves them the effort of having to replicate many simple behaviors to implement their own ideas.

1.3 Report Structure

The report that follows this introduction contains several different chapters, each of which details a specific aspect of this project. The next chapter reviews the relevant literature in the field of swarm robotics. This includes a more comprehensive overview of current swarm research and the many approaches to the foraging problem, as well as how these approaches align with the stated goals of this project. The ideas presented in this chapter are intended to review what has already been accomplished by experts in the field, and how this project aims to contribute to these accomplishments.

The third chapter contains details about the overall design and approach taken in this project. This includes a discussion about both the higher level aspects of the library's architecture, as well as the particulars of each foraging algorithm. The implementation of these design decisions is detailed in the following chapter, which discusses how each of the modules used in this library function. The final chapters explain the experimental results of each foraging algorithm and evaluate their performance in the context of the broader goals of the project, as well as exploring how future endeavors may expand upon the framework provided in this library.

2. Literature Review

As a nascent, emerging field, the current body of research in swarm robotics is constantly evolving. There are a myriad of possible applications of swarm technology, and research has taken the subject in many different directions. In order to properly define the scope of this project, it is important to understand the current state-of-the-art in the field, how individuals are using what currently exists, and what direction these technologies might gravitate towards in the future. This information allows for an informed decision about what types of behaviors and algorithms would be useful to include in a possible software library.

2.1 Toolsets

2.1.1 The Buzz Programming Language

The Buzz language is a high-level programming language developed specifically for swarm applications[1]. In general, programming swarms takes place in either a top-down or bottom-up fashion, where a top-down approach involves designing and implementing the high-level behaviors of the swarm while a bottom-up approach focuses on designing the behaviors of individual robots within the swarm. Each approach has disadvantages; a top-down approach makes it difficult to tune and refine the behaviors of individual robots, while a bottom-up approach makes it exceedingly difficult to coordinate behaviors at the swarm level. Philosophically, the developers of Buzz acknowledge that both approaches are fundamentally important to swarm development, and offer developers the ability to program swarms from both top-down and bottom-up perspectives.

The Buzz language accomplishes this goal by providing many abstractions that allow swarm developers to simplify the design and implementation of robot swarms. For example, Buzz abstracts away specific robot hardware through the use of the Buzz Virtual Machine (BVM), which is installed on each robot in the swarm. The BVM then executes a Buzz script that is loaded onto every robot, making it possible to program heterogenous swarms of robots without the need to tailor the script to each individual robot according to their specific hardware.

Syntactically, Buzz resembles popular high-level scripting languages such as Python and JavaScript. This provides a highly readable format and a more manageable learning curve for developers who are already familiar with these more common languages. Beyond syntax, a fundamental feature of the language is the inclusion of swarms as a data structure, allowing groups of robots to be elegantly organized and controlled. Another basic advantage that Buzz provides is set of swarm-specific primitives such as communication with neighboring robots. This saves significant overhead on the end of the developer by preventing them from having to facilitate communication by creating their own communication architecture. A related concept in Buzz is the idea of a virtual stigmergy, which allows robots to propagate information across the swarm in the form of key-value pairs. By including these capabilities as a basic feature of the language, Buzz saves developers significant time and effort by providing all the vital infrastructure needed to program robot swarms.

2.2 Applications of Swarm Technology

As swarm technology matures, it has the possibility to impact many different domains of application. Many of the properties inherent in robot swarms make them useful in many situations that are currently very difficult or impossible to automate. One obvious application of such technology is any task where a region of space must be covered in some form. Swarm robotics provides a distributed sensing framework that allows for more efficient detection and response to stimuli compared to conventional approaches. For instance, consider the situation presented by Brambilla et. al.[2], where a group of robots must scour an area in search of dangerous chemical leaks. In this situation, robot swarms provide a distinct advantage because the robots can efficiently localize and determine the nature of any possible leak. Once a leak is detected, these robots could then self-assemble in such a way that blocks the leakage.

Like many other branches of robots, swarms may also see heavy use in situations that are simply too dangerous for human workers, such as clearing a minefield. What makes swarm robots especially suited for these applications is the redundancy that robot swarms provide over traditional robots. In hazardous areas such as a minefield or battlefield, any given robot has the possibility of being rendered unoperational (for instance, by an enemy projectile or a detonating mine). Given their decentralized nature, robot swarms are particularly equipped to handle the loss of some constituent robots, making them a better fit than the sort of centrally coordinated systems that can be realized currently.

The scalability property of robot swarms also makes them candidates for applications which involve scaling up or down. Consider a situation where robots are foraging for resources; as they discover more resources, it may become necessary to mobilize additional robots to cope with the increasing demand. While these fluctuations in demand might be minor, they could also be exponential and require an order of magnitude more (or less) robots. The ability of swarms to scale well without performance degradation makes them a better fit for these sorts of applications when compared to conventional approaches.

The concept of scalability in swarm robotics includes not only the number of robots involved with a given task, but the size of the robots themselves. Any of the above scenarios can be envisioned on a macro- or microscopic level. A swarm of robots could be scouring a large field, but they could also be coursing through the human body. While the degree of miniaturization required for the latter example is not yet possible, the technology may someday come to exist. The same concepts that apply to swarms on a macroscopic level are still relevant to these miniaturized applications as well.

2.3 Foraging

Just as swarm intelligence as a whole has a myriad of possible applications, foraging behavior is the particular focus of this project because of how widely applicable it is to real world situations. While the definition can be framed in a strict sense as a simple search-and-retrieval behavior for “food” scattered throughout an environment, a more comprehensive look shows that this simple definition can be reframed and generalized to a much wider range of “tasks such as search and rescue, mining, agriculture, or exploration of unknown or hostile environments.”[3]

Foraging behaviors are amongst the most common in swarm research, and as a result there are many previous experiments and projects on the subject that precede this one. This research spans a wide range of different approaches, each with different assumptions, environments, and other parameters. Many forays into foraging behaviors involve the development of a foraging algorithm, which prescribes a set course of action for each robot or group of robots that results in the desired collective behavior over the whole swarm.

2.3.1 Foraging Algorithm Parameters

One important way in which foraging behaviors may differ is the concept of a central place, or “nest”. In some scenarios, this central place simply doesn’t exist. This is analogous to a hypothetical real-world scenario where the robots in the swarm use the resource as soon as they acquire it. Others may assume the use of a single nest, while more still may consider the possibility of multiple nests.

Food and food distribution are critical parameters for foraging that can vary the performance of algorithms significantly[4]. Firstly, foraging depends on the number of food placed in the environment, or more specifically the **density of food**. The food can also be distributed in several ways. For example, the food could be **uniformly** distributed, where each small area of the environment has an equal probability of having food. Another possible distribution method would be **scale-free**. In scale-free distributions, there tend to be more small number of areas with large amounts of food, and many areas with small amounts of food. This creates an environment that is not only dense in some areas but also sparse in others, allowing to fully evaluate and test the performance of foraging algorithms. In addition, at a given location, the number of food can vary as well. These ‘piles’ are food stacked on top of each other and the sizes of the piles may vary up to a **maximum pile size**. Other parameters include changing the surroundings such as **field size** and even the **duration** of the simulation. Additionally, obstacles may be present in the environment. These obstacles may be static, such as walls, or they may be dynamic objects such as other robots.

2.3.2 Pathfinding in Foraging Swarms

A related concern is the way in which robots are able to actually locate the resources they are foraging for. One method employed by many researchers is the use of virtual pheromones, which are intended to act in a way that is analogous to the actual pheromones used by social organisms[4]. In the case of ant colonies, individual insects leave trails of pheromone chemicals so that they can easily rediscover a food cache once they initially discover it. More importantly, this pheromone trail also allows many other ants in the colony to find the food as well, and thus many ants can be engaged in pathfinding to carry this food back to the nest at once.

When it comes to simulating the function of this pheromone in the context of a swarm foraging scenario, there are many possible implementations. The work of some researchers makes use of actual physical markers which robots use to manipulate the environment. For example, robots might be able to leave trails of a chemical, drop a path of tokens, or create marks on the ground beneath them. While these approaches have the advantage of being tangible, simply ways of mimicking the behavior of nature, they also require the robots in a swarm to have some capabilities beyond simply moving and communicating with other robots nearby.

To assuage this problem, some foraging algorithms use the robots themselves as the pheromone. Since the robots are taking the place of a physical pheromone marker, no additional capabilities are assumed of individuals beyond basic locomotion and communication. The robots that are designated as pheromones simply use their communication capabilities to advertise the location of food sources to other robots in the swarm. This achieves the same end result, but has the drawback of requiring some proportion of the swarm to be allocated for creating pheromone trails. This in turn means that less robots are available for the collection of the food. Even within these two approaches, there is great diversity in pathfinding algorithms, each with different assumptions and tradeoffs.

2.3.3 Foraging with Virtual Pheromones

The paper by Hoff, Sagoff et. al.[5] provides an illustrative example of the different approaches to pathfinding that use the robots themselves as path markers. One of the algorithms they propose involves the use of virtual pheromone as a means of navigating between food and the nest. In this method, the pheromone is not a physical marker or substance, but rather a numerical value that is communicated between robots in the swarm. In contrast to actual pheromone-laying behavior, this approach actually involves the use of two separate pheromones: one to navigate towards food, and another to navigate towards the nest. Robots may randomly stop their searching and become “beacons,” which are stationary markers that can store these pheromones. The pheromone itself is simply represented as two floating point numbers-- one for each the nest and food-- and the value decays steadily over time. Other robots that are still searching for food can receive communications from these beacons to decide where they should go, where higher pheromones value represents increasing proximity to food or the nest, respectively. Another aspect of this algorithm is that it allows searching robots to transmit back to these beacons as they travel near them, which increases their pheromone value. This mimics the behavior of pheromone trails in real ants, where trails will decay unless ants continue to follow the same path to maintain pheromone levels along the path.

The other algorithm proposed in this paper is similar to the virtual pheromone algorithm described above, in that it allows each robot to take on the role of either “walker” or “beacon” at any given time. However, this algorithm differs in that beacons store the virtual pheromone as an integer value instead of a floating point number. In this method, which the authors term the cardinality algorithm, the pheromone instead represents the number of communication hops away from the nest or food. For example, beacons directly adjacent to the nest would have a nest cardinality of 1, robots adjacent to those robots would have a nest cardinality of 2, and so on. The “walker” robots can then follow the gradients formed by these cardinalities to their ultimate destinations. This discards the notion of pheromone decay in favor of a simpler method for constructing paths and evaluating their cost.

2.3.4 Foraging with Sweeper Algorithm

The sweeper algorithm as discussed in the paper by Alcherio et. al. [6] presents a strategy for collecting food in a systematic manner. The core methodology behind this algorithm is virtual forces. Essentially, each robot exerts a force on its neighboring robots inversely proportional to the distance between them. Initially, the robots are all exerting these forces onto each other which spreads them out throughout the environment forming a line. In addition, two robots are designated as pullers. These pullers experience an additional force. This force moves the robot around the environment in a similar manner to that of the sweeping motion on a clock. The pullers would exert a stronger force on other robots, making them follow the sweeping motion. When the line of robots approaches a food, the entire line stops moving. Gradients are calculated and the robot closest to the food picks up the food and gradient follows back to the nest. After the food has been returned, the line continues its motion. This is then run until all the food has been collected.

This method overall covers a large amount of area, but is slow due to the robots stopping each time they encounter the food. However, the algorithm is able to perform well in environments where the food is sparsely allocated due to the line of robots. By having the line of robots extend out further in the environment, the robots may be able to encounter food that other algorithms which prefer food closer to the nest would not be able to.

2.3.5 Foraging with Honeybee Algorithm

Brabazon et. al.[7] present a foraging optimization algorithm based on the natural foraging techniques of honeybee colonies. The algorithm, a variant of a standard genetic algorithm, disperses a set of agents (bees) and uses them to search in the space for an optimal solution[7] . Dispersed agents search a local area for a candidate solution, and during the evaluation phase agents with solutions found to have the highest fitness recruit neighboring foragers for a more intensive local search around the candidate solution. This process continues using the recruitment strategy until the best candidate solution is found.

2.4 Graphical Representation Language

For the purposes of explaining the often abstract concepts used in swarm system design, it is useful to have a means of expressing ideas in a graphical format. This visual form of expression can be codified into a more formal design language; Unified Modeling Language (UML) is a widely-used example of such languages, and is often employed in the context of explaining and developing software systems. However, developing swarm-based systems differs considerably from traditional robotics and software design, because it involves ideas and constructs that have no direct analogue in conventional approaches. As a result, the existing tools for representing software graphically are inadequate for swarm applications, and a new convention must be developed to suit the needs of this project.

It is possible, however, to borrow ideas from other graphical protocols in the development of a new protocol. The use of finite state machines (FSM) is a common, simple, way of expressing state changes; that is, how a system changes in response to a given input. In an FSM, the machine can be in only one of a given number of states at any point, and can transition to other states based off inputs to the system. In the context of swarm robotics, it might be useful to use an FSM to demonstrate how a given robot or group of robots change their behavior in response to interactions with each other and the environment. However, both FSM diagrams and UML diagrams fail to capture many essential elements of swarm systems, such as the concept of swarms themselves. Owing to their shortcomings, these conventions are unsuitable for fully representing multi-robot systems by themselves.

An attempt to remedy these drawbacks was made by Pitonakova, Crowder, et al.[9] in their paper *Behaviour-Data Relations Modelling Language For Multi-Robot Control Algorithms*, which details an attempt to create a modeling language specifically for swarm applications. The core properties of this language are that it creates graphical constructs-- which they term primitives-- that represent both data and robot behavior, and it can represent interactions between these constructs. For example, the language contains different primitives for internal data in the robots' memory and external data that exists in the environment. The interactions between these behaviors and data, which are referred to as relations in the paper, are likewise able to encapsulate a number of actions, including sending, receiving, transitioning, reading, and writing. While diagrams in this language are meant to represent the actions of a single robot, they also can be generalized and applied to swarms as a whole.

3. Design

3.1 Project Objectives

The stated goal of this project-- to present Buzz developers with a useful, foundational tool upon which their own projects could be more easily created-- is the essential principle that guided the design of this library. This motivation informed the design of this project at a macroscopic level, but also impacted the smaller, more atomic aspects of the library's architecture as well. This idea was combined with basic swarm design principles such as scalability and robustness to maximize the impact of this project on its targeted need. The sections that follow detail how these ideas were incorporated into the design of the final library.

3.1.1 Utility

A central focus of this project was to maximize the utility of the finished library to the average Buzz user. As was mentioned before, the foraging problem was chosen for this project specifically due to its wide range of applicability. Not only is foraging itself a common behavior for swarms, but many other scenarios can be reframed as a variation of the foraging problem. However, on a more granular level, we acknowledge that there is a great diversity of foraging algorithms that function optimally under widely different circumstances. For this project it was important to choose multiple algorithms that cover as many different scenarios as possible. For instance, it is useful to have one algorithm that performs well in food-dense environments and one that performs well in food-sparse environments.

For the purposes of this project, it was also important that our design made as few assumptions about the capabilities of robots within the swarm beyond basic communication and sensing abilities. In practice, this meant generally avoiding approaches that required specific hardware such as precise odometry and physical pheromone markers. Instead, these capabilities were recreated by propagating information throughout the swarm and using this data to make collective decisions. This approach benefits the end user by not requiring any specific hardware to use the behaviors within the library.

3.1.2 Modularity

The other important guiding principle in the design of this project was the idea of modularity. In the context of the behavior library, having a modular design meant building a hierarchical system of components that were interchangeable, replaceable, and able to be used by themselves. In a more concrete sense, this meant decomposing the larger foraging algorithms into submodules that handled specific aspects of the overall behavior. These sub-modules could then be further divided into simpler, more granular behaviors.

To the extent possible, these sub-modules were reused and shared between algorithms. For example, several different foraging algorithms within the library might use the same path-finding behavior. This path-finding behavior would be its own module that is re-used by both algorithms, but could also be used by itself in a context other than foraging. Furthermore, a truly modular behavior would allow developers to come up with a new implementation of a path-finding behavior and use it interchangeably with the older implementation. This allows developers to use only select parts of the library according to their needs, and also allows them to easily extend or alter the basic behaviors with their own modules.

3.2 Foraging Algorithm Design

The sections that follow explain the theoretical basis for the several different foraging behaviors included in the library. The three algorithms described below all borrow from existing ideas in the literature to some extent, and all have specific advantages and disadvantages. Each algorithm is described in terms of how it functions at a high level; the exact structure of each sub-module is left for later sections.

3.2.1 Assumptions

There are many potential variables in the setup of these algorithms. However, evaluating every possible parameter would require complex, multi-dimensional analysis beyond the intended scope of this project. For this reason, each of the algorithms implemented share a core set of assumptions. These assumptions were based on the literature review as well as practical considerations.

For the purposes of experimentation, all robots are assumed to start near the nest, which is located towards one end of the environment rather than in the center. The size and shape of the experiment field remains constant across all tests, allowing for simpler comparisons between algorithms. However, exact distribution and topology of resources is randomized between trials. Furthermore, robots are equipped with sensors to detect food and nest, as well as being able to pick up and drop off the food by themselves. Other sensors include proximity, positioning, and range and bearing sensors. The Khepera IV was used in all of the experiments, however any robot with the above mentioned sensors will be able to run the algorithms. For the purposes of simplicity, all of the foraging algorithms discussed in this paper were run using the ARGoS multi-robot simulator[8], which enabled Buzz scripts to be run in a way that closely simulates the performance of actual robots.

3.2.2 Modeling Language

The algorithms used in this project are described in terms of a created modeling language to visually represent their function. This language is inspired by similar design languages such as the one described by Pitonakova et. al[9], as well as independent research and thought. The essential property of each diagram is that any robot can be thought of as belonging to only one state at any given time step. In this language, states are represented by ovals. These states determine the behavior that the robot performs, which are represented by rectangles in the diagrams. A robot can reach other states and behaviors through state transitions, which are represented by arrows. Sometimes these state transitions are dependent on a logical condition, which decides between one of many possible outcomes. These decision points are represented by diamonds in the modeling language. Dotted arrows are distinct from solid ones; they represent that a given state “uses” the behavior pointed to by the arrow, rather than indicating a state transition.

3.2.3 Puller Algorithm

The Puller algorithm takes inspiration from the sweeping algorithm discussed in the literature review. The sweeping algorithm proposed forming a line of robots and having that line perform a sweeping motion to explore the environment. However in practice, the pullers were not able to exert a force strong enough to move the rest of the robots. This made forming a line challenging, and thus we pivoted to an alternative strategy that still shared common methods such as forces. Like all the other foraging algorithms discussed in this paper, the Puller algorithm can also be broken down into three main tasks: finding food, retrieving food, and task allocation. An overview of the algorithm is shown in Figure 1 below.

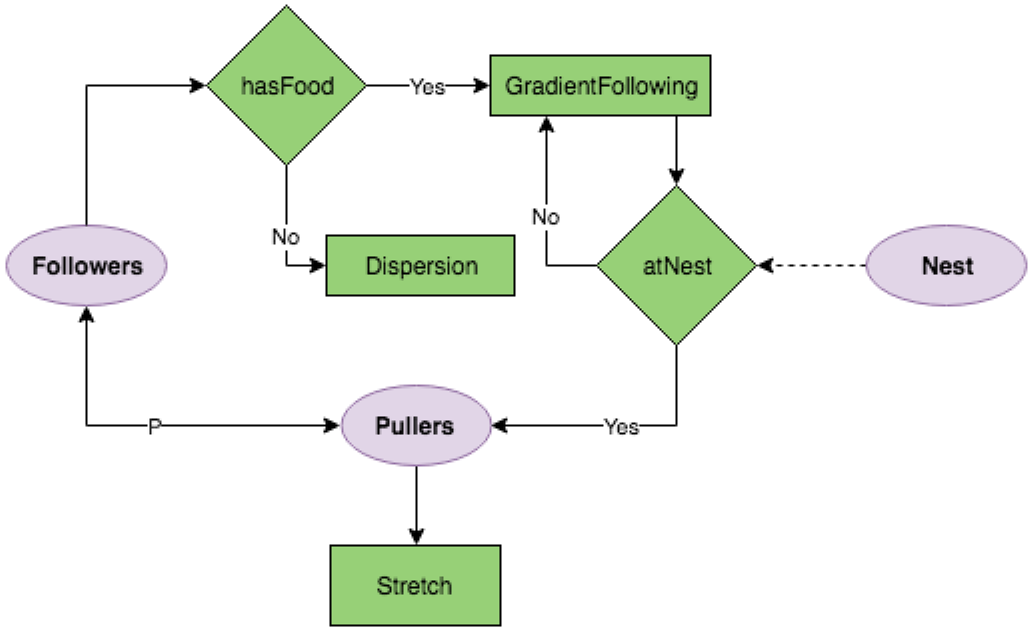


Figure 1: Swarm Modeling Language Representation of Puller Algorithm

For finding food under the Puller algorithm, robots predominantly use forces, specifically dispersion. Dispersion allows the robots to spread far from each other to cover the maximum amount of area while still being in communication range of other robots. This optimization however does result in the robots eventually reaching a 'stable' state where the net force they experience is zero. In this case, robots would be still and thus not find any food. In order to break this stable condition, robots could also be 'stretching', an entirely different algorithm for finding food. The stretching behavior is similar to that of the pullers described by Alcherio et. al[6]. The difference is that the vector is not time dependent. Instead the robots will travel in the direction opposite to the nest with some variance. The variance introduces randomness and allows robots to explore areas that otherwise they would not, which is the entire reasoning behind the stretching.

Retrieval of food is simple. If a robot is carrying food, then it will gradient follow back to the source of the gradient, which is the nest. The nest in this case is a collection of robots which is calculated at the beginning of the algorithm. The center module (as described in Section 4.1.7) is used to determine the robots representing the nests. The robots with the highest resulting count after some time are determined to be the nest. An example of the nest is shown in Figure 2. With the nest constantly broadcasting the gradient source, other robots maintain their gradients and are then able to gradient follow back to the nest.

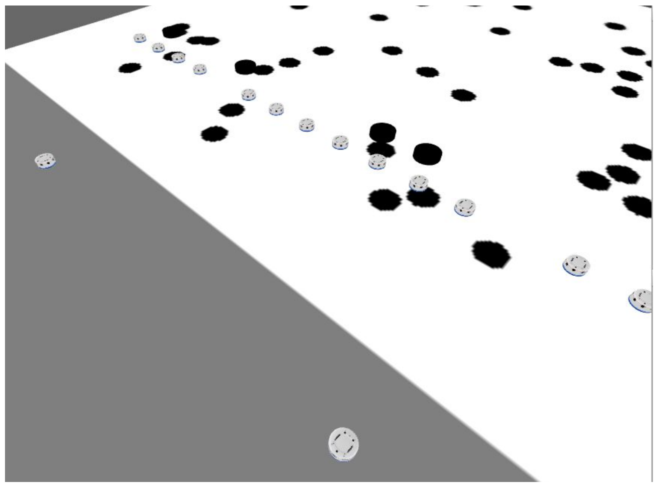


Figure 2: Example of Nests in ARGoS

Lastly, in order for the Puller algorithm to work, task allocation is needed in order to assign tasks to robots. The module used for task allocation is the stimulus model. With the stimulus model, robots have a certain probability of performing different algorithms for finding food. Since the stretching behavior is only needed when robots are close to the stability point when performing dispersion, the stimulus response for the stretching behavior is smaller than the stimulus response for the dispersion. The stimulus for retrieving food depends on whether the robots are carrying food. Robots will retrieve food if and only if they are carrying food.

The drop probability is currently a constant based on the priority of a task. Ideally, this probability would be a function of the priority and the time. As time goes on, the probability of a robot dropping the task should increase as well. This increases the randomness in the environment and helps break the stability conditions of the algorithms which are more likely to be met later during the experiment.

3.2.4 Forager-Beacon Algorithm

This foraging algorithm borrows the idea described in the previously-referenced paper by Hoff, Sagoff et al titled *Two Foraging Algorithms for Robot Swarms Using Only Local Communication***.** The central concept of this algorithm is to divide the entire swarm of robots into two main groups: “foragers” who search for and retrieve food, and “beacons” that act as pheromones to guide other robots to food and the nest. Robots are assigned into one of these states at random, and can switch between states.

The makes use of virtual “pheromones” to mimic the way a swarm of ants or other insects might find and retrieve food. The idea of these pheromones is used in many different foraging methods, but how the pheromones are simulated can vary. In this algorithm, we borrow an idea from others to use the robots themselves as pheromones. This has the advantage of not requiring any additional capabilities from individual robots, such as being able to place physical markers (pheromones) within the environment.

The beacons maintain two pieces of information that allow them to guide other robots-- we'll borrow a term from the previously reference paper and refer to these as the nest cardinality and the food cardinality. The food cardinality is a heuristic that incorporates how near a beacon is to a food source and how much food exists at that location. The closer a beacon is to food and the more food that exists, the higher this number will be. The nest cardinality simply represents the estimated distance from a given beacon to the nest.

Without guidance from beacons, forager robots simply wander about randomly. If a forager encounters food during this random walk, it instantly changes state to become a beacon. It then initializes its food cardinality as a function of the amount of food present. It then broadcasts this information to neighboring beacons. The neighboring beacons will then see this food cost, and generate a food cost of their own. This newly-generated food cost is a function of both the distance of that beacon from the food source, and the amount of food present at that source. This process then continues to the neighbors of that robot, and so on. In doing so, a gradient is constructed that approximates the relative amount of food near to any point in the environment.

The other half of this process is to construct a second gradient that represents the relative nest cost of each beacon. This process starts with beacons that are neighboring the nest, which initialize their nest cost to be the estimated distance between the robot and the nest. Neighbors of these beacons then set their nest cost to be the nest cost of the first beacon plus the estimated distance between the two neighbors. In this way, paths are constructed outwards from the nest to all reachable beacons. Foragers can then use this gradient to find their way back to the nest once they've retrieved food.

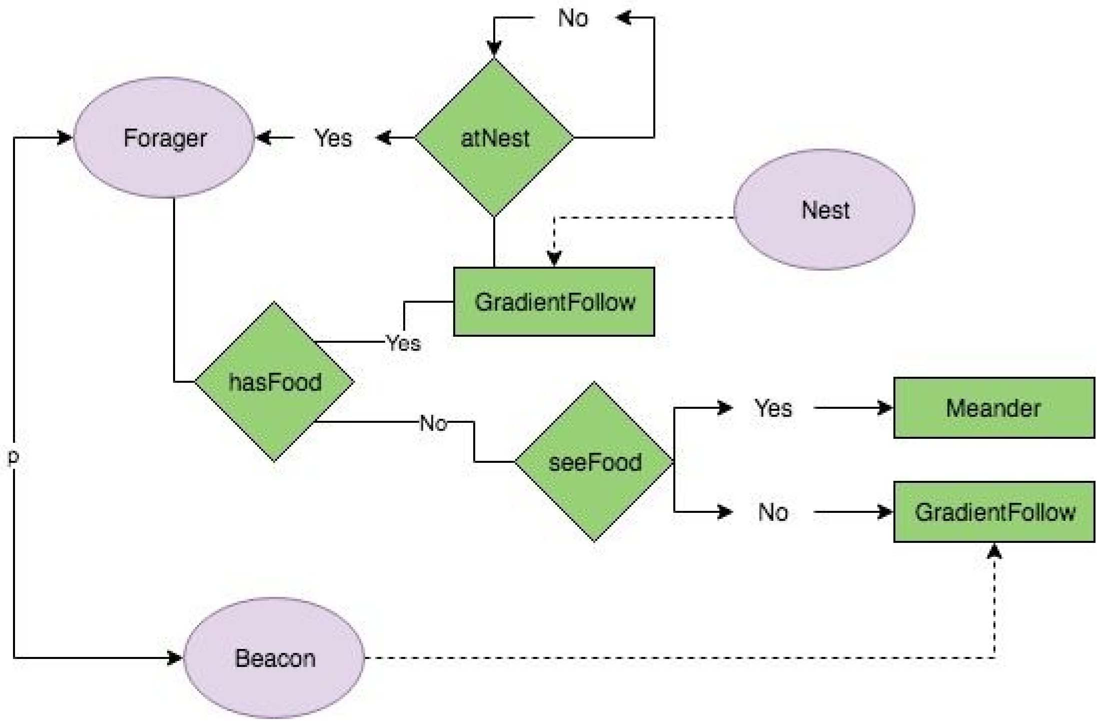


Figure 3: Swarm Modeling Language Representation of Forager Beacon Algorithm

The entire high-level operation of the Forager Beacon algorithm is described in the diagram above. The diagram shows that any point, robots may fall into one of three roles: foragers, beacons, or the nest. At least one robot must be designated as the nest so that returning foragers have a means to find the food drop-off area. This robot has no other purpose other than to remain in place and broadcast its position to the rest of the swarm. This is a simplification to make experimentation easier, and more complex setups could replace this nest robot with an some other construct that fills the same role. The declaration of the nest robot happens at the beginning of the simulation and is static, as that robot remains the nest for the rest of the simulation. In the diagram, this is evidenced by the fact that there are no state transition arrows leading to or away from the nest.

Foragers and beacons may change roles however. This switching happens probabilistically, as represented by the double-ended state transition marked *p* in the diagram. The actual implementation of foragers and beacons is slightly more complex than indicated in this diagram. The function of these roles is actually encapsulated in four separate tasks that are handled by the task allocation module. The entire foraging behavior, for example, is comprised of three tasks: finding food, returning food, and random walking. The random walking behavior can be thought of as a default task, which foraging robots will perform if they do not have food to bring to the nest and they cannot see a path to any food piles. If they can see a path to a food pile, they switch to the “find food” task and navigate to the food by gradient following. Once they’ve picked up a food item, they switch to the “return food” task and bring that food back to the nest, at which point the cycle repeats. However, at any time step there is a chance that a forager will be switched to being a beacon. Foragers will never be interrupted while they are returning food, but they can be forced to switch at any point while meandering or finding food.

Once a robot is a beacon, its behavior is much simpler. There is only one task for the robot to execute, which is to advertise its location relative to food to all other robots within range. Though they all perform the same basic behavior, it is important to note that there are fundamentally two different types of beacons. When a randomly walking robot happens upon a pile of food, it is instantly forced to become a beacon so it can guide neighboring robots to the location of that pile. Other robots become beacons randomly at positions where there is no food. Their job is to simply relay the source of the food and nest gradients to their neighbors. This is necessary because it maintains connectivity throughout the swarm. If stationary beacons were not present, groups of foragers would become cut off from the nest gradient as they flocked towards piles of food. The finding food task functions similar to a flocking behavior, where large groups of robots converge on a single location. This leads to “islands” of many robots, but these groups are often too far away from other robots to propagate gradient information and thus have no path back to the nest. Having some robots remain beacons even when they are not on a pile of food maintains this connectivity.

3.2.5 Honeybee Algorithm

The honeybee algorithm was developed with roots in the real-world foraging behavior of honeybee colonies, with slight modifications inspired by Brabazon et. al.[7]. At its core, the algorithm defines two behaviors - the “scout” behavior and the “worker” behavior. Scouts are tasked with dispersing into the arena to find food locations, while workers know of a food location and will travel back and forth between the nest and that location ferrying food back to the nest. Robots will switch between these behaviors as conditions dictate, or at random with some predefined probabilities. An overview of the algorithm in the Swarm Modeling Language is shown in Figure 4.

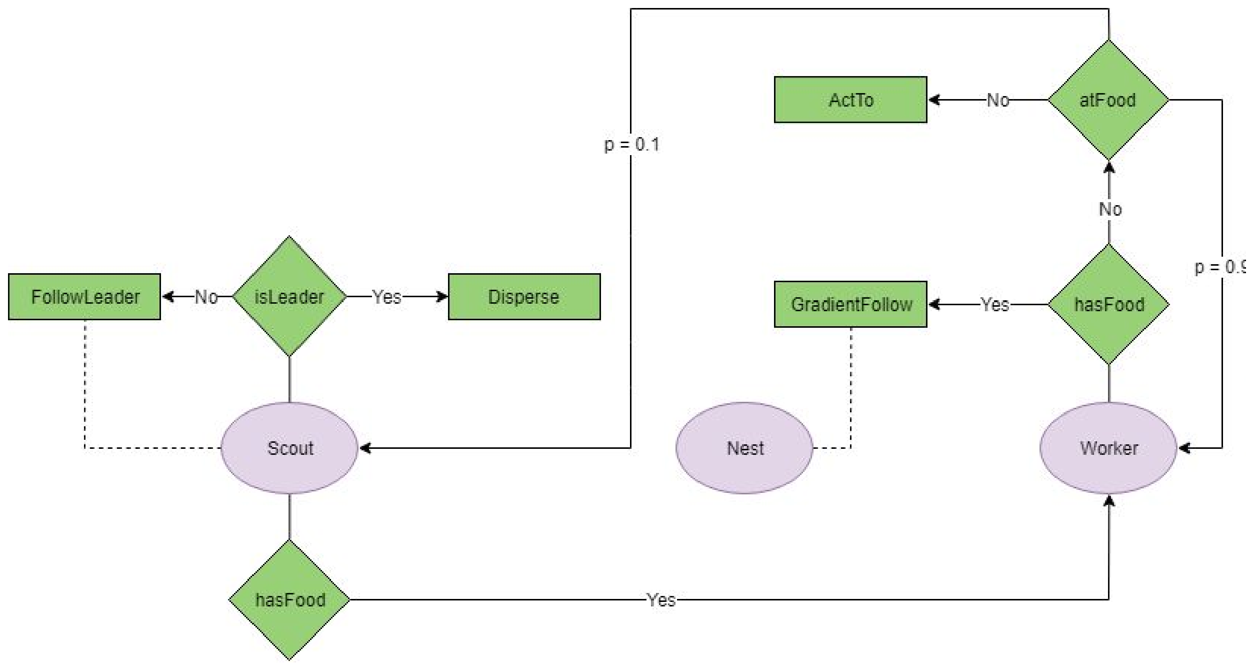


Figure 4*:* Swarm Modeling Language representation of the Honeybee Foraging Algorithm

The heart of the honeybee algorithm is the idea of recruitment. In nature, honeybee foraging relies on recruitment of teams by the scouts to fly out to the discovered location for retrieval of nectar and pollen. Scouts will memorize locations during the scouting phase and upon return to the hive will participate in a recruitment method known as the waggle dance, which encodes the location as a series of motions. Given the various dances, idle worker bees will decide to join a particular scout and travel out to that food location to retrieve the resources found there. The honeybee algorithm implements a similar mechanism for recruitment, allowing robots performing any given action to actively recruit direct neighbors to join them. This mechanism is what drives the switching between the scout and worker behaviors.

The scout behavior attempts to disperse into the arena in a way that allows for maximum coverage of the unexplored locations. For this reason, it makes uses of the recruitment functionality to form a scouting team that travels together out into the arena. One robot is considered the leader of a scouting team and heads away from the nest, and recruits other robots to follow. At the beginning of a given experiment, this allows the swarm to disperse fairly evenly across the environment and divides the search space. Scouts continue this behavior until recruited away.

The key to the worker behavior is encoding the location of a discovered food pile and broadcasting this location to potential recruits. Once robots have allocated to a worker task, they travel between the food location and the nest ferrying food from the location back to the nest. This behavior continues until a robot arrives at the location and sees no more food there; at this point it deallocates and is available to respond to recruitment messages from other active tasks.

4. Methodology and Implementation

The foraging algorithms contained in this library are composed of granular sub-behaviors, or modules, that are then combined to create more complex behaviors. This section details the implementation of the overarching algorithms in terms of individual modules. Understanding how these modular foraging behaviors work requires understanding how the individual sub-behaviors function and how they help to accomplish the larger goals of the algorithm. In addition to these components, this section also explains the experimental setup used to test and evaluate the foraging behaviors as a whole.

4.1 Modules

4.1.1 Gradient Module

Gradients are one of the most common tools in swarm programming and can be used in variety of ways. A gradient is a collection of robots that share a specific message type with each other. Each robot in a gradient contains a hop count. Each gradient has a source which starts with a hop count of 0. The hops of a robot is one more than the minimum hops of the neighboring robots. In order to implement this in Buzz, we created a table for gradients:

Each robot holds a table for each gradient. The table contains whether the robot is the source of the gradient, the id of the robot, the id of the parent robot, and the vector to the source of the robot. These fields allow the robots to perform gradient following where any robot can go to the source of the gradient. At each time step, the gradient for the robots need to be recalculated. In order to do so, each robot sends its own gradient message to neighboring robots. After comparing the current gradient table with the incoming one, the robots choose the table with the smaller hop count. If the hop count is exactly the same, then the robot which is closer to the source of the gradient is chosen. There were other more involved factors concerning updating the gradient as well. For example, if a robot receives a gradient status from its parent robot, then it will always update it's own gradient. These edge cases were essential to properly implement gradients and gradient following in our algorithms. Once every robot has done this, the new gradient has been established. For gradient following, the goal of a robot is to drive to another robot's location, usually the source. This is done by utilizing the parent id stored in the gradient message. The robots that are performing gradient following always go towards their parent using the forces module (as described in the next section) until they have reached the target.

Gradients can be used to determine the relative size of the area that has been covered by the robots. Gradients can also be used to allow robots to move from one location to another. Specifically in our applications, this involves robots going towards the nest and the food locations. There is a seperate gradient table needed for the nest and for the food, and robots are able to follow the gradients back to their source.

4.1.2 Forces Module

One of the key motion modules we implemented involved using forces. With this module, robots can exert virtual forces onto other robots based on the distance between them. This force can then either push robots away from each other or towards them.

Dispersion was a key implementation in the forces module. The dispersion functions maintains the robot's distance based on the given input. This is done by using the equation described below.

$$F = \frac{a \cdot d}{R} - b, \tag{1}$$

where a and b are constants, R is the desired radius of separation, and d is the distance vector between the robots.

With dispersion, developers can easily maintain distance between a swarm of robots to avoid congestion. Eventually the robots performing dispersion will reach a stabilization point where the net force they experience is zero. In this case, robots don't move and simply wait until the environment changes. In addition to dispersion, the forces module also offers other functions such as moving towards a particular force vector, obtaining the net force vector on a particular robot, and running away from and towards one robot. All of the functions in the force module are easily extensible to any application requiring motion.

4.1.3 Stimulus-Response Task Allocation Module

Task allocation is an essential functionality for a robot swarm in order to determine the tasks for each robot[10]. Rather than having individual robots decide by themselves on their tasks, the task allocation module assigns tasks to swarms of robots. The task allocation modules provided in this library define the notion of a task as a collection of functions and parameters encapsulating the selection and execution of the task. A detailed representation of the task structure is shown in Figure 5.

Parameter	**Description**
stimulus()	The function used to calculate the dynamic stimulus
execute()	The function that actually performs the task
priority	A value between 1 and 5 indicating the priority of the task

Figure 5: A Representation of the Internal Task Structure used in the Task Allocation Modules

Task creation involves defining the behavior functions and registering the task with the allocation module. Once registered, a task is available for activation by individual robots. A robot that activates a task will start performing the task (as defined by the behavior) and make the task available to the other robots based on the allocation strategy.

The stimulus-response allocation strategy operates on the assumption that every active task and its state is available to each robot at decision time[11]. From the state of a task, each robot calculates a dynamic stimulus value, which represents the “eagerness” of the robot to join that task. A higher values of the stimulus thus results in a higher likelihood of the robot allocating to the task providing that value.

The process for selection with the stimulus-response model consists of a calculation phase and a weighted random selection phase. In calculation, the robot calculates a stimulus value for every active task and generates the stimulus list. In the selection phase, the robot employs a weighted random selection strategy, using the stimulus values as weights. Once the robot has selected a stimulus, it joins the task that produced the maximized stimulus value. The calculation follows Equation 2, where s_p is the stimulus of task T_p for all n active tasks.

$$selection = uniform([s_1 T_1, s_2 T_2, ..., s_n T_n]) \qquad (2)$$

4.1.4 Recruitment Task Allocation Module

The recruitment allocation strategy is an extension of the stimulus-response model. Rather than having all active tasks available to all robots, tasks are instead broadcast by robots currently performing them to direct neighbors.

The module uses the same task structure as the stimulus-model based module and the selection process is similar. One important difference is that the selection process only occurs at a specific frequency (once every ten timesteps) due to the volume of messages being broadcast at any given timestep. However, the selection process still involves calculation of the dynamic stimulus and the same weighted random selection process as before.

4.1.5 Obstacle Avoidance Module

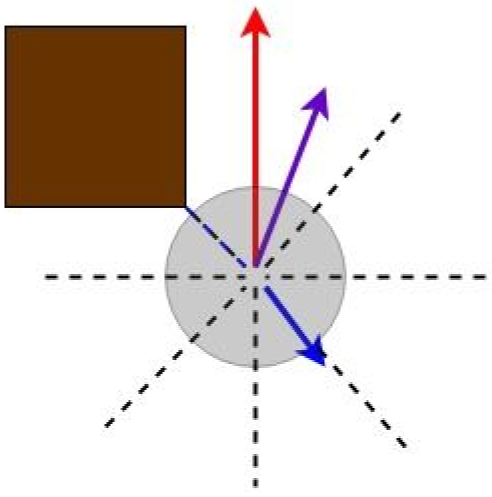


Figure 6: Illustration of obstacle avoidance module

Collisions between robots can disrupt the performance of an entire swarm, so it is important to have a method to avoid them efficiently. The obstacle avoidance borrows ideas from McLurkin[12] to create a module that provides this small but necessary functionality to all algorithms in the library. The figure above illustrates how the avoidance module functions when an obstacle-- such as a robot or an immobile structure in the environment-- is encountered.

The transparent circle represents the robot, while the red vector represents the current direction the robot is headed towards. In this scenario, the robot will collide with the brown object to its left if no corrective action is taken. At each time step, the obstacle avoidance module reads each of the robot's proximity sensors, which are distributed radially around the robot. In this diagram, these sensors are represented by the dotted black lines.

If an obstacle is detected, the robot then calculates the distance vector to that object, which is represented in the diagram as the dotted blue line from the robot to the obstacle. The opposite of that vector is then calculated, as shown by the blue arrow vector. By adding the original direction vector to the blue obstacle vector, a new vector can be calculated which represents the direction the robot should travel to avoid the object, which is represented by the purple arrow vector.

This approach works even if there is more than one obstacle, as the resultant heading vector will still typically point away from any obstacles within range. The robot receives the strongest pull away from an obstacle when the object is farther away, since the avoidance vector is largest at this point. This is desirable because it gives the obstacle avoidance behavior a "proactive" tendency, where robots move away from obstacles before they get too close.

Though this module is relatively small, its purpose is important and it is widely applicable across many different circumstances. In addition to being used in each foraging algorithm, obstacle avoidance is useful in many other swarm behaviors as well. This property makes it a valuable addition to this library, as future users can easily incorporate this module into their own projects even if the larger foraging behavior is ignored.

4.1.6 Random Walking Module

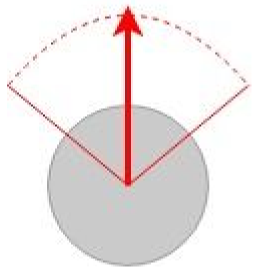


Figure 7: Illustration of meander behavior

This module contains a variety of movement behaviors that are useful for foraging, exploration, and other swarm behaviors. The most fundamental behavior in this module is the meander function, which provides a means for robots to wander aimlessly through the environment. The meander behavior works by randomly changing the direction that the robot moves in at each time step. However, randomly selecting from all possible directions on each timestep causes the robot to barely move at all, as it does not have time to make progress in any one direction. Instead, the random direction vector is constrained to be within some range of the previous direction vector, as shown in the diagram above. The red arrow represents the previous direction, and the new direction vector must be within about 45˚ of this direction to either side. The possible range of this new vector can be adjusted, where narrower ranges lead to straighter, smoother paths than wider ones. This behavior is useful for exploring the environment in the absence of other information, in the context of foraging or otherwise. In this project, it is used by the Forager-Beacon algorithm to forage for food when there is none currently visible.

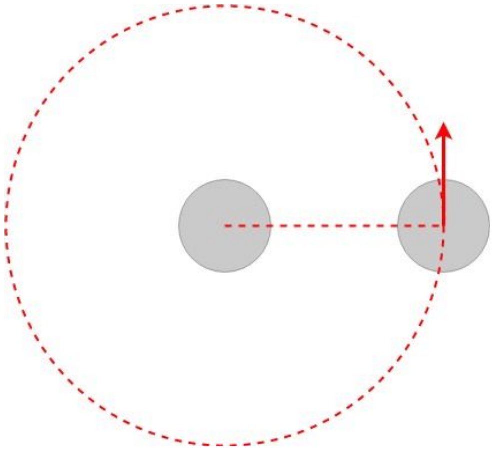


Figure 8: Illustration of circling behavior between two robots

A related behavior in this module is the circle function. Circling is a straightforward behavior that allows for one robot to travel in a fixed radius around another, as represented in the diagram above. This behavior is achieved by finding the vector between the two robots, and then adding 90 degrees to the angle of that vector. This creates a new vector that is perpendicular to the vector between the robots. The circling robot then travels along this vector with a magnitude value, which can be adjusted to control the speed of the robot along its path. Because this calculation is performed at each time step, the direction of the perpendicular vector shifts slightly on each iteration, resulting in a circular path around the other robot.

This function is used in the Forager-Beacon algorithm to find food. When a forager arrives at a beacon, it may not instantly detect food due to idiosyncrasies in the location of the beacon relative to the food pile. Having the forager circle the beacon makes the algorithm more reliable because it effectively forces the forager to search around the beacon if it does not pick up food immediately.

A simple modification to the circle behavior results in two additional functions that allow robots to travel in spiral patterns relative to each other. This is accomplished by adding or subtracting a small angle from the perpendicular vector calculated in the circle function. Adding a small angle to this vector will cause the robot to travel slightly outwards relative to the other (spiralling outwards), while subtracting a small quantity will cause the robot to travel slightly inwards (spiralling inwards). Spiralling inwards is useful for much the same reason as circling, as it is simply a different way to search the area around a robot. It is in some respects superior to the basic circling behavior, as it covers more area. Spiralling outwards is an effective way to explore the environment. It is more methodical and consistent than meandering, and allows for guaranteed exploration of a certain area. However, meandering has advantages where guaranteed exploration is not a concern, as it can typically cover a wider total area in a shorter amount of time.

4.1.7 Center Module

One of the most important aspects of foraging is for robots to always be able to travel back to the nest. Often in swarm programming, robots aren't assumed to have specific sensors like smell that allow them to locate the nest. In order to solve this problem, a group of robots remain at the nest and broadcast to all other robots their location. This way every robot is always aware of the nest's location. Choosing such robots can be a difficult problem however. If there are too few robots representing the nest, there may be traffic jams and congestion when other robots travel back to the nest. This can hinder performance significantly. On the other hand, if there are too many robots representing the nest, then there may not be enough robots out on the environment, and the algorithm would still perform worse. The center module chooses such a group of robots in an efficient manner that remain at the nest. To calculate the nest, robots initially send each other messages representing a count. Robots then accumulate the count received from their neighbors and then send the resulting count with some decay factor to their neighbors. After waiting some time steps, the robots with the highest accumulated count are chosen as the nest. The count for each robot is also stored in a global stigmergy, which then allows all the other robots to calculate the closest robot representing the nest. The center module is utilized in all of our algorithms as a tool to efficiently choose robots to be the nest so that the other robots can cover as much of the environment as possible.

4.1.8 Miscellaneous Modules

This library also contains a variety of functions that did not fit neatly into any other module, but were nonetheless useful. Many of these functions were small, simple operations that satisfy a use case that is not currently covered by any other standard function in Buzz. These functions were included as part of the library because they are generally useful, even apart from their applications in foraging. Their inclusion prevents future developers from having to create their own functions that effectively perform the same task.

Many of these miscellaneous functions perform actions related to neighbor interactions between robots. For instance, the there is a function that can be called on any robot with a specific robot id as a parameter, and returns a value indicating whether the two robots are neighbors. A similar function in this module returns a value representing the distance between the two robots. Other included functions allow users to find the number of neighbors that a given robot has, or retrieve the ID number of a random robot within neighbor range.

4.2 Performance Testing Framework

4.2.1 ARGoS Environment Setup

Our project creates the environment based on the following given information: density of food, x and y length for the range of food displacement, the food distribution method, density of robots, and two output files for data collection.

Two environment variables are used in ARGoS that are required to be declared in the accompanying Buzz script: *hasFood* and *foodCount*. These two variables are held by each robot and represent whether the robot is carrying food, and if the robot is on a pile, how many food are located on the pile. These are necessary in order for the simulation to notify the robots they have reached a food or nest location.

4.2.2 Cluster

As one of the main goals of our project, we tested each of the algorithms performance by performing thousands of simulations. To do so, we utilized the Turing Cluster at WPI. We created two bash scripts for setting up and running the experiments. The purpose of the main job was to iterate through the parameters of the simulation, and then launch the simulation job with the chosen set of parameters. The simulation job then took these parameters and filled them into a template argos file which was then run. One important consideration while we were creating our framework was the maximum number of jobs that could be queued onto the cluster. In order to avoid spamming the cluster with thousands of jobs each representing one simulation, we grouped simulations with the same set of parameters excluding the random seed into one job. This reduced the number of jobs, while also still allowed for parallelization to run multiple simulations at the same time. The time taken for each simulation heavily depended on the number of robots. The typical time ranged from 15 minutes to 25 minutes. We were able to complete and collect data from over 2500 simulations in total.

In addition to the job scripts, we also developed programs for visualizing the data retrieved from the simulations. We utilized Jupyter and python for reading and creating the graphs from the data using libraries such as numpy, pandas, matplotlib, and scipy.

5. Results

5.1 Reusability and Modularity

Modules	Times Used
Gradient Following	3
Obstacle Avoidance	3
Find Center	2
Stimulus Model	2
Forces	2
Random Walk	2
Transforms	2
Neighbor Based Recruiting	1

Figure 9: Breakdown of Modules Reused between Algorithms

A major goal of this project is to demonstrate a high degree of reusability across all the software modules contained in the library. The degree to which this goal is met can be measured in many different ways, such as by quantifying the amount of shared code between algorithms. The diagram above shows how modules are shared and reused between the three algorithms included in this library. There is a considerable degree of overlap, with some modules being used by all three algorithms and several others by at least two.

Another major design objective of this project is to ensure a high degree of modularity among the software components included in the library. For instance, the Forager-Beacon algorithm uses the meander behavior to explore the environment, whereas the Puller algorithm makes use of the pulling behavior. However, these components are modular in the sense that the meander behavior could be swapped for the pulling behavior, yet the overall performance of the algorithm would remain the same. Many other components of the library share this interoperability, and future developers could easily extend the current options with their own, improved behaviors. This is useful because it allows users to potentially tailor the sub-behaviors of the larger foraging algorithm to a specific environment.

Creating self-contained software modules also simplifies and shortens the necessary code for foraging algorithms. During the development process, both the pulling and forager-beacon algorithms were partially created before they were re-made using the task allocation module. The process of porting to a task allocation-based module shortened the overall code body of each algorithm by several hundred lines, and also improved subjective characteristics such as readability. This example is illustrative of how this library might benefit future users by providing ready made swarm development tools. By providing much of the overhead, this library can prevent developers from having to do unnecessary base work to create their own projects.

5.2 Foraging Parameter Values

Table 1: Breakdown of Foraging Parameter Values

Parameters	Values
Topology	Uniform, Scale-Free
Robot Density	0.3, 0.5
Food Density	0.2, 0.5, 0.8
Maximum Pile Size	1, 5, 10

Table 1 shows the different parameters that were considered during the experiments, and the values associated with the parameters. These values were chosen so that there was a wide range of different arena setups so that we could properly measure the performance of the algorithms.

5.3 Puller Algorithm Performance

The performance of the Puller algorithm was evaluated based on over 500 simulations with varying parameters as shown in Table 1. In order to effectively evaluate the performance, we considered the effects of each of the parameters on the algorithm, including the maximum pile size, and the topology. Furthermore, a more in-depth analysis was performed for locating areas of the environment the algorithm performed better in. Overall, these measures combined yielded in the strengths and weaknesses of the Puller algorithm.

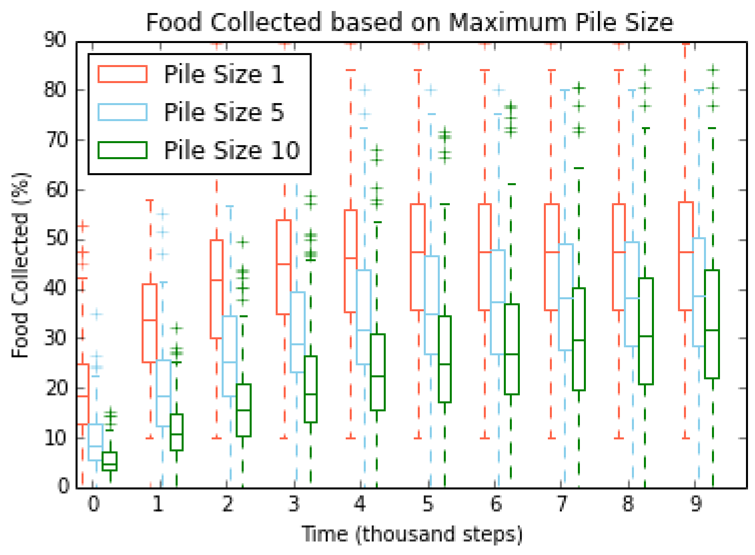


Figure 10: Food Collected by the Puller Algorithm based on Maximum Pile Size

Figure 10 above shows the performance of the Puller algorithm based on the maximum pile size. The huge variance in the performance is likely due to the other parameters affecting the results of the algorithm. Therefore, we looked once again at the performance in most occupied environment.

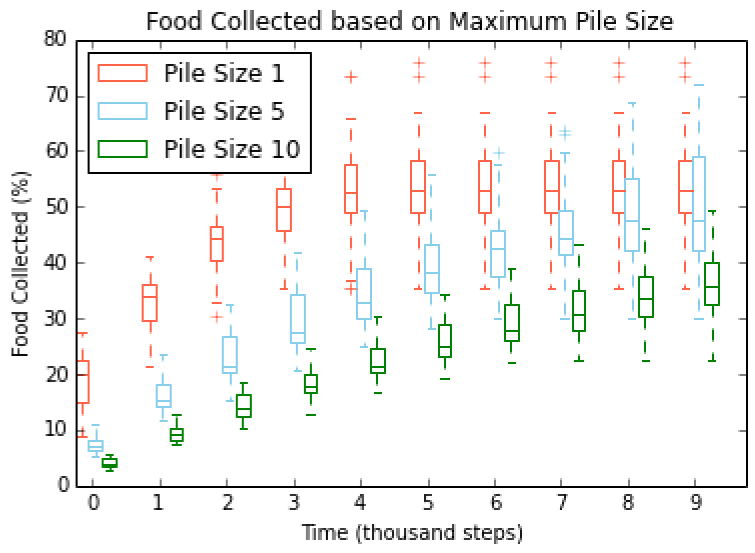


Figure 11: Food Collected by the Puller Algorithm in Most Occupied Environment based on Maximum Pile Size

After narrowing the food density and robot densities to their highest tested values, the result shown in Figure 11 is much clearer than in Figure 10. The performance with the maximum pile size at one clearly outperformed the performance with maximum pile size ten, and outperformed maximum pile size five during the start and then reached roughly the same towards the end. It is also important to note that both performances for max pile sizes five and ten were still steadily increasing at the end of the experiment, while the performance at pile size one remained stagnant. This is likely due to the robots not having enough time in collecting all the food. In order to prove that the performance is correlated to the maximum pile size, we also conducted the Wilcoxon Rank Sum test to measure the likelihood of the distributions being the same.

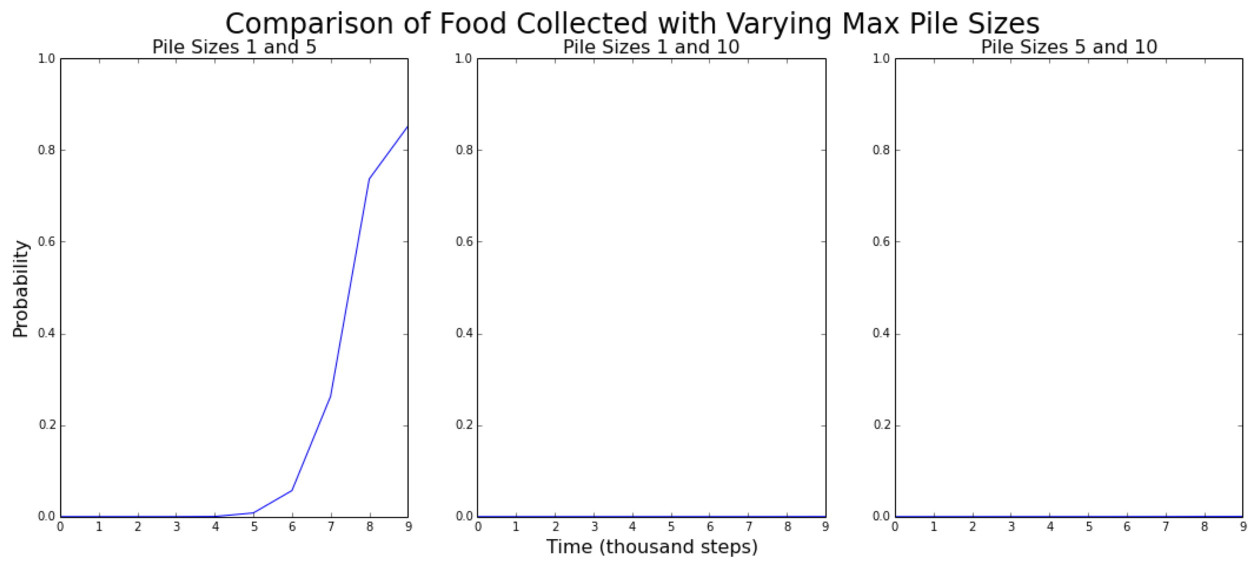


Figure 12: Wilcoxon Rank Sum Test on Food Collected with Varying Max Pile Sizes

Table 2: Average Wilcoxon Rank Sum Test on Food Collected with Varying Max Pile Size

	1 and 5	**1 and 10**	**5 and 10**
Probability	0.192	0.0001	0.0001

Figure 12 compares the distributions in Figure 11 using the Wilcoxon Rank Sum Test. This test allow us to check the probability that two distributions are equal, and from the results it is clear to see that the performance on max pile size one differed significantly from max pile size ten and on max pile size five and max pile size ten. Overall for both of these comparisons, the probability of the two distributions being equal was nearly 0.01% as displayed in Table 2. For the comparison between max pile size one and five, the average probability was roughly 19.2%. However, as shown in the figure, the probability during the start of the experiment was low, and increased up to over 80% during the final step. Therefore, the performance on max pile sizes one and five could be similar especially towards the end. Overall, the results show that the Puller algorithm performs better when the maximum pile size is lower.

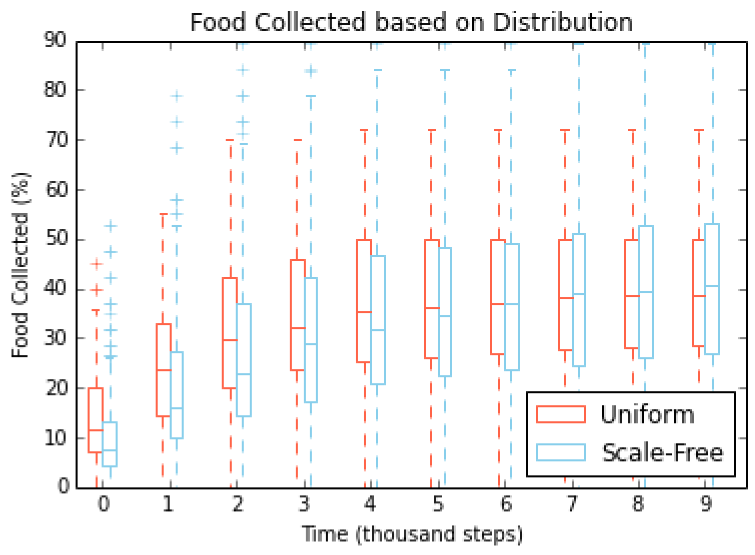


Figure 13: Overall Food Collected by Puller Algorithm in Uniform and Scale-Free Distributions

Figure 13 shows the performance of the Puller algorithm on uniform and scale-free distributions. The robot density, food density, and maximum pile size all ranged with the values specified in Table 1. The massive range presented in Figure 13 for the boxplots is most likely due to the performance of the algorithm being tied heavily to the other parameters such as robot and food densities. In order to better analyze the performance of the algorithm, we took a look at fixed robot densities and food densities, as well as maximum pile size of one which was determined to be the ideal case.

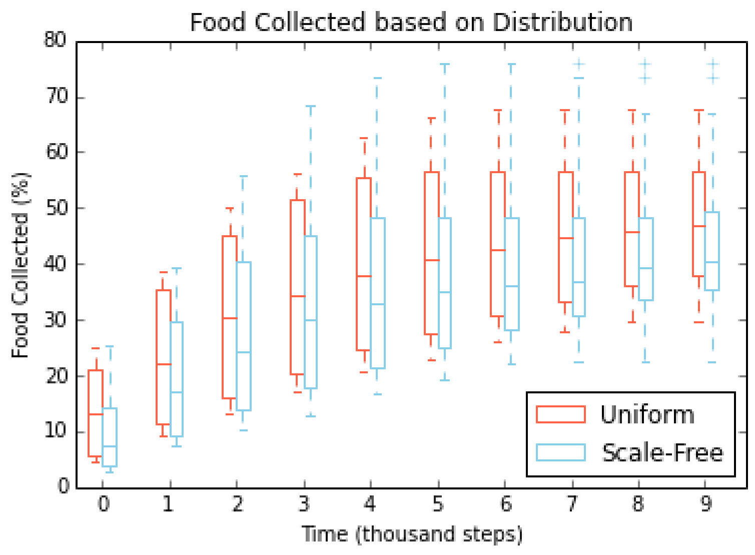


Figure 14: Food Collected by Puller Algorithm in Most Occupied Environment based on Distribution

The figure above shows the performance of the Puller algorithm in the most occupied environment. As we can see, the two series are very similar to each other. The performance with the uniform distribution had a higher mean consistently over the scale-free distribution. However, the performance for scale-free also yielded a better maximum percentage of food collected compared to the uniform distribution. Overall, the two distributions at first glance look fairly similar.

To analyze the distributions further, we applied the wilcoxon rank sum test for the two distributions as well. The wilcoxon rank sum test yields with an average probability of 23.5% for the two distributions being the same. This is a fairly high probability compared to previous probabilities, though the topology of the food seems to have an effect on the Puller algorithm slightly. Overall, the performance boost is so small, that the Puller algorithm performs essentially independently of the topology.

Analyzing Food Location Preferences

In addition to comparing the food collected based on the food distribution and maximum pile size, we also analyzed locations of the environment the algorithm tends to perform better or worse at. In order to do so, we looked at the heatmaps for the positions of the food collected in hundreds of runs.

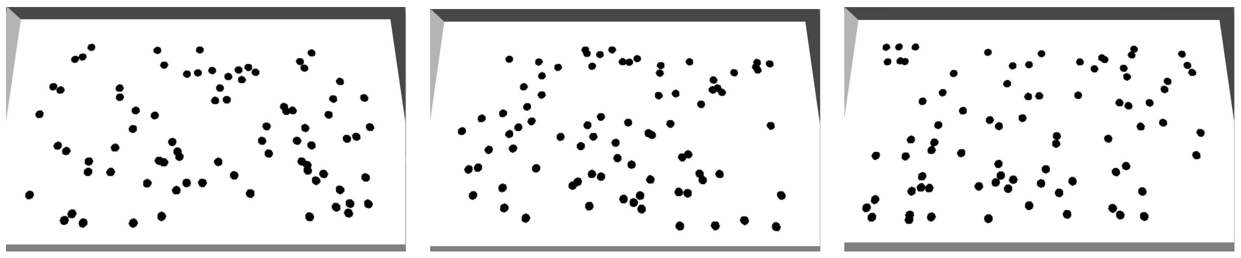


Figure 15*:* Random Uniform Distributions in ARGoS

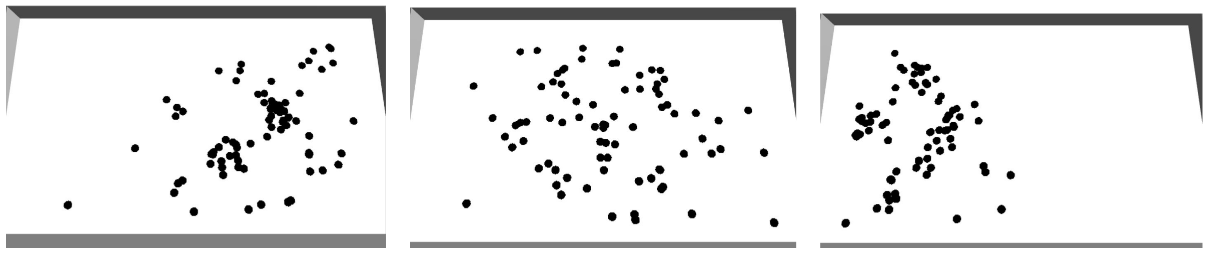


Figure 16*:* Random Scale-Free Distributions in ARGoS

Figure 15 and Figure 16 show different runs of uniform and scale-free distributions. As is evident from the figures, scale-free distributions tend to differ a lot more from each other compared to the uniform distributions. This was an important finding as it meant our simulations involving scale-free distributions must have a large enough sample size to accurately depict the performance.

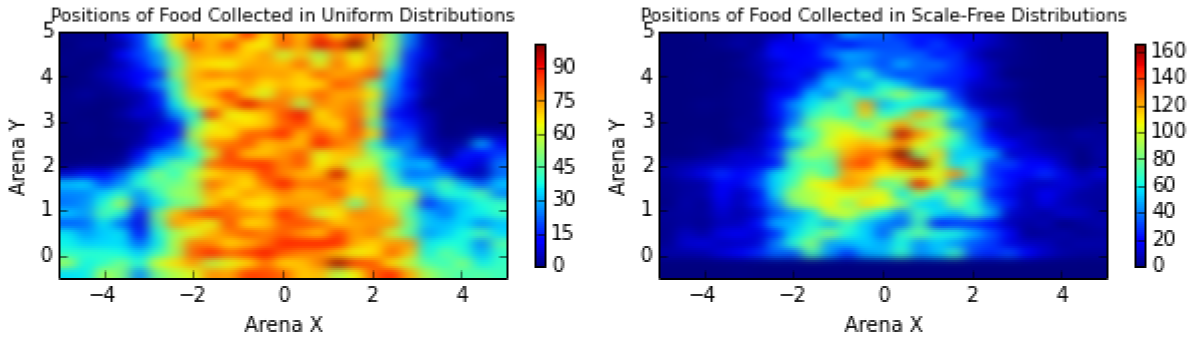


Figure 17: Heatmaps of Collected Food Positions with the Puller Algorithm

Figure 17 shows the positions of the food collected by the Puller algorithm in a total of a five hundred runs for each distribution. The food densities and robot densities were set to the maximum tested values in order to see the effect of the algorithm in the most cluttered environment. In addition, the maximum pile size of the environments was set to one due to the Puller algorithm's preference as mentioned earlier.

As we can see in Figure 17 above, the Puller algorithm prefers food to be located near the central area of the arena and tends to not find food on the far corners of the area in both the scale-free and uniform distributions. This is most likely due to the stretching behavior as even with the variation in angle the robots aren't consistently going to the far corners. However, food in the center of the arena is collected consistently regardless of distance from the nest. Furthermore, the algorithm performs fairly well at the near corners of the arena, though not nearly as well as if the food was closer to the center of the field.

The performance at the far right corners could potentially be improved by distributing more robots to the opposite ends of the arena. In this case, there will be multiple robots designated as the nest rather than one central nest. By having this, robots will no longer crowd toward the center of the field as much and rather spend more time exploring on their respective sides. Another potential fix would be to increase the variance in the angle, however this may cause worse performance on other areas of the environment.

5.4 Forager-Beacon Algorithm Performance

The performance of the Forager-Beacon algorithm was assessed on the results of several hundred experimental runs. The diagrams that follow demonstrate the outcomes of these experiments in terms of several different parameters, such as the amount and location of food collected. Many of these metrics are contextualized against several other variables as well, including the maximum pile size of food and other environmental factors. Together, these measures help to form an accurate overall impression of the Forager-Beacon algorithm's performance.

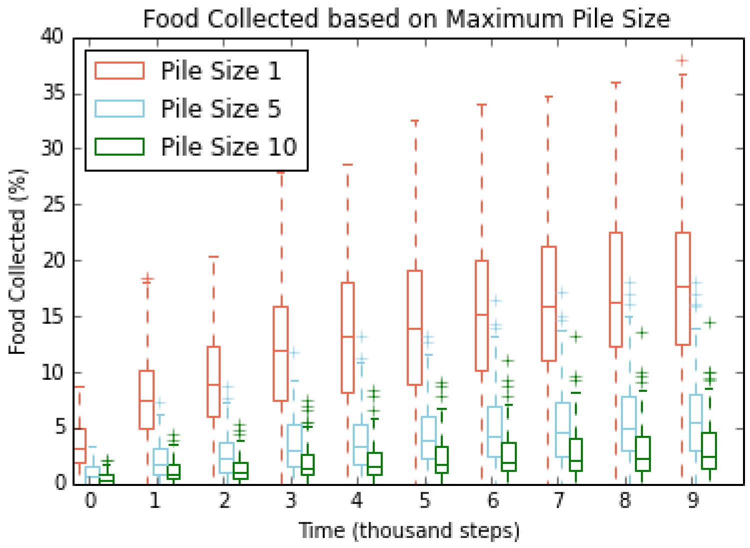


Figure 18: Overall Performance of Forager-Beacon Algorithm for Varying Pile Sizes

The graph in Figure 18 shows the overall performance of the Forager-Beacon algorithm in the form of box plots. The distributions are grouped by maximum pile size, where each of the three options accounted for 160 runs out of a total of 480. This graph indicates that the rate of food collection appears to taper off as the simulation time increases. This can be explained by the notion that the swarm collects the food that is close to the nest (and thus easier to retrieve) earlier in the simulation, leaving only the more inaccessible food as time runs on. The maximum percentage of food collected was achieved in an experimental run with a maximum pile size of 1, where around 38% of the available total was retrieved.

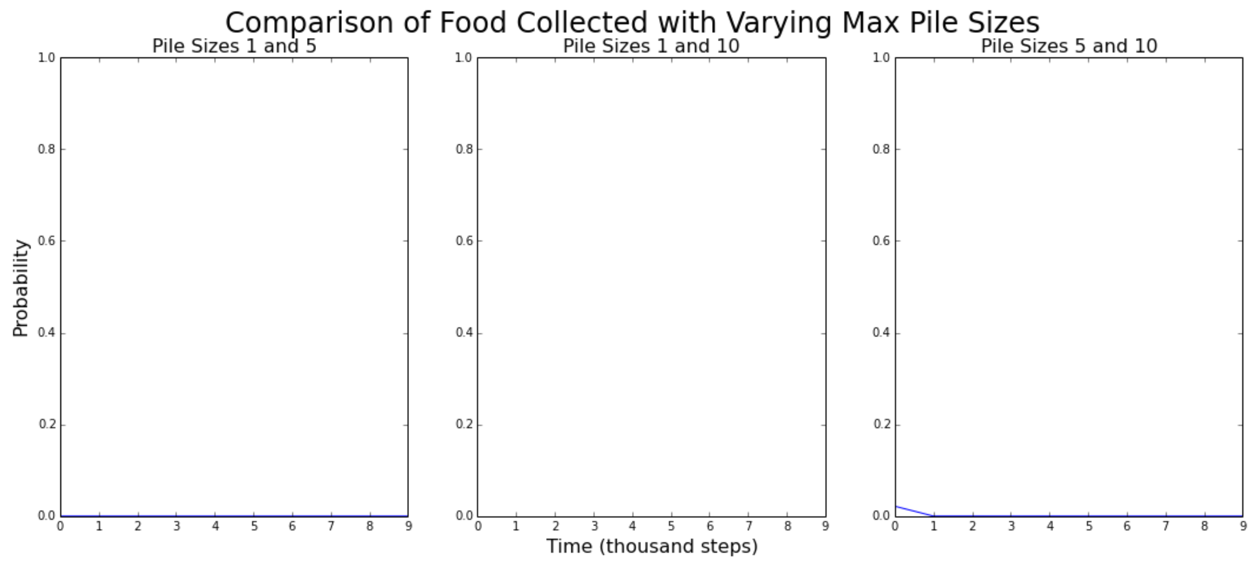


Figure 19: Results of Wilcoxon Rank Sum Test on Different Pile Size Distributions over Time

	1 and 5	**1 and 10**	**5 and 10**
Probability	4.09e-07	4.49e-08	0.0022

Table 3: Average Results of Wilcoxon Rank Sum Test on Different Pile Size Distributions

By merely looking at the figure, it appears that the maximum pile size has an effect on the total percentage of food collected. The distributions of each type of run appear to be distinct, with pile size 1 runs generally collecting the most runs and pile size 10 runs gathering the least. However, given that these results were drawn only from a small sampling, it is hard to draw definitive conclusions about the true nature of these distributions only by using this figure. As before, a Wilcoxon Rank Sum test is useful for determining the probability that any of these distributions are in fact the same. The outcome of these tests are shown in Figure 19 and Table 3 above.

These results suggest that there is an exceedingly low probability of the distributions being the same. The average values in Table 3 show that the that the runs with pile sizes of 1 and 5 have a near-zero probability of having the same distribution. The probability of the pile size 1 and pile size 10 runs is similarly low. This data corroborates the informal picture presented by the graph in Figure 19, where there is very little overlap between these pairs of distributions. This suggests, at least informally, that these distributions are indeed different.

The only variation in this observation comes when comparing the distribution of pile size 10 runs against those with pile size 5. In this comparison, the probability of the two distributions being the same is 0.0022. Though this is a much larger value than the other two Wilcoxon tests, it is still a comparatively low value that suggests these distributions can be treated as distinct. Much as before, this observation is reflected in Figure 19, which appears to show that the distributions of pile size 5 and pile size 10 have only slightly more overlap than in the other comparisons.

The fact that the percentage of food collected improves with decreasing pile sizes is expected. These performance graphs only track the percentage of food collected, rather than absolute amounts. When the food is distributed in the environment, only the total number of piles is considered. Thus an environment where every pile has ten food items can be expected to have ten times the total amount of food compared to an environment where the maximum pile size is one.

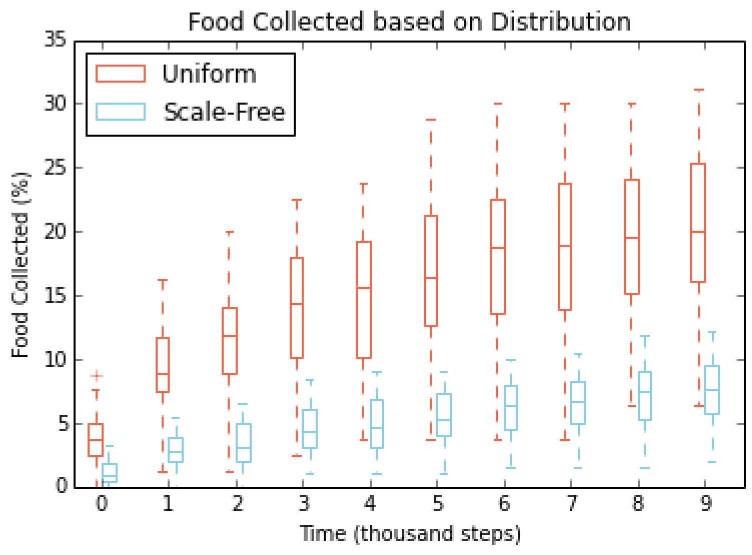
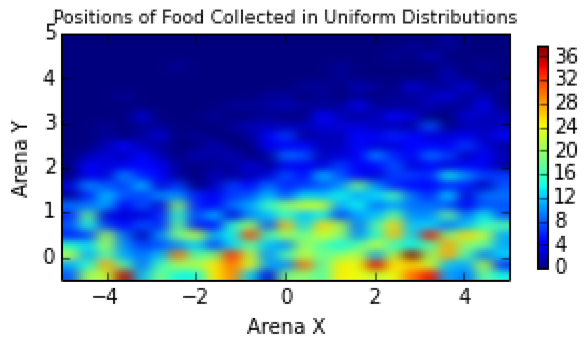


Figure 20*:* Performance of Forager-Beacon Algorithm with Varying Food Topology

The graph in Figure 20 shows that the performance of the Forager-Beacon algorithm also varied depending on whether the food was distributed in the environment uniformly or in a scale-free topology. Exactly half of the 480 total experiments were used for each topology, and runs using each topology were in turn distributed equally among each of the three pile size options. The two distributions depicted in the graph appear to show that the Forager-Beacon algorithm performs much better in environments where the food is distributed uniformly. This observation is largely confirmed by a Rank Sum test comparing the two data sets, which shows that there is only a 0.00000006 probability that the two distributions are the same. The cause of this discrepancy is not immediately clear from this data, but the heatmap diagrams shown in Figure 22 help to explain how this algorithm performs differently depending on the food topology.



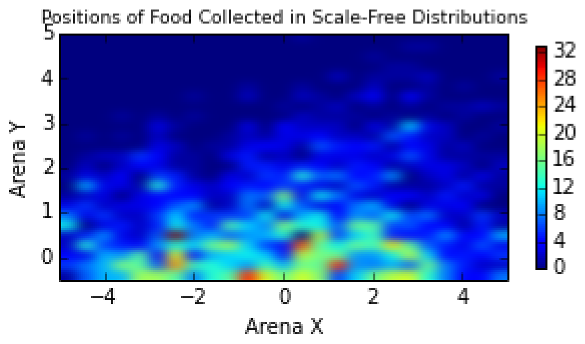


Figure 21: Positions of Food Collected by Forager-Beacon Algorithm within the Arena

These show the relative position of food collected by the Forager-Beacon algorithm within the environment. Areas where food was more regularly collected are indicated by warmer hues, while regions where food was seldom collected are visualised as colder hues. The data used to generate each of the diagrams was collected over 480 experiments, and the results are illustrative of the general behavior of the Forager-Beacon algorithm.

Both diagrams show that regardless of which food topology is used, the algorithm collects much more food in areas near the nest, which is directly below the x-axis in this visualization. This observation can be explained partially by the fact that all of the robots start near this nest area. The meandering behavior used to disperse robots in the swarm is of limited effectiveness in this setup, as the robots do not distribute themselves across the environment quickly. Though the algorithm appears to be quite efficient at gathering food that is placed near the nest, the swarm never covers enough area to find the food in the more distant reaches of the environment. A major reason for this is the fact that the meander behavior tends to cause the robots to spread out radially from their starting point, while in this environment all the robots are positioned towards one end of the field. If the nest was instead located in the center of the environment, it is likely that this algorithm would be able to gather more food.

These diagrams also clearly show that the Forager-Beacon algorithm collects more total food when resources are distributed uniformly. Though both topologies contain the same total number of food items, scale free distributions can often be skewed towards one side of the map. Since this algorithm tends to only collect food close to the nest, some scale-free experiments will collect very little food. Uniformly distributed food tends to create more consistent performance, which results in more food collected overall.

5.5 Honeybee Algorithm Performance

The performance of the honeybee algorithm was measured using similar metrics to those discussed above in Table 1. The data gathered was accumulated over several hundred experiments on the cluster.

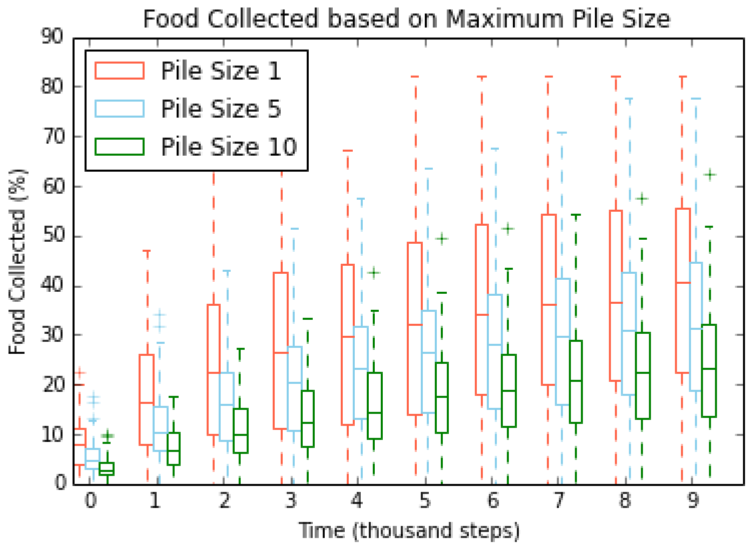


Figure 22: Overall Performance of Honeybee Algorithm for Varying Pile Sizes

The overall performance of the honeybee algorithm displayed above for various pile sizes demonstrates the shape of the collection as a function of time, indicating that as time passes the value levels off at a certain percentage of the food. This fits with the performance of the other algorithms and can be explained by looking at the derivative of the line of best fit with the medians of the box plots. The derivative gets smaller as time increases, indicating that the rate of food collection decays as time increases. This aligns with the observation that for the honeybee algorithm, food collection at first is relatively fast due to the recruitment behavior, as multiple robots flock to one food location and can very quickly bring an entire pile back to the nest. As time passes, robots run out of food at a pile and must then search for more, which causes a dropoff in the collection rate.

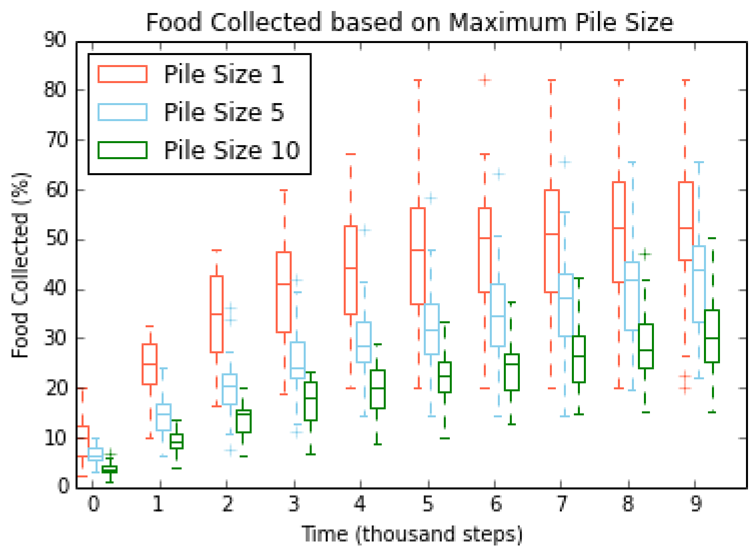


Figure 23*:* Performance of Honeybee Algorithm for Varying Pile Sizes in Most Occupied Environment

Figure 23 shows the results of running the honeybee algorithm with the environment parameters set to maximize the food density. At peak performance, the algorithm was able to collect just above 60% of the food in the arena.

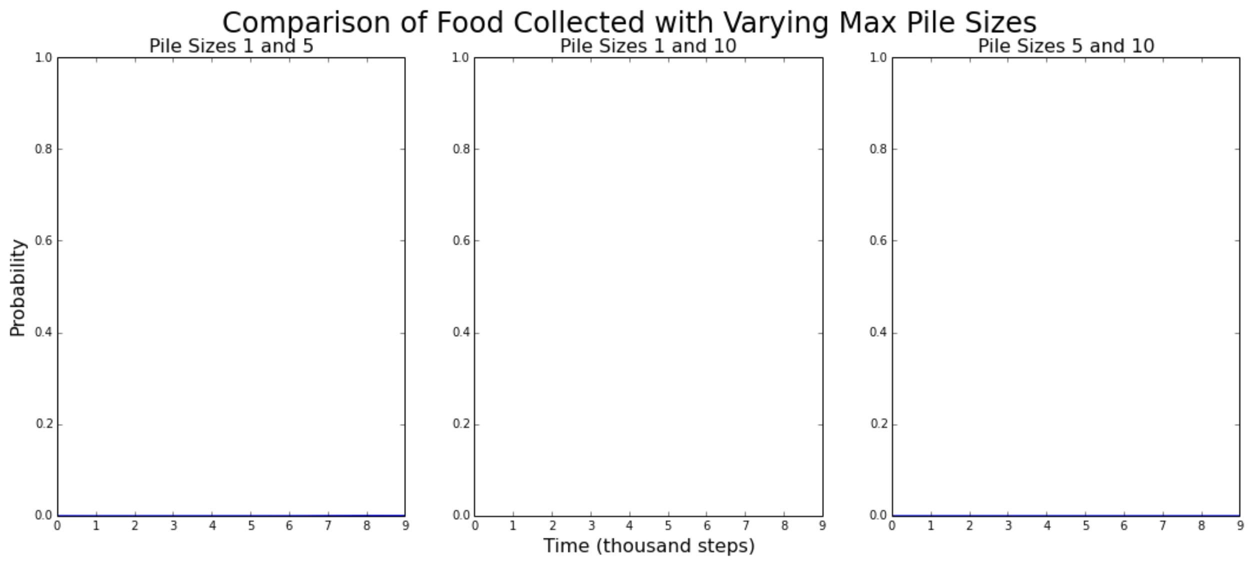


Figure 24: Results of Wilcoxon Rank Sum Test on Different Pile Size Distributions over Time

	1 and 5	1 and 10	5 and 10
Probability	0.0001574	1.647e-07	5.467e-06

Table 4: Average Results of Wilcoxon Rank Sum Test on Different Pile Size Distributions for Honeybee Algorithm

Using the rank sum test as with the other algorithms, it can be seen that the distributions of each pile size have negligible correlation; that is to say that each set of runs is statistically different and thus the conclusion can be drawn that the pile size has a major effect on the performance of the honeybee algorithm. As with the other algorithms, the difference in collected percentages between pile sizes can be explained by recognizing that the data is a percentage and that for larger pile sizes, there is more total food and thus foraging the same amount of food will yield a lower percentage. However, there is more to the performance than just that trivial factor, as the rank sum test shows.

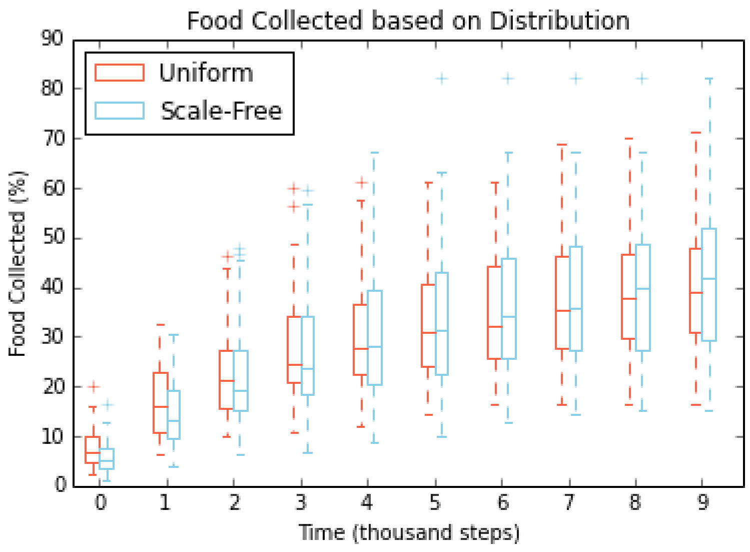


Figure 25: Performance of Honeybee Algorithm with Varying Food Topology

Figure 25 shows the resultant box-plots of running the honeybee algorithm with different topologies. As can be seen in the image, the algorithm performs roughly the same regardless of the topology of the food, collecting roughly the same amount in both cases after running for the same amount of time. This suggests that the strength of the algorithm lies in its decoupling from the environment, leading it to be a more general-purpose algorithm for foraging.

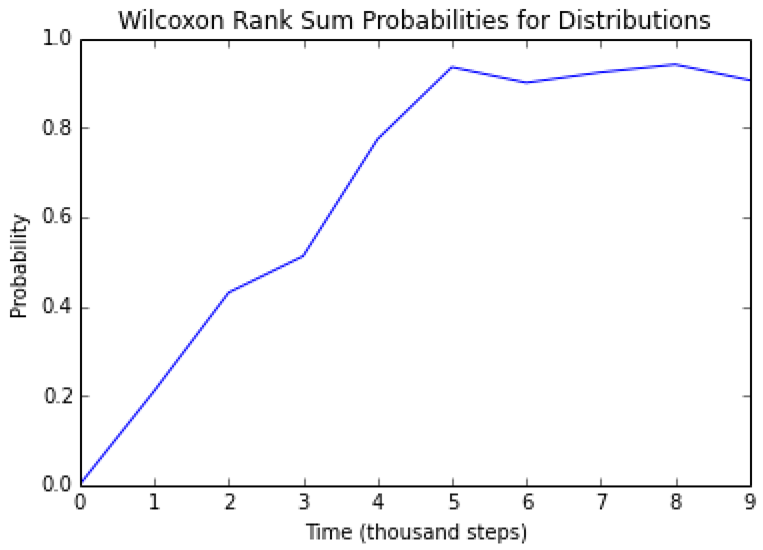


Figure 26: Wilcoxon Rank Sum Test for Different Food Topologies

The rank sum test shown in Figure 26 corroborates this observation, showing that the distributions of uniform and scale-free topologies is very similar for this algorithm. It is clear to see in Figure 25 that the medians and the ranges for both the distributions are consistently similar. Figure 26 further proves the point with the average probability of the two distributions being 65.3%. This high chance confirms that the honeybee algorithm's performance is independent of the topology of the food.

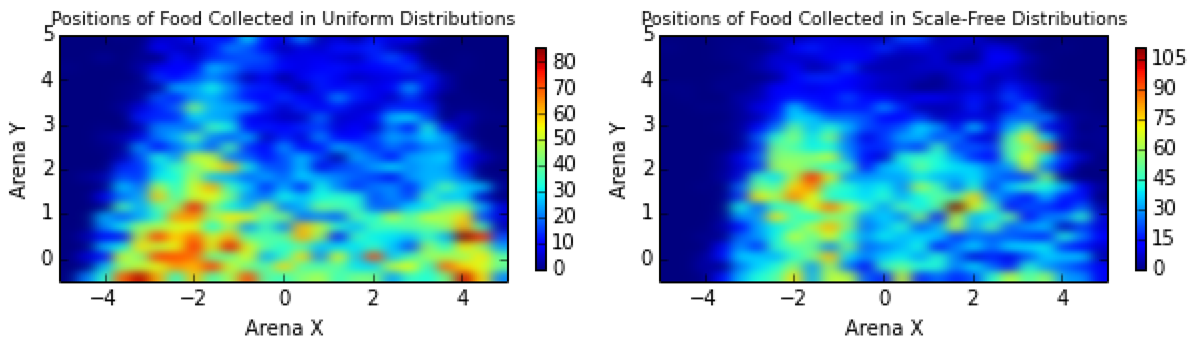


Figure 27: Positions of Food Collected by Honeybee Algorithm within the Arena

Figure 27 shows a heatmap of the positions of the collected food by the honeybee algorithm in all 500 runs. Based on the figure, the algorithm mostly focuses near the nest and towards the center of the arena, and isn't able to reach the the far corners of the arena. Both the heatmaps for the uniform and scale free distributions have a similar shape, which tells u that regardless of the topology, the effective path of the algorithm and the positions of the food collected remain the same. This was the same result that we found when comparing the performance of the actual distributions in Figure 25. However, for uniform distributions, the honeybee algorithm performed better toward the left corner of the field. This is likely due to the dispersion pattern of the scouts at the beginning of each experiment, which seemed to favor a distribution of scouts skewed toward the left side of the field. This would in turn lead to more food on the left hand side being discovered and subsequently foraged.

5.6 Comparison of Foraging Algorithms

The performance of each foraging algorithm helps to illustrate their respective advantages and disadvantages. For example, the performance of the Forager-Beacon algorithm appears to be somewhat inferior to the Puller and Honeybee algorithms based on the total percentage of food collected. The median percentage collected is significantly lower compared to the these algorithms for all three maximum pile sizes, and even the most favorable runs peak between 35% and 40% of the total food available. While this result is understandable in situations where the maximum pile size is one, it is somewhat counterintuitive for scenarios where piles contain multiple food items. The Forager-Beacon algorithm actively recruits nearby robots to the locations of piles with many food items, while the Puller algorithm treats all piles equally. As such, one could reasonably expect the Forager-Beacon algorithm to perform better in situations where piles contain multiple food items. However, the results obtained from experimentation seem to contradict this reasoning.

Part of this discrepancy can be attributed to the fact that the setup of the environment is unfavorable to the behavior of the Forager-Beacon algorithm. Foragers tend to spread out radially, and thus optimal performance would require a nest in the center of the environment instead of towards one end of the field. The meander behavior used in the Forager-Beacon algorithm is also partially responsible for this performance gap. This particular random walking method does a poor job of scattering robots in the swarm across the environment. This is especially true in contrast to the dispersion behavior used in the Puller algorithm. Where the meander behavior only spreads robots apart from each other passively, the dispersion behavior actively forces them apart. The net effect on overall performance is that robots executing the Forager-Beacon algorithm never extend deep enough into the environment to collect food. This explains the relatively small field of exploration shown in the heat map diagrams of the Forager-Beacon algorithm when compared to the two other options.

For the topology of the food, some algorithms preferred a certain distribution over the other. The Forager-Beacon algorithm performed better in uniform distributions, while the Puller and Honeybee algorithms performances were independent of the distribution. This result has a large effect on the higher level switching algorithm as if the robots find food evenly distributed, then they may execute the Forager-Beacon. On the other hand, if robots find clusters of food near each other, the Puller or Honeybee algorithm would be more optimal.

Another trend that emerges from these results is the notion that different algorithms seem to perform better under different conditions. Looking at the heatmaps for each of the three foraging behaviors, it appears that the Forager-Beacon algorithm excels at collecting food in close proximity to the nest, the Honeybee algorithm is the most effective at farther away food, and the Puller algorithm seems to be most successful in intermediate ranges. An algorithm that could combine these qualities would be much more successful at gathering food across the entire environment.

Another important consideration is the rate at which the algorithms were collecting food. While the Puller algorithm may have collected a larger percentage of the food overall compared to Forager-Beacon and Honeybee, the performance for the algorithm started plateauing towards the end of the experiment. On the other hand, both the Forager-Beacon and the Honeybee algorithms were still finding more food towards the end of the experiment. This can be explained by the dispersion module present in the Puller algorithm. After enough time steps, the robots will have dispersed far enough from each other that the net force they experience is zero. In this case, the robots would not be able to explore further areas. While the stretching behavior does break this condition, it may not be happening consistently and thus the Puller algorithm slows down rapidly. A higher level algorithm that decides which of these to use may then decide to execute the Puller algorithm for a shorter duration than the other two algorithms. This would allow for quick retrieval of food, while also maintaining a steady rate of food collection.

6. Conclusions and Recommendations for Future Work

In general, the foraging algorithms that comprise the finalized library largely meet the initial goals set for this project. At a basic level, each of the algorithms works as intended, and completes the same basic task in a different way. Many of the smaller sub-behaviors are shared and reused between algorithms, and many of these self-contained, interoperable modules that can be used on their own. These design choices allow future users considerable freedom in how they use this library. Not only is it possible to use the foraging algorithms as they are described in this paper, but future developers could easily construct their own novel behaviors by reusing the myriad of modules and components included in this library. It is also possible to create new modules that improve upon the ones created in this project, and then simply replace them into the foraging algorithms to create a better result.

6.1 Suggestions for Future Work

While the library in its current form provides modular, reusable base upon which users can build their own projects, there are certainly areas that could be improved or extended. For example, further testing can be done on the algorithms under different environment conditions. Our environments were static and always assumed a single nest in the same layout as shown in Figure 2, where the food area is north of the nest area. It would be useful to adapt these foraging algorithms to other experimental setups for the purposes of making them more robust. Robot could be distributed in other methods than the current one of distributing them in a line, which could provide a different perspective on the performance of each of the algorithms. Further testing in environments with multiple nests would provide a deeper evaluation for the performance of each of the algorithms. Creating and testing environments where the nest is centrally positioned-- as opposed to near the edge of the environment-- would be likewise useful in characterizing the overall performance of foraging algorithms.

The contents of this library could also be easily extended to other behaviors beyond foraging as well. Although foraging is a common, generic swarm behavior with many applications, it is far from the only important problem in the literature. The construction problem, where robots gather resources that they then assemble into structures and other arrangements, is a natural extension of the foraging problem. A construction-related extension would thus be a sensible next step for future extensions to the library.

Conducting a more rigorous reusability evaluation would also help to strengthen the overall utility of this project. The results of this project included a limited analysis of the degree to which modules were reused between algorithms, but it would also be a useful exercise to construct an entirely new algorithm or behavior from the modules that already exist. This would be a much more emphatic demonstration that the components in this library are, in fact, reusable.

At a higher level, it would also be interesting to create a higher level algorithm that switches between foraging behaviors according to environmental conditions. The results of the three algorithms included in this library suggest that each algorithm seems to perform better relative to other algorithms depending on the distance of food from the nest. It is also possible, for example, that one algorithm performs better in food-sparse environments while another excels in food-dense ones. Creating a higher-level switching algorithm would allow for the creation of an even more effective foraging behavior that can dynamically adjust to its environment. This would in turn provide more utility and flexibility to future users of the library.

It is worth noting that all of the results gathered in this project came from simulations rather than experiments on physical robots. As future users might seek to perform foraging within a physical environment, it would be very useful to demonstrate these foraging algorithms on actual robots as a proof-of-concept. All of the Buzz code that runs on the simulated robots should theoretically perform the same on actual robots, but non-trivial work is required to recreate the simulated food and environment in reality.

7 . References

[1] C. Pinciroli and G. Beltrame, “Buzz: A Programming Language for Robot Swarms,” IEEE Software, vol. 33, no. 4, pp. 97–100, Jul. 2016.

[2] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, “Swarm robotics: a review from the swarm engineering perspective,” Swarm Intell, vol. 7, no. 1, pp. 1–41, Mar. 2013.

[3] A. Campo et al., “Artificial pheromone for path selection by a foraging swarm of robots,” Biological Cybernetics, vol. 103, no. 5, pp. 339–352, 2010.

[4] Q. Lu, J. P. Hecker, and M. E. Moses, “The MPFA: A multiple-place foraging algorithm for biologically-inspired robot swarms,” 2016, pp. 3815–3821.

[5] N. Hoff, A. Sagoff, R. Wood, and R. Nagpal, “Two Foraging Algorithms for Robot Swarms Using Only Local Communication.” .

[6] M. Alcherio et al., “Distributed Colony-Level Algorithm Switching for Robot Swarm Foraging,” in Distributed Autonomous Robotic Systems, pp. 426–429.

[7] A. Brabazon, M. O’Neill, and S. McGarraghy, “Other Foraging Algorithms,” in Natural Computing Algorithms, Springer, Berlin, Heidelberg, 2015, pp. 171–186.

[8] C. Pinciroli et al., “ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems,” Swarm Intell, vol. 6, no. 4, pp. 271–295, Dec. 2012.

[9] L. Pitonakova, R. Crowder, and S. Bullock, “Behaviour-Data Relations Modelling Language For Multi-Robot Control Algorithms,” 2017.

[10] E. Bonabeau, A. Sobkowski, G. Theraulaz, J.-L. Deneubourg, and others, “Adaptive Task Allocation Inspired by a Model of Division of Labor in Social Insects.,” in BCEC, 1997, pp. 36–45.

[11] G. Pini, A. Brutschy, M. Frison, A. Roli, M. Dorigo, and M. Birattari, “Task partitioning in swarms of robots: an adaptive method for strategy selection,” Swarm Intelligence, vol. 5, no. 3–4, pp. 283–304, 2011.

[12] James D. McLurkin, “Stupid Robot Tricks: A Behavior-Based Distributed Algorithm Library for Programming Swarms of Robots,” Massachusetts Institute of Technology, 2004.

[13] A. Reina, G. Valentini, C. Fernández-Oto, M. Dorigo, and V. Trianni, “A Design Pattern for Decentralised Decision Making,” PLOS ONE, vol. 10, no. 10, p. e0140950, Oct. 2015.

[14] G. Francesca, M. Brambilla, A. Brutschy, V. Trianni, and M. Birattari, “AutoMoDe: A novel approach to the automatic design of control software for robot swarms,” Swarm Intell, vol. 8, no. 2, pp. 89–112, Jun. 2014.

[15] “Distributed Exploration with a Robot Swarm « ASCENS Project Blog.” [Online]. Available: http://blog.ascens-ist.eu/2015/01/distributed-exploration-with-a-robot-swarm/index.html. [Accessed: 28-Sep-2017].

[16] Q. Lu, M. E. Moses, and J. P. Hecker, “A Scalable and Adaptable Multiple-Place Foraging Algorithm for Ant-Inspired Robot Swarms,” arXiv:1612.00480 [cs], Dec. 2016.

[17] E. Şahin, “Swarm Robotics: From Sources of Inspiration to Domains of Application,” in Swarm Robotics, 2004, pp. 10–20.

[18] G. Beni, “From Swarm Intelligence to Swarm Robotics,” in Swarm Robotics, 2004, pp. 1–9.