Multi–Robot Foraging for Swarms of Simple Robots

A dissertation presented
by

Nicholas Roosevelt Hoff III

to
The Department of Computer Science
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts
May 2011

Thesis advisor
Radhika Nagpal and Robert Wood

Author
Nicholas Roosevelt Hoff III

Multi–Robot Foraging for Swarms of Simple Robots

Abstract

In multi–robot systems, a large number of individual robots work together to perform a task. Large swarms of robots have the potential to be useful for types of tasks for which traditional robots are not well suited, such as large area surveillance, distributed contaminant cleanup, and in–vitro medical applications. Although a variety of possible robots could be used in such swarms, the large numbers involved usually force each robot to be small and simple. This dissertation focuses on robots with very simple hardware capabilities. In this work, the robots are not assumed to have global position information, global communication, odometry, or a central leader, and their only communication capabilities are simple local neighbor–to–neighbor communication. To overcome their hardware limitations and leverage their large numbers, the robots must work together.

This dissertation is focused on foraging as a multi–robot task. Robots start at a home location, explore the world in search of their target, and return the target incrementally to the home. It is difficult in swarm robotic systems because of the lack of global localization, communication, and odometry, which make it impossible for the robots to acquire or build maps, for example. Once a robot loses contact with the other robots, it is effectively lost and has no way (other than random movement) to return home.

This thesis begins by developing three distributed foraging algorithms for robot swarms and analyzing them in simulation. The performance of each algorithm is strongly affected by the environment in which the swarm operates, specifically the distance separating the home from the target. To allow the swarm to forage in different environments, a method is developed by which the swarm as a whole can choose its algorithm based on the home–target separation it encounters. Finally, I consider the feasibility of running these algorithms on physical robots. Adding accurate sensors and actuators to robots increases their cost, making it desirable for the robots to be as simple as possible while still being able to execute the algorithms. I directly measure the effect of sensor and actuator quality on swarm performance.

Contents

Bibliography **92**

Citations to Previously Published Work

Most of the work presented in this dissertation has been published in the following places:

Two Foraging Algorithms for Robot Swarms Using Only Local Communication, Nicholas Hoff, Amelia Sagoff, Robert J. Wood, Radhika Nagpal, IEEE International Conference on Robotics and Biomimetics, ROBIO 2010

Distributed Colony-Level Algorithm Switching for Robot Swarm Foraging, Nicholas Hoff, Robert Wood, Radhika Nagpal, International Symposium on Distributed Autonomous Robotic Systems, DARS 2010

Effect of Sensor and Actuator Quality on Robot Swarm Algorithm Performance, Nicholas Hoff, Robert Wood, Radhika Nagpal, International Conference on Intelligent Robots and Systems, IROS 2011 (in review)

Acknowledgments

I would like to first thank my advisors, Radhika Nagpal and Rob Wood, for their consistent guidance and support throughout my PhD research at Harvard. They truly are model advisors – emerging leaders in their fields and top–notch researchers, and also friendly and thoughtful mentors. They encouraged me to run with my ideas while simultaneously guiding me through a young and newly–developing field.

It was my good fortune to have David Parkes on my thesis committee. I thank him for his insights into my work, its connections to other areas of AI, and his suggestions for possible directions I could take my research. David has the remarkable ability to give insightful comments on a piece of work after a brief discussion. His feedback has expanded my viewpoint on multi–agent systems.

It has been a pleasure to be a part of the Harvard SSR group, led by Radhika, and the Harvard Microrobotics Lab, led by Rob. Fellow students and postdocs in these groups have given me fun memories and valuable research feedback, including Tyler Gibson, Kirstin Petersen, Michael Rubenstein, Spring Berman, Christian Ahler, Nils Napp, Mark Woodward, Neena Kamath, Justin Werfel, Daniel Yamins, Ankit Patel, Julius Degesys, Ian Rose, and Chih-Han Yu.

My mother has been especially supportive, giving me advice and helping me see the bigger picture. There are no words to describe the never–failing love she has given me my whole life.

Finally, I thank my girlfriend Sabine Leisten for her constant loving support. Being together has sustained me throughout my work. She has given me love, advice, and encouragement, and I aim to do the same for her.

Chapter 1

Introduction

Classical robots are usually single complex pieces of electromechanical hardware designed to perform a task. They are typically expensive, specialized, and complicated. An alternate approach is to use a "swarm" composed of a large number of very simple robots that work together, rather than a single highly capable robot. In this approach, the strength of the system comes not from the complexity and power of a piece of hardware, but from the cooperation of a large number of individuals. As the cost of simple robotic components goes down and availability goes up, this approach becomes increasingly feasible. It is currently possible, for example, to build a swarm of 100 small robots, each of which has a few simple sensors and can move in an environment.

The transition from a single complex robot to swarms of multiple simple robots brings both benefits and drawbacks. It reduces the design and hardware complexity of each individual robot, which reduces cost and design complexity and increases robustness. The savings in hardware complexity, however, are paid for with an increase in algorithm difficulty because of the need for coordination. An algorithm no longer instructs a robot how to do the whole task. Instead, each robot runs a simple algorithm using its simple capabilities, and the interaction of all these robots must produce the desired result. The challenge is to program a collection of simple robots, with simple communication and individual capability, to perform a useful task as a collective.

1.1 Natural to Artificial Systems

Large collections of simple individuals are ubiquitous in nature, making it a natural place to look for inspiration. Consider schools of fish as a simple example. The school must avoid predators, but no individual fish can be sure to see the predator, they can not communicate what they see to all the other members of the school, and even if they were all aware of the presence and location of a predator, there is still no

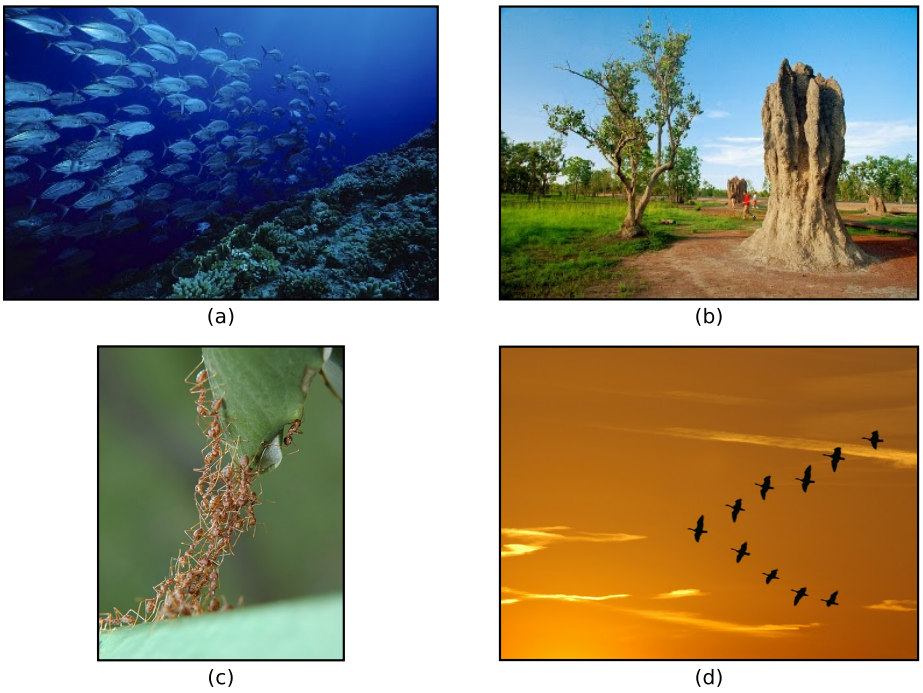


Figure 1.1: Several examples of group behaviors in nature: fish, termites, ants, and birds. (All images from iStockphoto.com.)

central leader to tell them how to swim to avoid it. Through local sensing, local communication, and distributed decision making, the school is able to operate as a whole to avoid predators [19, 40].

Termites exhibit a different kind of collective behavior. Instead of working together to avoid predators, they work together to construct and maintain their nest. The nests can be several meters tall with remarkable internal complexity (including thermal regulation, ventilation, a garden, a nursery, etc.) [8, 4]. No individual termite is in charge, no termite knows the overall plan, each termite has only simple sensing and communication, and environmental conditions are constantly changing, but somehow the collective manages this amazing feet of construction.

The foraging behavior of social ant colonies provides a third example. Ants are capable of searching a very large area for potential food sources, and are capable of returning food to their nest even though it may be very far away. They can build a trail from nest to food even though no individual ant knows its own location or the food's location, and no ant is in charge [24, 17]. In each of these examples, there was no central leader with all the information making decisions for each individual. Leaderlessness is a central aspect of distributed swarm algorithms.

These biological systems are a useful place to begin in the effort to build multi–robot systems because the restrictions on physical capability of the individuals are similar. If a physical system will consist of 1000 robots, each robot must be simple.

Coordination would be easy if each robot could sense the whole world and communicate with all other robots, but the hardware required for this would be far too complex to place on each of 1000 robots. Instead, the robots should develop group behaviors relying only on simple sensing and communication, which is exactly what many biological systems do. Perhaps we can leverage the strategies used in nature in order to build artificial swarms.

1.2 Multi–Robot Foraging

There are many possible applications for multi–robot swarm–like systems, including collective construction, coverage, self–assembly, collective transport, formation control, and search. In this dissertation, I focus on search–and–return as an application, based on ant–like foraging. The goal is to develop an algorithm which can run on a large number of simple robots and enable the collective to search its environment for a target and to incrementally return the target to the nest.

This would be useful in several situations. Consider a collection of robots exploring Mars [9]. To explore a large area, engineers have opted for a multi–robot system, but because of mass and complexity constraints, each robot must be very simple. The robots must depart their landing module and explore the surface for items of scientific interest, for example, a collection of rocks in a crater wall. Once such a site is found, the robots must pick up the rocks (each robot can only carry a small amount) and return them to the landing module for analysis. There is no GPS and while they are exploring they loose sight of the landing module, so they are essentially lost. They must work together not only to find what they are looking for, but also to figure out where they are and how to get the rocks back to the landing module.

Another example where a foraging behavior would be useful is in environmental cleanup. Consider a situation in which a toxic contaminant has been spilled somewhere, but the location is unknown. A collection of robots could search the area and locate the contaminant. Once found, the robots would build a trail from their base to the site of contamination. Using the trail, they could bring a neutralizing agent from the base to the site of contamination, or simply return the contaminant to the base for disposal, depending on the particular situation.

1.3 Challenges

In foraging specifically and swarm robotics in general, coordination between robots is critical because performing the desired task is beyond the capabilities of any single robot. The robots must work together, coordinating their actions. (Random algorithms are an extreme example requiring no coordination, and are used as a comparison throughout this dissertation.) The problem is amplified by the lack of significant sensing and communication capabilities on the robots. In a foraging task,

when a robot finds the target, that information is not immediately useful for two reasons. First, the robot does not know where it is because of the lack of an absolute position system, and second, even if it did, it could not broadcast this information to the rest of the robots because of the lack of a global communication system. Ants solve this problem using pheromone marks in the environment which provide a means of sharing information with other individuals. Developing a coordination method for robots, and using this method to build a foraging strategy, is a central challenge in robotic foraging algorithms.

Foraging can be treated as a broader problem, however, than just a coordination mechanism and foraging algorithm. We want the swarm to forage and operate as a whole, sensing its environment and adapting its behavior. If the swarm detects that one particular foraging algorithm is not working because of some environmental effect, it should decide to switch algorithms and try a different method. This requires significant sensing and decision making on the swarm level. It is likely, for example, that no individual robot will be able to sense that a foraging strategy is not working. There will also probably not be a central leader to make the decision about when to switch algorithms. Developing a collection of robots which can operate as a whole collective is a central challenge of swarm robotics.

When constructing large swarms of robots, there is significant pressure for each robot to be simple and inexpensive, which is the root cause of many of the challenges in swarm algorithms. Cost, power, and manufacturing concerns all drive swarm robots toward fewer sensors, less computational capability, and less accurate locomotion. This research ultimately drives toward swarms of robots that weigh $\sim$5 g and cost $\sim$\$10 each. In this realm, every gram, penny, and milliwatt counts. Locomotion tends to be unreliable (either from walking legs or vibration, for example), and there are few sensors.

Small robots with simple capabilities (with a size of the order of 10cm) are common in swarm robotics research. When moving toward microrobotics (size of the order of 1cm), the hardware capabilities are even more severely restricted. One of the long–term goals of multi–robot swarm research is to build collections of microrobots which can run swarm algorithms and work together. The work in this dissertation is driving ultimately toward microrobotic platforms, which is why the focus here is on such simple hardware capabilities. Figure 1.2 shows several of the small robots currently in use for research, as well as current prototypes of several microrobots.

1.4 Dissertation Contents and Contributions

In this dissertation, I focus on foraging as a multi–robot task. This work is part of a broader effort to develop collective algorithms which will ultimately run on microrobots such as the tiny flying and crawling robots pictured in Figure 1.2. Swarms of these robots will need to navigate, maintain adequate power reserves,

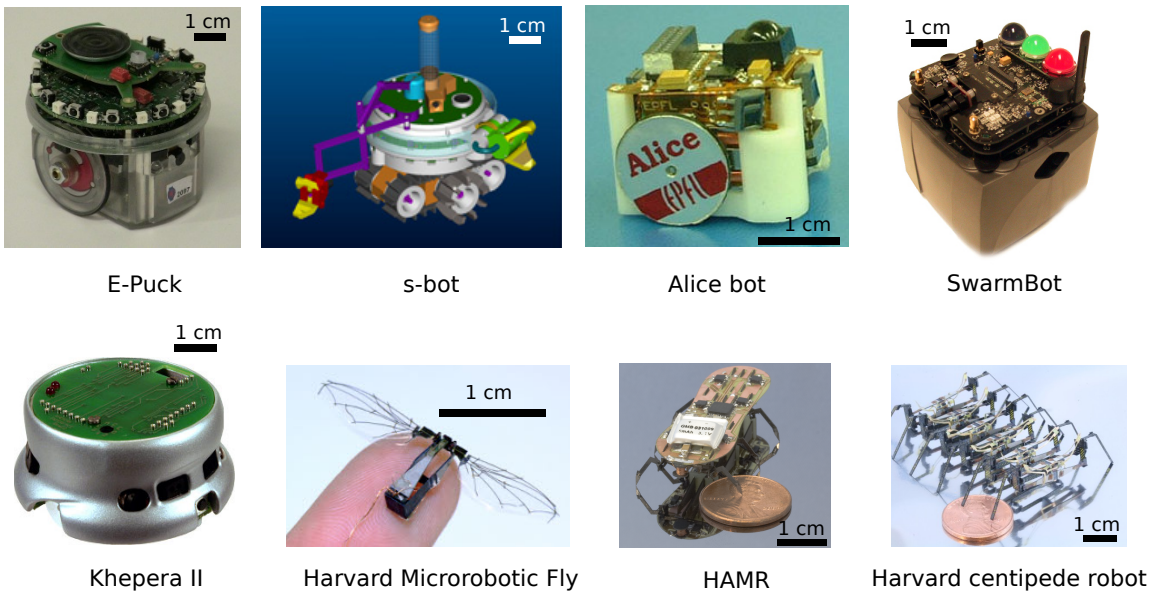


Figure 1.2: Several small robots in use for swarm robotics research. E-Pucks and Kheperas are commonly used general purpose research robots. The S-bot[15], Alice bot [10], and SwarmBot [34] platforms were developed by researchers for use in their specific projects. (S-Bot photo kindly provided by Marco Dorigo, coordinator of the Swarm-bots project. Alice bot photo used with permission of Gilles Caprari. SwarmBot photo used with permission of James McLurkin. Harvard Ambulatory MicroRobot (HAMR) photo kindly provided by Andrew Baisch. Harvard centipede robot photo kindly provided by Katie Hoffman.)

recharge, search their environment for items of interest, and perform simple tasks (repair structures, pollinate flowers, observe a subject). Foraging is an important subset of the multi–robot algorithms which will be required on such a system. This dissertation is concerned with multi–robot foraging algorithms and their hardware implications for a swarm of very small robots.

With such goals in mind, I develop three swarm foraging algorithms for robots with simple sensing and actuation, and analyze them in simulation. Two of these algorithms are gradient based (covering an area with a navigable gradient to and from the search target), and one is physics based (using virtual forces to create a line of robots which sweeps the world). Running these algorithms, the swarms are able to find and return their target (usually referred to as "food" because of the connection to ant–like foraging). The number of robots in the swarm and the distance between the nest and the food are important factors, and their effect is investigated. As expected, these various algorithms perform differently depending on how far the nest and food are from each other, with some algorithms working best when the food is near and others when the food is far away. The dissertation continues by developing a system by which the swarm as a whole can choose the best algorithm for the given situation. This requires distributed sensing and coordination to detect if the algorithm should be changed, and if so, to synchronize the change.

Next, because hardware capability is such a driver in swarm robot systems, I will study the effect of hardware quality on swarm performance. The robots are under significant pressure to be simple to construct, but must still be able to execute the algorithms. This raises the question, how good does the hardware need to be? For example, it may be desirable from a cost and fabrication standpoint to use inexpensive and inaccurate motors for locomotion, but a robot may not be able to reliably follow a path if its movement is 50% random. There is a design tradeoff between hardware quality and algorithm performance. To measure this relationship, I implement several common swarm primitives on a swarm of robots, reduce the quality of the hardware, and measure the effect on performance.

This dissertation makes the following contributions:

- It develops three distributed swarm foraging algorithms for use on a collection of robots with a specific simple hardware model. The algorithms achieve a coordination benefit, yielding better performance than the robots could achieve working alone. Gradient and physics–based strategies have been used before to control robot swarms, but they have required more sensing and communication capability on the part of each robot than I assume. The contribution is the adaptation of these general strategies for use on a simple robot and an analysis of the pros and cons of using these strategies for foraging.

- It develops a framework for integrating multiple swarm foraging algorithms and allowing the swarm to decide as a whole how and when to switch between them. Different algorithms perform best in different environmental situations,

and allowing the swarm as a whole to choose which algorithm to run can combine the benefits of multiple foraging strategies by choosing the right algorithm for the given situation. Swarm–level algorithm switching is not always possible; there are requirements on the communication structure of the swarm and the sensors on each robot. "Triggers" can be defined for each switch from algorithm to algorithm, and the robots must have the sensors and communication to be able to detect and share these triggers.

- The quality of the sensors and actuators on the robots in a swarm affects the performance of an algorithm running on that swarm, and this effect is measured quantitatively. The dissertation explores the design tradeoff between hardware quality and algorithm performance. The results suggest a design point at which hardware quality has been reduced (reducing construction cost and complexity), but performance has not appreciably suffered.

1.5 Related Work

This dissertation draws from and builds on other work in the areas of multi–robot systems and biologically–inspired algorithms. These are broad fields, involving a wide range of applications and hardware platforms [20]. The hardware capabilities of the robots are critical, and there is a sub–field for multi–robot algorithms in which the robots are very simple. If the robots are capable of GPS, global communication with a central leader, and powerful senors and actuators, for example, the foraging algorithms would be fundamentally different from the ones studied in this dissertation. Here, I survey the simple–hardware realm of multi–robot foraging algorithms.

1.5.1 Foraging as a Task

In the foraging task, a large number of individuals must search the environment for some search target and return it to their base. The parallel to ant–like foraging is obvious, and multiple researchers have studied foraging and other pheromone–mediated coordination in ants [4, 24, 43, 42], and have reduced this behavior to algorithms [18, 46, 12, 29]. When translating to robots, there has been some work assuming that the robots have global communication, but the focus has mostly been on foraging methods involving local communication or leaving information in the environment [3]. Establishing a means of communication and coordination is a central challenge, especially in the simple–hardware regime.

1.5.2 Foraging Methods for Robots

This section covers work related to three robot foraging strategies: gradient–based foraging, area–sweeping foraging, and adaptive foraging. Gradient–based foraging

and area–sweeping foraging are built on different strategies of achieving coordination between robots, and adaptive foraging is a method of combining multiple behaviors in one overall algorithm.

Gradient–based Foraging Gradient-based foraging methods create a gradient leading to the goal using the sensing capabilities of the robot, such as chemical sensing or communication. Algorithms using many different types of sensing capabilities have been studied. Algorithms exist for robots with global positioning and global communication [49], robots that use physical marks to leave a trail [25, 30], robots that use a pre-deployed sensor network [27], and robots that use deployable beacons [16]. This dissertation focuses on robots with directional communication. Payton et.al. have developed an algorithm in which each robot can receive messages in a small radius, and use this to create a virtual pheromone. The gradient algorithm discussed in this dissertation is similar to [13], [53], [23], and [56] in that directional communication is used to transmit relative position information to establish the gradient. Networking researchers would also recognize the gradient algorithm as being very similar to "hop-count" routing [58].

One of the chief difficulties in implementing ant–inspired foraging is implementing the pheromone itself, or some way for the robots to interact. Global communication is generally assumed to not be available, so some other simple means of communication or coordination is required. There have been many approaches to this problem:

- Physical marks. Robots can physically mark their trails in a variety of ways, such as leaving alcohol [46], heat [44], odor [45], visual marks [25], or RFID tags [30].

- Use existing communication channels. In the work of Vaughan et al., robots maintain an internal pheromone model with trails of waypoints, and share it with other robots over a wireless network [47, 48, 49].

- Virtual pheromones. In [13], authors use direct infrared-based communication between robots to transmit a kind of virtual pheromone. They study the use of these signals to create world–embedded computation and world–embedded displays. It is assumed that the robots that receive the pheromone can measure the intensity of the IR reception to estimate their distance from the transmitter. One of the algorithms described in this dissertation extends the virtual pheromone approach.

- Pre-deployed sensor network. The GNATS project [27] uses robots which operate in an environment which is pre-populated with a regular grid of beacons on which information can be stored. They have had success in finding close-to-optimal paths from one place in the environment to another, even in the presence of obstacles.

- Deployable beacons. Some groups have explored the idea of having each robot be able to deploy beacons as it moves through the environment. The beacons can be movable or non-movable [16], and contain pheromone-like information.
- Robot chains. In [52, 5], the robots form chains and attempt to remain in close proximity with each other, through communication or even by physically gripping the next robot in the chain.

There are several shortcomings of these approaches which the present work aims to address. Making permanent physical marks in the environment is generally not acceptable, and temporary or decayable marks are difficult to physically implement. Relying on predeployed sensor networks is highly restrictive and prevents operation in a new or unexplored environment. Deploying beacons is a good method but requires building robots which can carry many beacons and can also intelligently recover previously laid beacons. Instead, the algorithms in this dissertation use the robots themselves as beacons. (see also [26])

Area–Sweeping Foraging In the area–sweeping foraging strategy, inter–robot virtual forces are used to make movement decisions which cause the swarm to adapt a line shape. Virtual forces have been used in the past in other areas of multi–robot systems. Spears studied physics–based control of vehicle swarms, with attractive and repulsive forces forming a lattice of vehicles [55]. In the field of sensor networks, Howard et.al. have used potential fields to achieve dispersion of nodes [21]. The closest work to our application is Balch's notion of social potentials [7]. Social potentials involve robots navigating to a goal while remaining in a formation, feeling virtual forces based on the position of the goal and the relative positions of other robots. These algorithms are focused on maintaining a formation. This dissertation uses a similar concept in which robots feel virtual forces, and shows how to use the formation to search the region, developing a virtual forces–based foraging method.

Algorithm Switching Multiple behaviors within a single algorithm are well known inside the swarm algorithm community [36]. Matarić and Arkin have worked extensively in behavior–based robotics [3, 11]. It is common for individual robots to switch between the behaviors of food collecting, obstacle avoidance, and resting, for example. Parker has studied quorum sensing and distributed consensus in a swarm setting, using it to enable a swarm to move between subtasks in an overall task [37, 38, 39]. In quorum sensing, robots cast votes, and tally each others votes. No robot can tally all the votes, and communication links are always changing, so a distributed threshold system is used to determine if enough robots voted for a particular decision. This system can take a long time to converge to an approximate consensus. Instead of using formal quorum sensing, this dissertation takes a simpler approach based on "triggers" to initiate algorithm switches. McLurkin has developed a large range of robot swarm behaviors[33] as well as dynamic task assignment methods for individual

robots within a swarm [32]. These methods focus on individuals, whereas this dissertation develops a method for the swarm as a whole to switch algorithms. Chapter 4 of this dissertation presents an adaptive foraging method in which the swarm makes colony-level decisions based on distributed information, choosing the algorithm best suited to the given food location.

1.5.3 Hardware Measurements

Robustness and error tolerance are often–claimed aspects of multi–agent swarm algorithms [17], but quantitative hardware–based confirmations are less common. Often, a new algorithm is shown to perform well in the face of the particular hardware imperfections already present in the system [54], [25] or unplanned changes in the environment [22, 59], but it is not common for hardware quality to be independently varied while performance is measured.

In one study which takes some hardware quality dependent measurements, a coverage algorithm is shown to be robust to some degradation in position measurement accuracy and communication range [51]. In this case, position error is modeled as a probability of calculating a completely random position. It has also been shown for search tasks that randomness of target position and robot movement error can increase the attractiveness of random algorithms relative to coordinated ones [41]. This dissertation does not study the benefit of coordinated algorithms, but rather the effect of hardware inaccuracies on the performance of coordinated algorithms. The approach also differs in that I study several primitive behaviors rather than a few complete algorithms, covering a broader space of algorithms.

There is little information published using physical robots to measure the effect of the quality of a robot's sensors and actuators on the performance of common tasks running on those robots. Chapter 5 systematically varies sensor and actuator quality as the independent variable, both in simulation and hardware.

1.6 Thesis Organization

Chapter 2 presents the hardware model and the algorithms, as well as describing the algorithms' behavior. Chapter 3 analyzes and compares the performance of the algorithms. Chapter 4 presents and analyzes the algorithm switching behavior. Chapter 5 describes the hardware quality vs algorithm performance tradeoff and presents results. Finally, Chapter 6 concludes.

Chapter 2

Algorithms

In this chapter, I will describe three distributed swarm foraging algorithms. These algorithms run on a swarm of identical robots, each of which is so simple that it could not effectively accomplish the task alone. The robots must coordinate, working together in an area much larger than the sensing and communication range of an individual robot.

The targeted application is a swarm of robots which must search the world for an object of interest (the 'food') then return the food in pieces to the 'nest'. Because of the focus on simple hardware requirements, these algorithms can not use information like absolute robot position, food or nest location, the size of the swarm, or obstacle locations, because the hardware required to obtain that information is too complex. Instead, these algorithms are designed to work with only a small amount of information.

Coordination is required for the robots to leverage their large numbers and collectively perform a task, and this coordination must be mediated by some type of communication. In this chapter, I choose a simple type of communication which is reasonably easy to implement on robots, and then build algorithms using it. (In Chapter 5, I reduce the communication model even further and measure the impact.)

The simple sensing capability of the robots is the central challenge in developing swarm foraging algorithms. The robots will not have global communication, meaning that one individual can not simply inform the rest of the swarm concerning some information it has. Local communication can be used for short range simple message passing only. The robots also do not know their global position. This means that they can not navigate back to a place they have been in the past, and they can not share the locations of places they have been in the past. The robots do not have odometry, meaning they can not estimate their position by integrating the path they have traveled, and they can not retrace their steps to get back to a location they have visited before. Additionally, the robots have a simple sensor suite, consisting of a single bump sensor and a single target sensor (for detecting the presence of the search target in the immediate vicinity of the robot). This means that navigation

algorithms can not rely on long distance obstacle or target sensing. The algorithms also can not assume other complex pieces of hardware such as a chemical deposition and detection system, which would be tempting to use when imitating ant behavior.

In this chapter, I begin by describing the assumptions and models governing the robots, the environment, and the foraging task, then I describe the algorithms themselves on both a local and global level, and finally I demonstrate the behavior and characteristics of each algorithm.

2.1 Robot, Environment, and Task Models

2.1.1 Robot Model

For the robots, I use a simple model inspired by recent swarm robot hardware, such as the E-Puck [35] (shown in Figure 5.1 on page 66), McLurkin's SwarmBots [31], and Payton's pherobots [13]. I assume a simple non–holonomic circular robot 8cm in diameter that moves and turns in continuous 2D space. Each robot has sensors for nest, food, and obstacles in direct proximity to the robot. The `sweeper` algorithm also requires two of the robots to have compasses. Each robot can communicate with nearby robots in a range of 80cm, and measure the range and bearing from which each transmission came (using, for example the RBZ communication board, an E-Puck extension described in [2] and installed on the E-Puck in Figure 5.1). The messages which can be communicated between robots are two bytes long. Robots do not have global position measurement or global communication. Figure 2.1 diagrams the communication and sensor ranges used in the robot model.

2.1.2 Environment Model

The environment is modeled as a simple 2–dimensional square region with boundaries at the edges. The environment model uses continuous space, as opposed to discrete 'grid squares' to more accurately model the physical space that real robots operate in. The world can contain a nest, a food pile, robots, and obstacles. Robots can not occupy the same physical space as another robot or an obstacle. Figure 2.1 illustrates the possible objects in the world.

All robots begin clustered around the nest. When a robot is very close to a food pile, it can pick up a unit of food. The food pile can contain any amount of food, and the amount decreases as robots pick it up. Each robot can carry a maximum of one food unit. When it is very close to the nest, a robot can drop its food unit at the nest.

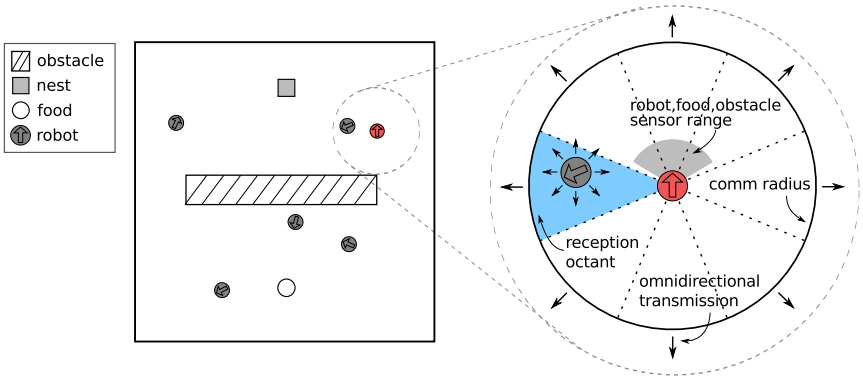


Figure 2.1: This figure shows the communication and sensing structure of the robots, approximately to scale. The figure in the left shows an example world setup. The blowup details a hypothetical communication situation. When the robot on the left transmits, the robot in the center can receive the transmission, and also knows that the transmission came from the left–facing octant. The light gray range in front of the central robot indicates the approximate area in which the nest, food, and obstacle sensors are sensitive. These diagrams are approximately to scale.

2.1.3 Task Model

For the foraging task, I assume a world with a nest in the middle and one unlimited food source placed at a prescribed distance from the nest. The robots must find the food, then return food units to the nest one at a time. Because the robots have no direct position measurement system, they must coordinate in order to maintain and share information about their own position and the food position.

2.2 Description of Algorithms

This section will describe three algorithms, called the virtual pheromone (`VP`) algorithm, the `gradient` algorithm, and the `sweeper` algorithm. Overall, given a specific number of robots, the swarm must cover as much area of the world as it can in search of food. Because of their lack of global localization and communication, the robots must stay in communication with each other while they explore. One strategy is for the robots to expand as much as they can as a cluster away from the nest while still remaining connected. This is the strategy used by the `VP` and `gradient` algorithms. A second strategy is for the robots to form a line reaching away from the nest and sweep this line around the world. This is the strategy used by the `sweeper` algorithm.

These algorithms draw inspiration from the foraging behavior of ants, so a brief description of this behavior will precede the algorithms themselves.

2.2.1 Ant Foraging in Nature

Some species of termites and ants, such as *Lasius niger* and *Linepithema humile* (commonly known as the Argentine ant) exhibit group behaviors far beyond what one would expect from a simple collection of individuals. These ants are capable of searching a very large area and finding and returning food without a group leader. Ants do not have global communication in the typical sense, but are nonetheless capable of passing information to each other using a communication strategy called stigmergy. Stigmergy differs from traditional communication in three important ways:

- The environment itself is the communication medium. For example, when a termite modifies the environment by leaving a bit of dirt in a specific place, that can indicate that that location is a good place to build, and other termites can detect the presence of that dirt and place another piece to continue the structure. The individuals communicate by modifying the environment.

- Messages have no specific recipient. It is not necessary for every individual in the swarm to receive the message, but it is also not addressed to a specific individual. Continuing the termite example, although any termite could have received the information, only the termites that happen to find the piece of dirt actually see it. The communication is received by individuals for whom it is useful.

- Finally, the messages are localized in space. The information can only be received by individuals who are in the close vicinity of where the message was created. This adds to the information content of the message. It's not just a simple message, but a simple message at a specific place. For the termites, an individual does not have to communicate "someone should put a piece of dirt next to the other piece of dirt at the north wall of the tunnel after the third branching intersection from the fourth tunnel out of the main chamber." Instead, they just have to communicate "someone should put a piece of dirt here." Instead of having to be ignored by 99% of the individuals, it is only received by the few individuals to whom it matters, i.e. those in the vicinity of the message who would be most affected by it and could most capably act on it.

Ants use stigmergy to tell each other where the food is and how to get there. They leave small pheromone marks in the environment to give information to other ants who detect those marks [42]. Each ant can both deposit and detect this chemical, and each ant uses the distribution of pheromone in its immediate vicinity to decide where to move. Ants explore the environment and when an ant is carrying food, it lays pheromone as it returns to the nest. Over time, a trail develops between the nest and the food, and other ants follow this trail to locate the food and return it to the nest [4, 43].

This distributed leaderless pheromone algorithm is a starting point, but several important adaptations must be made. The first change has to do with the way the robots return to the nest. The pheromone trail is laid by individuals once they leave the food, but the individual still needs a way to return to the nest. Ants have various methods to do this, including visual landmarks and odometry (remembering how far and in what direction the nest is) [4, 57]. Because of the focus on miniaturizability and hardware simplicity, neither of these approaches is attractive – vision requires significant computation and memory to interpret the image, and odometry requires high accuracy encoders. As an alternative method to navigate back to the nest, two distinguishable pheromones are used instead of one. One pheromone leads to the food and a second leads to the nest.

A second change from the biological algorithm is that *virtual pheromones* are substituted for real ones. Implementing real chemical pheromones on robots would require fabrication of an accurate chemical deposition and sensing system, which would make the robots much too complicated, large, and expensive. Given the focus on simple, small, and inexpensive hardware, using real chemicals is not an option. However, using the simple local direct communication between robots, virtual pheromone information can be transmitted.

2.2.2 Virtual Pheromone Algorithm

The `VP` and `gradient` algorithms are based on the idea of creating a field of beacons (which are robots standing still), and using the beacons as places where information can be stored and read by the walkers (the remaining robots which haven't decided to become beacons). The field of beacons will expand roughly circularly around the nest, in search of the food. In the `VP` algorithm, the information on the beacons is used as an abstraction of ant pheromones..

In the `VP` algorithm, the beacons act as locations on which virtual pheromone can be stored. The virtual pheromone is simply a floating point number. Other wandering walker robots can read the pheromone level by receiving a transmission from the beacon, and they can lay virtual pheromone by transmitting to the beacon. Given a network of beacons, the walker robots could use the distribution of virtual pheromone which they can sense to decide how to move. Virtual pheromones decay at a specific exponential rate (which is why they must be floating point numbers) and robots follow the pheromone distribution by moving toward the strongest value. A diagram of an example situation of the `VP` algorithm is shown in Figure 2.2a.

Local Description

Beacon: Robots acting as beacons are stationary and broadcast two floating point numbers, called `nestPheromone` and `foodPheromone`. Walker robots "lay" pheromone on a beacon by transmitting a special message, which will cause the beacon to in-

crease its `nestPheromone` or `foodPheromone`, depending on the message. Beacons slowly decrease their `nestPheromone` and `foodPheromone` over time, to mimic the pheromone decay in nature. They decrease their pheromone values by 0.5% per second, meaning that after 5 minutes a deposited pheromone will have lost about 80% of its strength. This 0.5% decay rate was chosen empirically, and yields a balance between the pheromone decaying too fast causing the information to disappear, and the pheromone decaying too slowly causing the information to be stale. The algorithm is not particularly sensitive to the exact decay rate.

Walker: Walker robots always attempt to navigate either to the food or the nest, depending on whether they have food. In either case, a walker measures the bearing to the maximum pheromone value (`nestPheromone` or `foodPheromone`, depending on its destination), turns toward that beacon, and moves in that direction. The beacon ranges overlap, so that for example, as a walker is moving toward a beacon with a `nestPheromone` of 40.0, it would come into the range of a 50.0 before it reached the 40.0. If a walker has no information about where it should go (it can only hear 0.0), it does a random walk.

Walkers lay `nestPheromone` on all beacons in their range when they are not carrying food, and `foodPheromone` when they are carrying food. In each case, they lay one unit of pheromone per time step. This number is arbitrary and its magnitude does not meaningfully affect the algorithm.

Beacon to Walker transition: If a beacon robot can detect more than 4 other beacons, it will become a walker robot with a 20% chance. This probabilistic effect is required to prevent several beacons, all of whom can collectively hear each other, from becoming walkers at exactly the same time and leaving a hole in the beacon field. (The specific numerical value of the probability, 20% in this case, is not critical. A wide range of numbers work and do not affect the performance.)

Walker to beacon transition: A walker robot will decide to become a beacon if it can only detect 1 or 2 other beacons.

Global Behavior

All robots start as walkers clustered around the nest. Some of them will decide to become beacons immediately because initially there are no beacons. The remaining walkers will begin searching for the food, by random walking at first. As they wander away from the nest, some will decide to become beacons, and the beacon field will expand away from the nest. The walker robots will lay `nestPheromone` on the beacons as they expand. Eventually one of the robots could stumble across the food, pick it up, and begin laying `foodPheromone` as it returns to the nest following the `nestPheromone`. At this point, other walkers can use the newly laid `foodPheromone` to find the food.

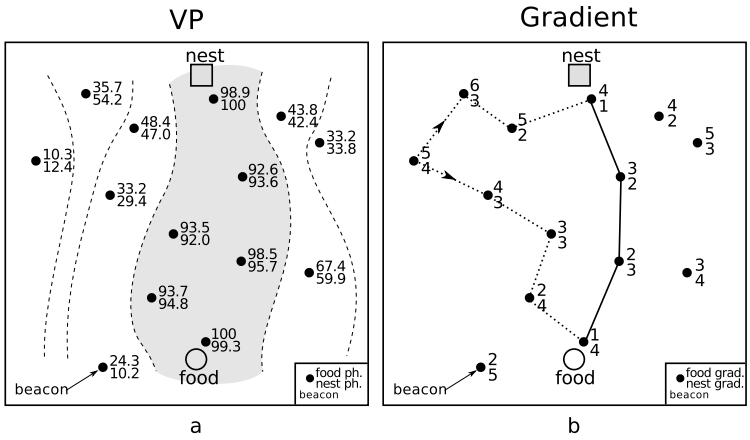


Figure 2.2: This diagram shows example situations in both the `VP` and `gradient` algorithms. Each black dot represents a beacon (walkers are not shown). The numbers to the right of each beacon indicate the virtual pheromone levels (figure a) or gradient values (figure b) – food pheromone/gradient on the top and nest pheromone/gradient on the bottom. In figure a, the dotted lines indicate approximately level curves in both pheromone values. As can be seen, the area between the nest and food has high concentration of pheromone. In figure b, the solid line indicates the shortest path between the nest and the food. As an example, the dotted lines indicate paths that a robot near the $\frac{5}{4}$ beacon on the left could take to get either to the nest or to the food.

A diagram of the `VP` algorithm is shown in Figure 2.2a. For clarity, only beacon robots are shown. Both virtual pheromone values are strongest along a path connecting the food and the nest, and they decrease further away. By following this trail, the walker robots can walk back and forth between the nest and the food.

2.2.3 Gradient Algorithm

The `gradient` algorithm is similar to the `VP` algorithm in that robots can decide to act as either beacons or walkers – beacons transmit values, and walkers use those values to decide where to move. The main difference is that instead of the values that the beacons store and transmit being real–valued floating–point numbers, they are integers. The first beacon standing next to the nest transmits a 1, then the next beacon slightly further out would be able to hear the 1 and so would transmit a 2. In general, each beacon transmits the minimum of all the other beacons it can hear, plus one. In this way, the transmission value of each beacon can be interpreted as the number of beacons between that beacon and the nest, in other words, a rough gradient. Furthermore, a walker robot can use these gradient values to find a path to the nest by always moving to the lowest gradient value it can hear. In a similar manner to the `VP` algorithm, each beacon actually transmits two values – one indicating how

many beacons away from the nest it is, and the other indicating the number of beacons away from the food.

Local Description

Beacon: Robots acting as beacons are stationary and broadcast two numbers, called `nestGradient` and `foodGradient`. They listen for all other beacons in their communication range and record the `nestGradient` and `foodGradient` of each one. Beacons find the minimum of all `nestGradient` values they have received, increment that by one, and take that as their own `nestGradient`. An analogous procedure is used to calculate `foodGradient`. These new values are then broadcast by the beacon. Any beacon directly next to the nest/food broadcasts a 1 for its `nestGradient`/`foodGradient`.

If a beacon has no information about its distance from the food (as happens early in the run, before the food has been found), it broadcasts 0. The value of 0 is treated specially—when a beacon hears a 0, it does not include it in its normal 'minimum plus one' calculation.

Walker: Walker robots always attempt to navigate either to the food or the nest, depending on whether they have food. In either case, a walker measures the bearing to the minimum gradient value toward the target of interest, and moves in that direction. The beacon ranges overlap, so that for example, as a walker is moving toward a beacon with a `nestGradient` of 4, it would come into the range of a 3 before it reached the 4. If a walker has no information about where it should go (it can only hear 0), it does a random walk.

Beacon to Walker transition: If a beacon robot can detect more than 4 other beacons, it will become a walker robot with a 20% chance. This probabilistic effect is required to prevent several beacons, all of whom can collectively hear each other, from becoming walkers at exactly the same time and leaving a hole in the beacon field.

Walker to beacon transition: A walker robot will decide to become a beacon if it can only detect 1 or 2 other beacons.

Global Behavior

All robots start as walkers clustered around the nest. Some of them will decide to become beacons immediately because initially there are no beacons. The remaining walkers will begin searching for the food. There will be no `foodGradient` (it will all be 0), so they will random walk. As they wander away from the nest, some will decide to become beacons, and the beacon field will expand away from the nest. Eventually one of the robots could stumble across the food, and would then begin transmitting 1 for its `foodGradient`, causing the food gradient to form. At this point, any walker

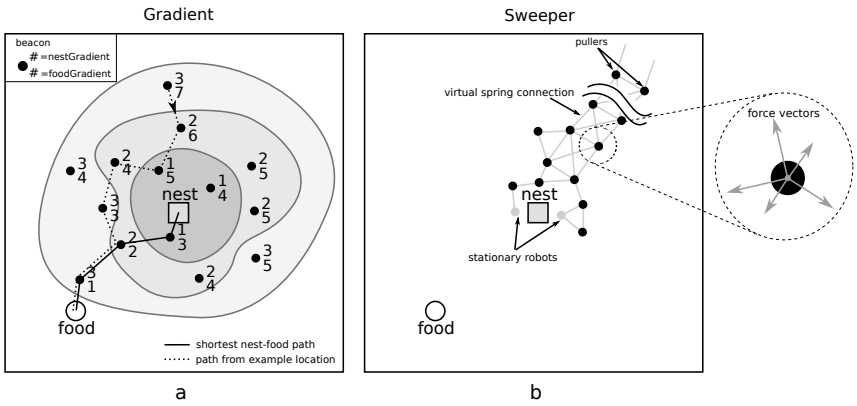


Figure 2.3: Example situations in the `gradient` algorithm (a) and the `sweeper` algorithm (b). Walkers in (a) are not shown to reduce clutter. The gray contours roughly depict the gradient to the nest.

can listen to the beacons near it and know how far it is from the food and how to get there. All the walkers immediately start moving directly toward the food. As walkers pick up food, they use the gradient field to bring it to the nest.

Figure 2.3a illustrates an example snapshot of the gradient algorithm. The beacon standing directly next to the nest is transmitting a 1 for its nest gradient, and the beacon standing next to the food is transmitting a 1 for its food gradient. All other beacons are updating their gradient values accordingly. Now, by always moving down the appropriate gradient, the walker robots can walk to the nest or to the food from wherever they are.

One significant difference to notice between the `VP` and `gradient` algorithms is that `VP` requires the walkers to transmit information to the beacons every time they want to lay a virtual pheromone. The `gradient` algorithm, however, only requires beacons to transmit – the walkers only need to receive.

2.2.4 Sweeper Algorithm

A different strategy for search or foraging involves individuals forming a "search front" and systematically sweeping an area to find an object. Here I describe an algorithm that uses virtual forces to form a line of robots extending from the nest that sweeps the world like the hand of a clock. When the line finds food, some of the robots remain as beacons while others act as walkers to return the food.

Fundamentally, this strategy creates a 1D structure of robots (roughly a line), as opposed to the gradient strategy which creates a 2D structure of robots (roughly a circle). The 1D structure is expected to be able to sweep a larger area than could be 'filled in' with the same number of robots.

Local Description

Normal: In the `sweeper` algorithm, all robots are always transmitting. Each robot measures the range and bearing to all the other robots in its communication range. Based on the position of each other robot, it calculates a virtual force on itself. For example, the robot could place a simple virtual spring between itself and each other robot. In this case, for each robot detected at relative position $\overrightarrow{r}$, the virtual force $\overrightarrow{F}(\overrightarrow{r})$ would be

$$\overrightarrow{F}(\overrightarrow{r}) = a\frac{\overrightarrow{r}}{r_c} - b\hat{r}$$

where r_c is the communication range of the robot and a and b are empirically chosen constants, with a controlling the strength of the spring and b controlling the resting length. The choice of $\overrightarrow{F}(\overrightarrow{r})$ is critical to the proper functioning of the algorithm. Various spring-like functions are conceivable, and this choice will be explored further in section 2.3.2. However $\overrightarrow{F}(\overrightarrow{r})$ is calculated, the robot sums the virtual force from each other robot in its communication range, and moves in that direction by an amount proportional to the magnitude of the force. There is one special case: two robots directly next to the nest never move regardless of the virtual forces on them.

While the robots are calculating forces and moving, they are simultaneously using communication to establish a gradient field similar to the one described in the gradient algorithm. This does not require extra communication. The data content of the signal encodes the two gradient values, and range and bearing to the transmitter are used to calculate the virtual forces.

The robot treats the `nestGradient` exactly as before, updating it using the min+1 algorithm. `foodGradient` is treated slightly differently. Any time a robot sees a non-zero `foodGradient`, it temporarily stops executing the `sweeper` algorithm and switches to `gradient`. If the `foodGradient` returns to zero, the robot returns to executing the `sweeper` algorithm.

Puller: Two robots are pre-determined to be "puller" robots. This choice could be made in a distributed fashion using a distributed leader election algorithm, for example [14]. The pullers participate in the virtual forces system described above, but they also feel one additional force. These two robots must use their compasses to measure the relative bearing to north (the unit vector $\hat{N}$), then put a virtual 'clock' force on themselves equal to

$$\overrightarrow{F_c} = c\hat{N}R\left(\frac{2\pi t}{T}\right)$$

where R is simply the 2D rotation matrix,

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$\overrightarrow{F_c}$ is a force which simply rotates around like the hand of a clock as time t increases. The parameter c is an empirically chosen magnitude and T sets the period of the

rotation. T is determined by the puller, and can be changed at any time based on the puller's `nestGradient` value.

Global Behavior

When the `sweeper` algorithm begins, all robots calculate forces and begin moving appropriately. Initially, repulsive forces cause the swarm to expand into a tight clump around the nest. The pullers will be forced to the edge of the pack and a line of robots will form extending from the nest to the pullers. This line of robots will rotate as the pullers pull it around, sweeping around the world like the hand of a clock. When the line encounters food, it stops moving and the swarm returns the food using the `gradient` algorithm, with walkers moving along the line of robots already established. When the food source is exhausted (as detected by a loss of food gradient), the forces resume and the line keeps sweeping.

One can imagine that longer lines (with more robots) would need to rotate slower than shorter lines. Hard-coding the period T would require knowing the number of robots in the swarm, which is not a scalable solution. Instead, the pullers set T based on their `nestGradient`. A higher value for `nestGradient` indicates a long line, so the puller will choose a larger T. This allows the rotation rate of the line of robots to slow down as the puller 'pulls' the initial crowd of robots out into a line.

Figure 2.3b illustrates an example snapshot of the `sweeper` algorithm.

2.3 Behavior of Algorithms

In this section, I will demonstrate the behavior and discuss several characteristics of each algorithm. A full analysis comparing the performance of each algorithm is given in Chapter 3.

2.3.1 VP and Gradient

Overall Behavior

The `VP` and `gradient` algorithms have similar behavior. Figures 2.4 and 2.5 show screenshots of the `VP` and `gradient` algorithms during the course of a run. In each case, the swarm expands from the nest and forms the gradient to the nest using either virtual pheromone values or gradient values. Once the food is found, `gradient` is able to spread that information nearly instantly through the beacon network. `VP` must rely on other walkers to spread food pheromone. `gradient` can begin returning food soon after it is found. `VP` must establish a trail of both pheromones for the walkers to follow.

Figure 2.6 shows plots of food returned vs time for sample runs of the `gradient` and `VP` algorithms. At each time step during each run, the total food that had been

returned to the nest was measured, along with the fraction of non–beacon robots that were carrying food, and the total number of beacons. These examples were run in a 4m×4m world in configuration A (see Figure 3.3) with 70 robots. In the first 50-100 time steps, the number of beacons rapidly increases to a steady value as the ad-hoc network of beacons is deployed. Next, the fraction of food–carrying robots becomes non–zero, as the first robots begin finding and picking up the food. Some time after that (about 100-200 time steps in these examples), the first robots are able to return food to the nest. For the rest of the run, the amount of food returned increases and the number of walker robots carrying food approaches roughly 50%. It makes sense that half of the walkers would be carrying food in steady state, because approximately half would be on their way to the food and half would be returning.

The major difference between the two algorithms is that `gradient` finds food faster than `VP`, and returns more. (This will be shown in more detail in Chapter 3.) `VP` requires time for the trail between the food and the nest to be built and reinforced after it is found, whereas no such time is required in `gradient`. In `gradient`, once any robot has found the food, all robots have a path to get there, and then once they pick up food, they have a path to get back to the nest. No reinforcement of the trail is required.

Coverage

Both the `gradient` and `VP` algorithms rely on a beacon field to cover the world and hopefully find the food. The amount of the world that this beacon field can cover determines how much of the world an algorithm can search and ultimately affects its overall usefulness.

Because of the lack of central planning, the `gradient` and `VP` algorithms are not designed to attempt a beacon placement which would yield optimal area coverage. The low–level rules produce some beacon arrangement, and here I compare the actual area covered by the beacons to the theoretical maximum area that could be covered by an optimal beacon arrangement.

What is the beacon arrangement which yields optimal area coverage while still producing a beacon field which provides useful in information to the walkers? It is not enough to simply spread beacons throughout the world if they can not talk to each other and develop a gradient. So, each beacon must be able to communicate with its "neighbors" and must have a path of communication hops leading back to the nest. For the swarm to spread out as much as possible, each beacon should have as few neighbors as possible while staying connected ultimately to the nest. It is not possible for each beacon to have just one neighbor because then there would only be two beacons in the world which would be each others neighbors. It is also not reasonably possible for each beacon to have two neighbors because this would simply create a line of beacons. It is possible for each beacon to have 3 neighbors. Figure 2.7 shows beacon fields in which each beacon has 3, 4, or 6 neighbors.

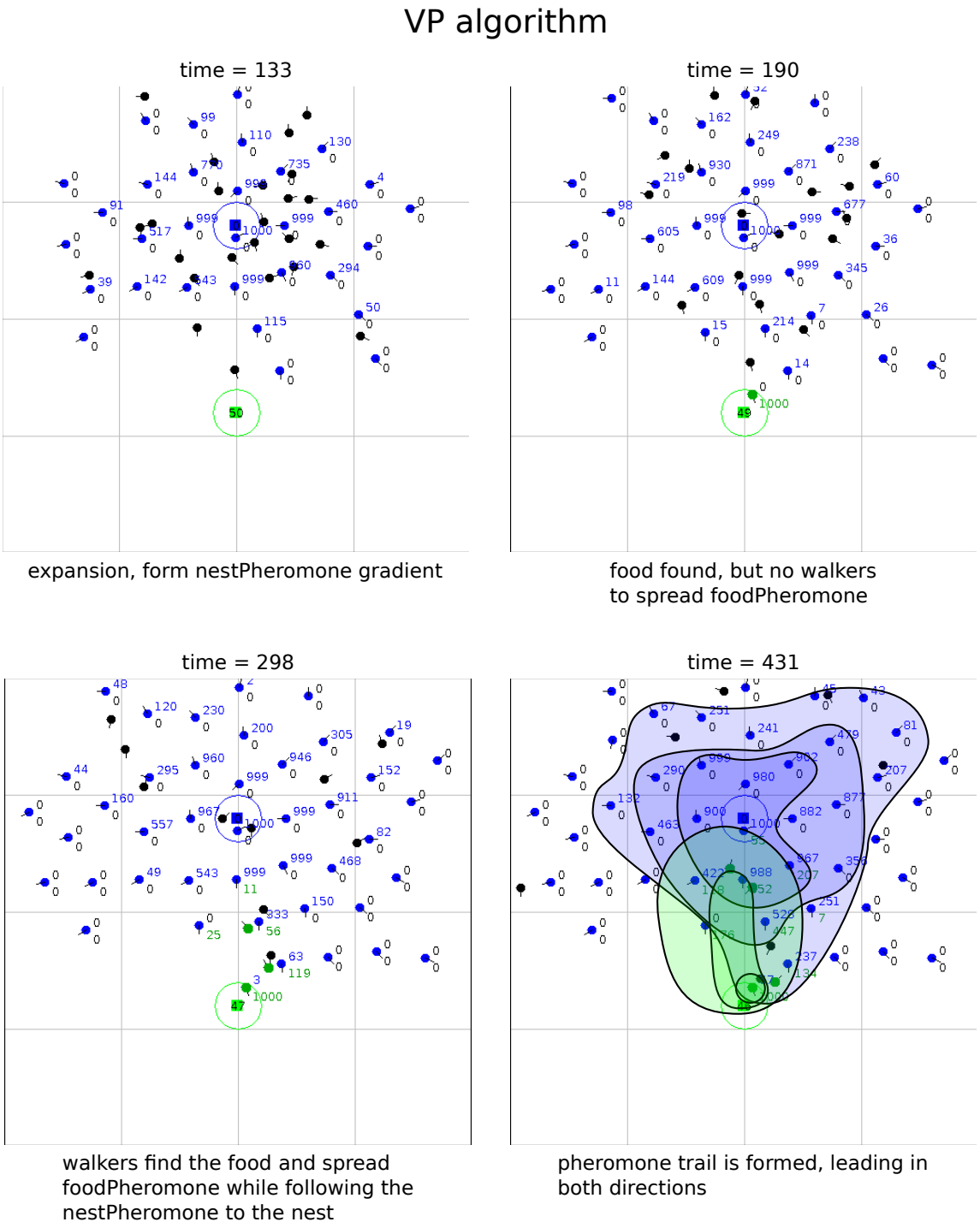


Figure 2.4: This shows screenshots from an example run of the `VP` algorithm in the world configuration of Figure 3.3A (no obstacles). For clarity, contours indicating the intensities of the virtual pheromones are shown on the lower-right image.

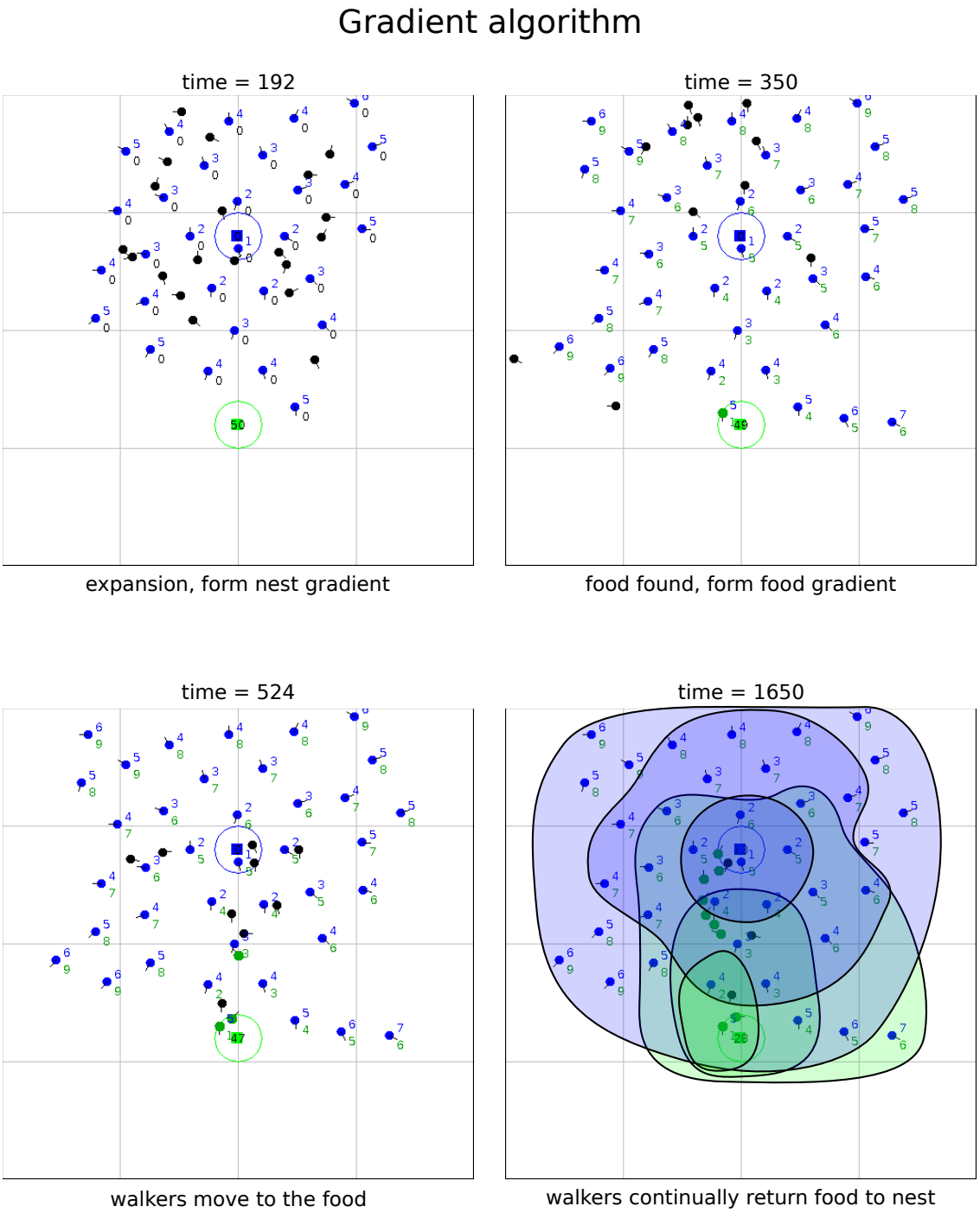


Figure 2.5: This shows screenshots from an example run of the `gradient` algorithm in the world configuration of Figure 3.3A (no obstacles). For clarity, contours indicating the gradient values are shown on the lower-right image.

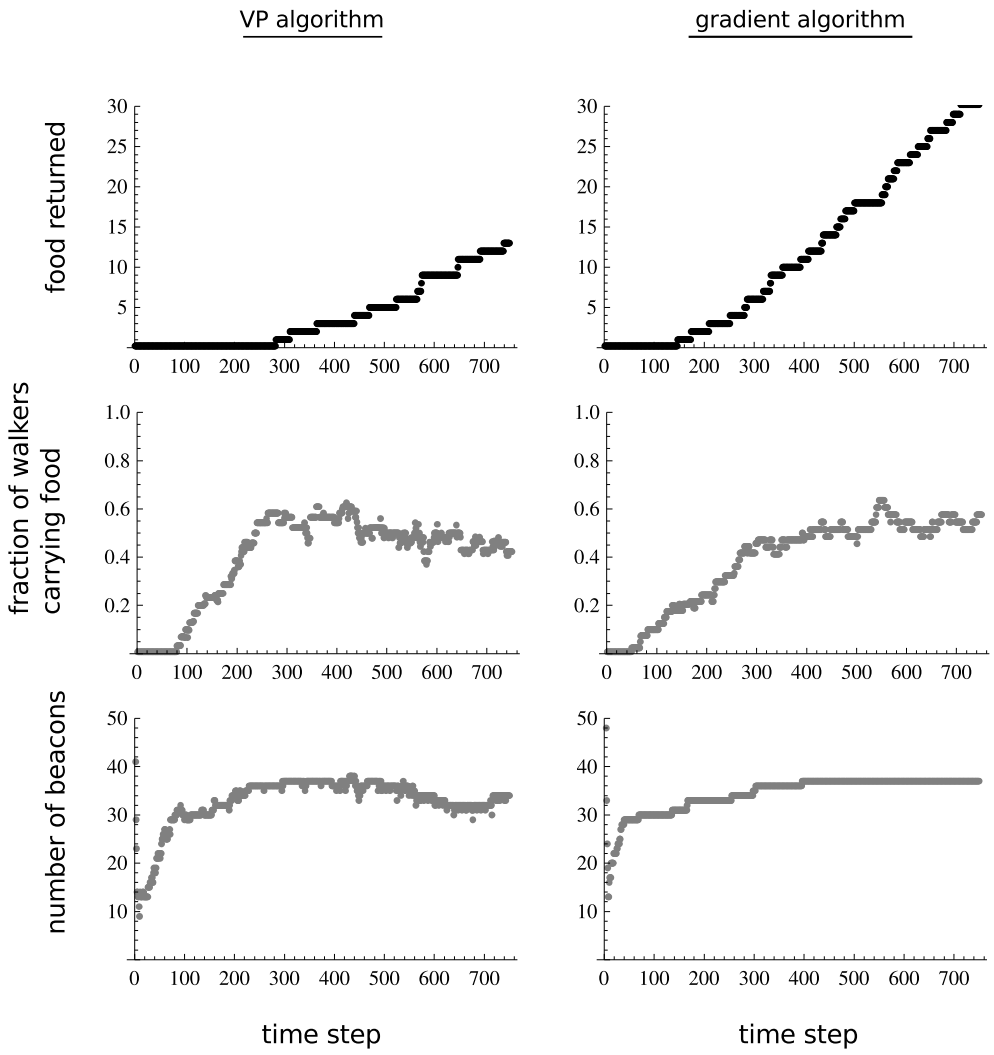


Figure 2.6: This shows an example run of the `VP` and `gradient` algorithms in the world configuration of Figure 3.3A. Each run was performed with 70 robots.

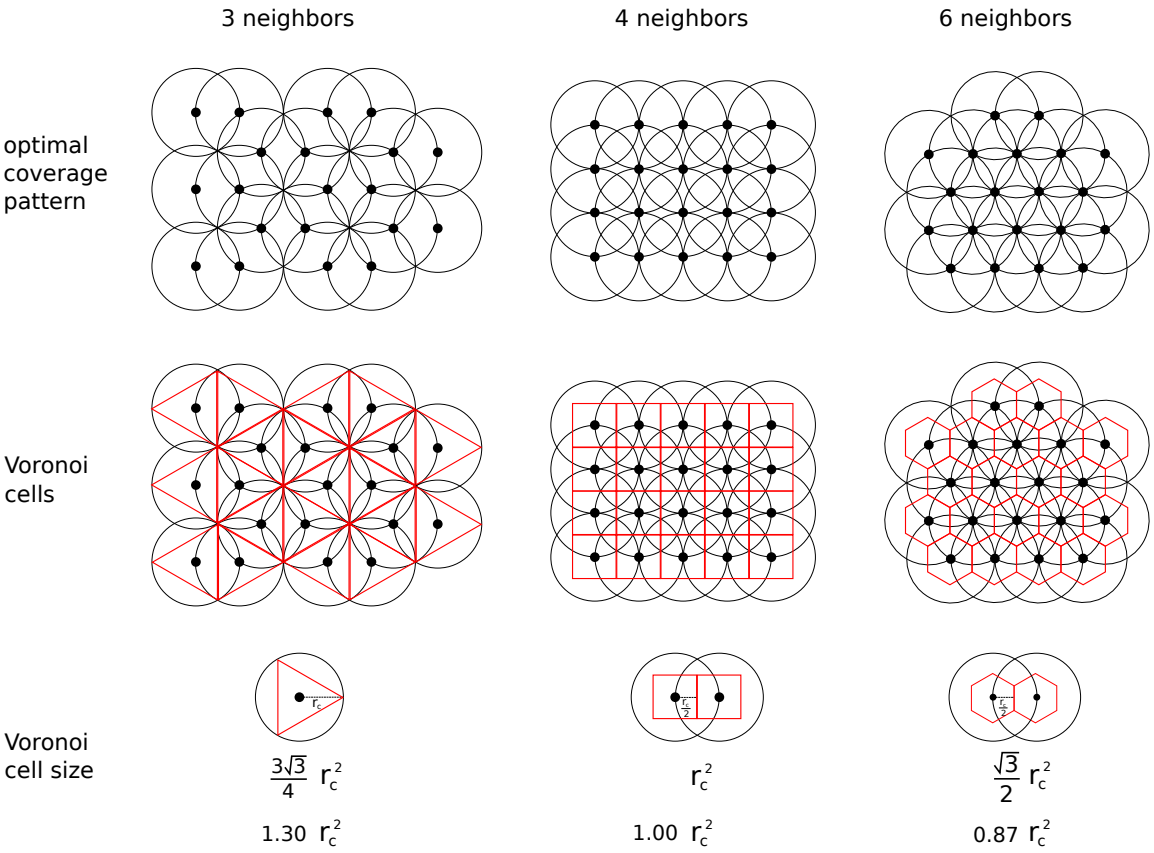


Figure 2.7: The optimal coverage pattern is shown here, in which each beacon has 3 neighbors. Other coverage patterns (4 and 6 neighbors) are shown as well. Each group consists of 20 robots. The size of the Voronoi cell is an estimate of the total area that can be covered by each beacon.

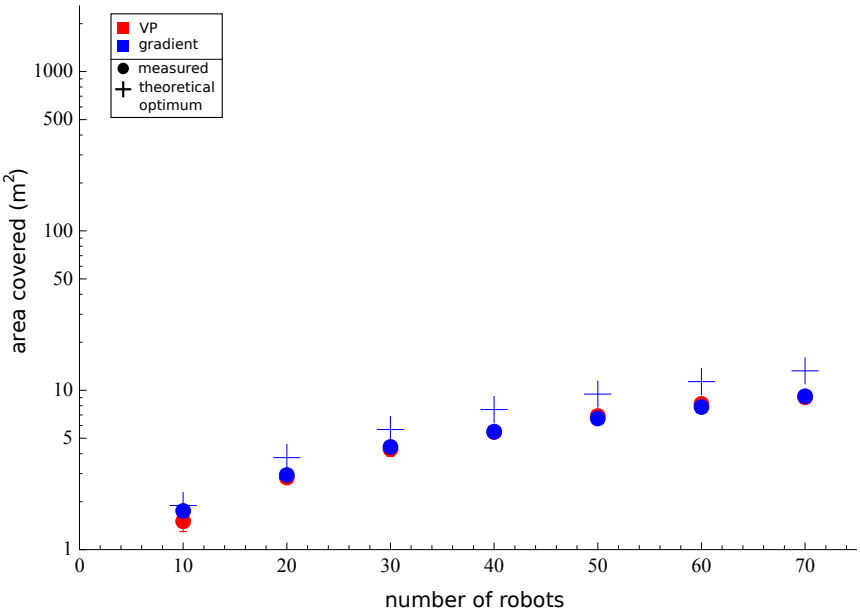


Figure 2.8: This shows the total area of the world which was explored by the `VP` and `gradient` algorithms, as well as their theoretical optimal coverages. There are standard-deviation error bars on all the measured points, but for most of them, the error bars are smaller than the points themselves.

In each case, the beacons are arranged into a regular pattern which optimally covers the area. Then, the Voronoi cell of each robot is drawn. The Voronoi cell is used as an estimate of the amount of area covered by each beacon, such that the total area covered by the swarm is approximately the number of robots N_r times the area of the Voronoi cell. I will denote the area of a Voronoi cell as V_n in a coverage pattern in which each beacon has n neighbors. In the case of 3 neighbors, each Voronoi cell is an equilateral triangle with area $V_3 = \frac{3\sqrt{3}}{4} r_c^2$, where r_c is the communication radius. For 4 neighbors, the Voronoi cell is simply a square with area $V_4 = r_c^2$, and for the case of 6 neighbors, the Voronoi cell is a hexagon with area $V_6 = \frac{\sqrt{3}}{2} r_c^2$. Numerically, $V_3 > V_4 > V_6$, so as expected, having more neighbors leads to smaller Voronoi cells and tighter packing. The optimal arrangement is the 3 neighbor configuration.

The theoretically optimal area that could be covered by a swarm of N_r robots can be approximated as $N_r V_3$. This approximation is an underestimate because it does not count the additional area covered by the robots on the periphery of the group. So the theoretically maximal coverage area is *at least* $N_r V_3$. Because the `VP` and `gradient` algorithms use beacon coverage in the same manner, this theoretical maximum is valid for both.

To measure how close the actual algorithms come to the theoretical optimum, simulations were run in a large world with no food, and the actual area covered by the swarm when it had finished expanding was measured. The results are shown in

Figure 2.8. The algorithms achieve performance very close to the theoretical maximum, and follow the same trend. These trends are expected to hold regardless of r_c. Of course, this maximum is a lower bound, so the true maximum is slightly higher, but the actual algorithms are close nonetheless. Without central planning and global sensing, the algorithms can still achieve near–optimal coverage.

2.3.2 Sweeper

Overall Behavior

The `sweeper` algorithm is intended to cause the swarm to form a line of robots originating at the nest and sweep the world in search of food. When food is found, the robots will establish a gradient to the food and nest using a beacon network consisting of the robots themselves, and a few robots will become walkers and return the food to the nest. This process is shown through several screenshots in Figure 2.9. In these screenshots, the lines drawn on each robot indicate the virtual forces that it puts on itself. At the beginning, all the robots are clustered together so they all feel strong repulsive forces (`t=1`) which cause them to expand (`t=421`). Slowly, a line is formed (`t=1381`), which sweeps the world (`t=2261` and `t=3900`). When it finds food (`t=4821`), the swarm returns the food to the nest (`t=4821` and `t=5701`).

The length of the line and the rate at which it sweeps the world depend on the number of robots. More robots yield longer lines and slower sweeping. To demonstrate this, the `sweeper` algorithm was run in a world with no food, and the position of the robot furthest from the nest was recorded. When measured in polar coordinates using the nest as a reference, this should show a constant distance and steadily increasing angle for the furthest robot. This is the case, as shown in Figure 2.10. Each pair of plots shows distance and angle of the furthest robot, for varying swarm sizes.

The steady–state distance of the furthest robot indicates the length of the line of robots. As can be seen from the top row of plots in Figure 2.10, this increases as the number of robots increases. Also, the rate at which the angle to the furthest robot increases indicates the rate at which the line is sweeping the world. As can be seen from the bottom row of plots, this rate slows as the swarm size increases.

Force Function

As discussed in section 2.2.4, the virtual force function $\vec{F}(\vec{r})$ is a centrally important part of the `sweeper` algorithm. Intuitively, a virtual spring of the form

$$\vec{F}(\vec{r}) = a\frac{\vec{r}}{r_c} - b\hat{r}$$

seems reasonable. When robots get too close, they should be forced apart and when they get too far apart, they should be forced together. When such a force function is

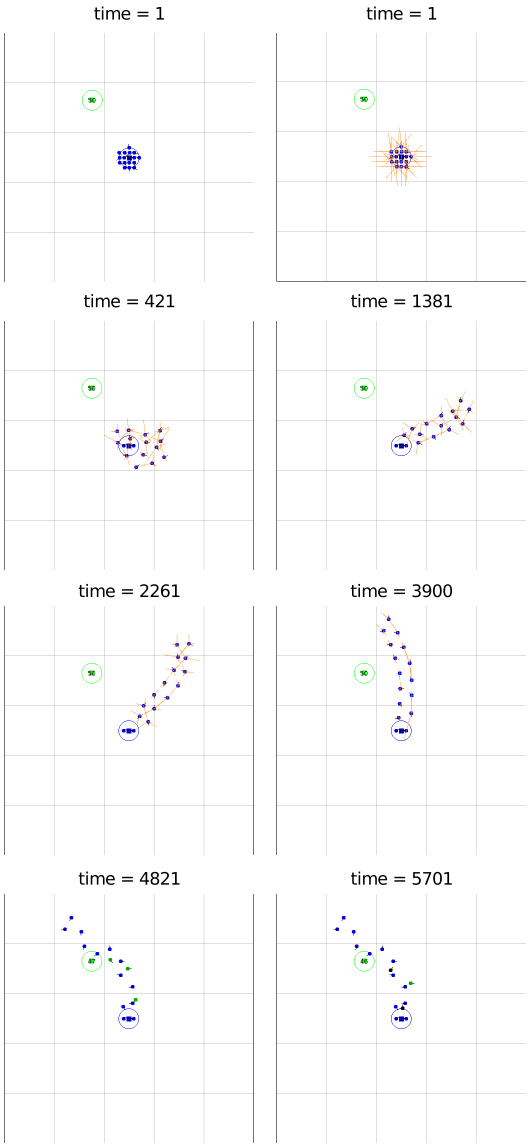


Figure 2.9: This figure shows screenshots from the simulator at different points during the execution of the `sweeper` algorithm. The first image, in the upper left, shows the starting configuration. As time (measured in simulator time steps) moves on, the swarm forms a line which then sweeps the world in search of food, ultimately returning the food to the nest. Lines drawn on the robots indicate virtual force vectors.

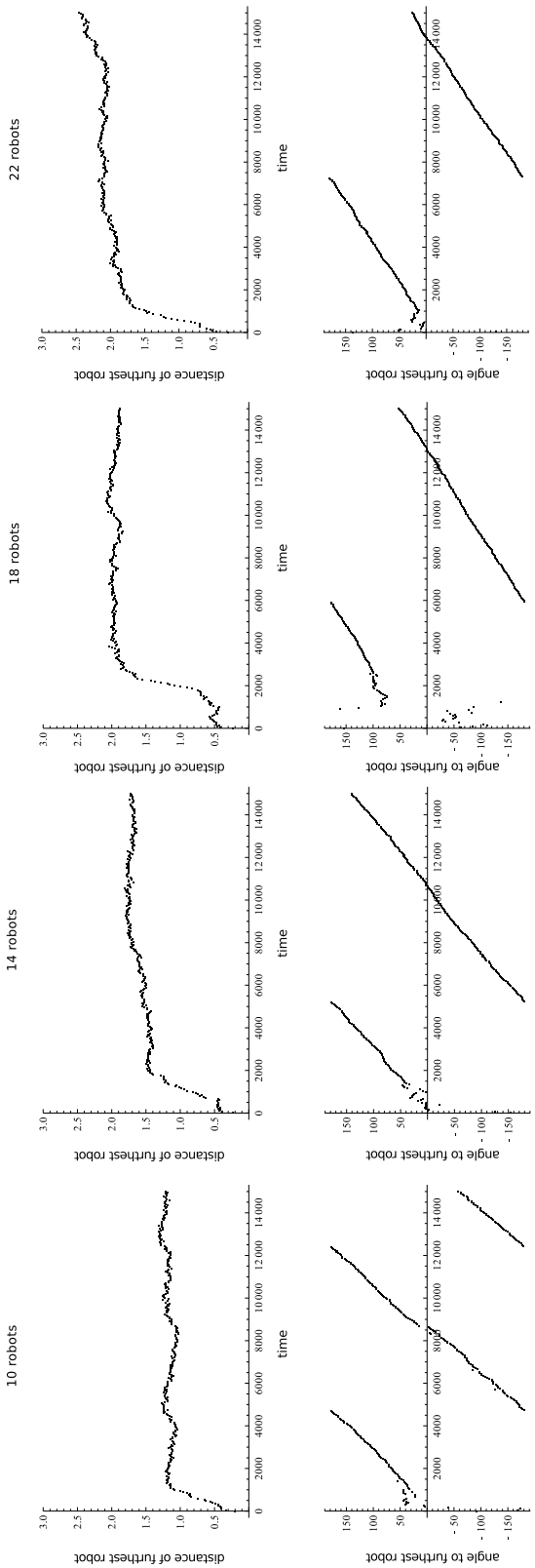


Figure 2.10: Each pair of plots shows the distance and angle of the furthest robot relative to the nest. The swarm forms a line which quickly reaches a steady length, and sweeps smoothly around the world.

used, the resting length of the spring determines how close together the robots will try to be. The rightmost example in Figure 2.11 shows a force function for a simple spring as well as the result for the swarm. In this case, the resting length is too small and the robots just clump together.

However, increasing the resting length of the spring will not solve the clumping problem. The next example in Figure 2.11 shows what happens when the resting length of the spring is simply extended. The robots get too far apart. When two robots get further apart than their communication radius, they are not connected by virtual forces at all, and their spring is 'broken.' When the resting length is increased, the pullers break away from the pack and the algorithm fails. Anecdotal evidence showed that no combination of the parameters a and b would yield satisfactory line formation behavior.

A solution to this problem is to keep an extended resting length, but add an exponential extension component to the spring of the form

$$\overrightarrow{F}(\overrightarrow{r}) = a\frac{\overrightarrow{r}}{r_c} - b\hat{r} + ce^r\hat{r}$$

so that when it nears its maximum length (the communication radius), the force gets larger. Such a force function is shown in the third example in Figure 2.11. This force function works. The pullers are able to exert enough force on their neighbors to pull the swarm into a line, but the resting length is large enough to prevent the entire swarm from clumping. Parameters a, b, and c were chosen empirically and tuned to yield successful line formation. As discussed later, the fact that this tuning is required could be a problem when adapting the `sweeper` algorithm for other robots.

One could imagine adding exponential components to each side of the spring, as in the final example, with a force function such as

$$\overrightarrow{F}(\overrightarrow{r}) = a\frac{\overrightarrow{r}}{r_c} - b\hat{r} + ce^r\hat{r} - de^{-r}\hat{r}$$

In this case, the robots are strongly repelled when they get too close to each other. This force function also causes the line to break, however. Sometimes the robots need to be close to each other in tight portions of the line, but this is forbidden under the fourth case which causes them to move too far apart and break the line.

Coverage

The `sweeper` algorithm covers the world in a fundamentally different manner from the `VP` and `gradient` algorithms. Instead of simply attempting to cover as much area as possible with beacons in a static manner, the `sweeper` method forms a longer 1D structure and covers the world dynamically. This yields much larger area coverage capabilities, as shown in Figure 2.12 (a copy of Figure 2.8 with the `sweeper` data added). However, the `sweeper` algorithm is not stable for large numbers of robots.

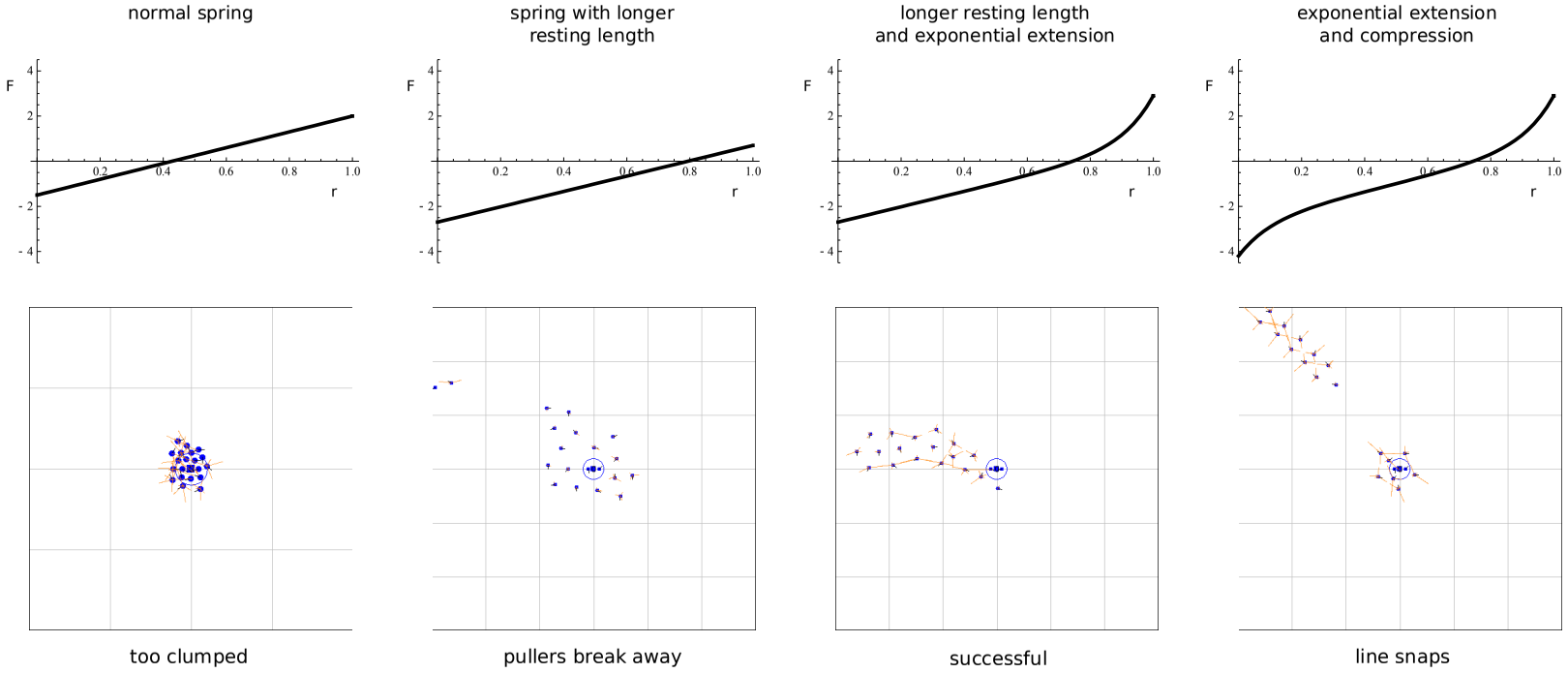


Figure 2.11: Various types of virtual spring force profiles are shown, along with the resultant behavior of the `sweeper` algorithm.

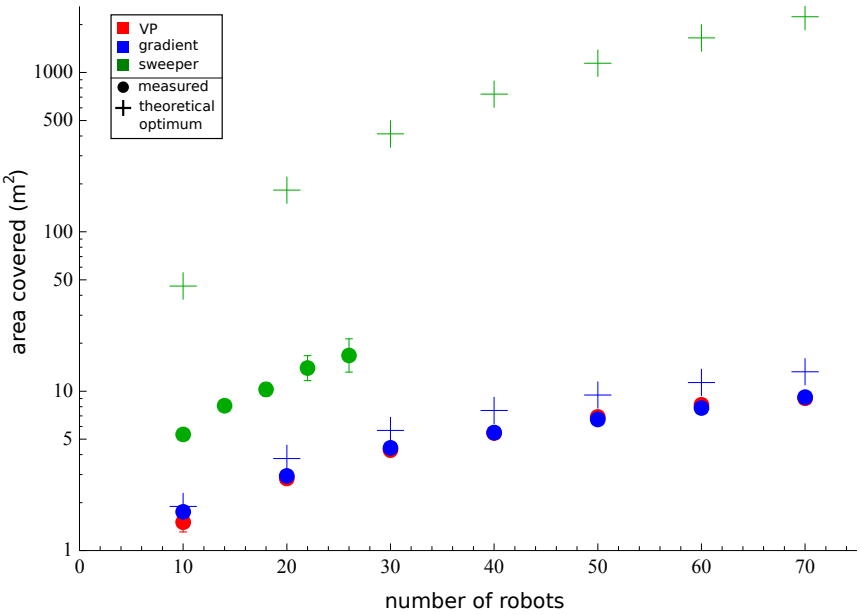


Figure 2.12: This shows the total area of the world which was explored by each of the three algorithms, as well as their theoretical optimal coverages. There are standard–deviation error bars on all the measured points, but for most of them, the error bars are smaller than the points themselves. With more than about 25 robots, the `sweeper` algorithm does not work reliably.

One could imagine a perfect area sweeping method in which the beacons form a single-file line reaching out from the nest and sweep the world. That is to fragile for the forces-based method described here (any loss in communication would cause failure), but it is a valid theoretical optimum. The amount of area that could be swept by such a long chain of robots is shown as the theoretical maximum in Figure 2.12. To increase robustness, the `sweeper` algorithm forms a double–file line of robots by using two anchors and two pullers. The fact that the theoretical optimum assumes a single–file line but the `sweeper` algorithm forms a double–file line is the reason for the difference between the theoretical and actual coverage of the `sweeper` algorithm in Figure 2.12.

2.4 Summary

This chapter presented three distributed foraging algorithms for robot swarms – the `VP` algorithm, the `gradient` algorithm, and the `sweeper` algorithm. These algorithms are designed to work on robots with very simple sensing, actuation, and communication capabilities. Although the `VP` and `gradient` algorithms are based on ant foraging, they do not require the robots to deposit real chemical pheromones or

electronic beacons in the environment, rather the pheromone is an analogy for the way information is transmitted between robots.

The `VP` algorithm is the most directly biomemetic algorithm. It performs better than `randomWalk`, but when building robots, there is no reason to be restricted to bioinspiration. The `gradient` and `sweeper` algorithms take advantage of hardware not available in nature to extend their capabilities. Biological inspiration is a good place to start, but should not be restrictive.

All three algorithms search the area of the world for food, but the `VP` and `gradient` algorithms search in a static 2D manner, while the `sweeper` algorithm searches in a dynamic 1D manner. This allows `sweeper` to cover more area.

Each of the algorithms have parameters, the values for which were chosen either by intuition or experimentation. `VP` has several parameters, such as the pheromone decay rate and the amount of pheromone the walkers lay, but there is some freedom in choosing these parameters and a range of reasonable values will yield success. `gradient`, being a simpler and cleaner algorithm, is fairly parameter-free, which is an attractive property. `sweeper`, by contrast, has many parameters (mostly in the shape of the force function) and their values must be carefully chosen by experimentation, causing the algorithm to be "brittle". If using these algorithms in a new situation, for example, `gradient` is a good choice because of its lack of parameter choices, whereas `sweeper` would need to be tuned again to the new situation.

The contributions discussed in this chapter are:

- The `VP` and `gradient` algorithms, which use an ant–inspired strategy to enable a swarm of robots with simple sensing and no global communication to search the world, meaningfully share information with each other about food location, and return the food to the nest, without requiring chemical pheromones or physical marks of any kind. The concept of using stationary robots themselves as the manner in which information is localized, stored, and communicated, as opposed to chemical pheromones, physical marks, or dropable hardware, is the central novel contribution of these algorithms.

- The `sweeper` algorithm, which uses a virtual–forces inspired strategy to form a dynamic search front which sweeps the world and returns food. This algorithm is novel both in its use of robots as beacons, and in its use of virtual forces to form a foraging search front.

Chapter 3

Performance Analysis

In this chapter, the performance of the `VP`, `gradient`, and `sweeper` algorithms will be measured and analyzed.

The swarm foraging algorithms described in the previous chapter are designed to find food and return it to the nest. In some applications, one might only care about how fast the swarm finds the target, about how many robots are needed, or about the furthest food a particular swarm can find. This study chooses three metrics which broadly capture the performance of the foraging algorithms and give insights into their performance and relative strengths and weaknesses. When analyzing the performance of swarm foraging algorithms, the chosen metrics are: the success rate of the algorithm (what fraction of runs returned food), the speed with which the swarm finds the food, and the total amount of food the swarm returns to the nest. These metrics give a sense of whether the algorithm worked at all, how well it worked, and how fast it worked.

The `VP` and `gradient` algorithms sacrifice robots to be stationary beacons, in the hope that the navigation benefit they provide will aid the remaining walkers. It could be the case that this sacrifice is not worth it. If it requires so many beacons to provide good navigation capability that there is only one walker left, then even though the walker can navigate well, it still may perform worse than random walk. The metrics selected here will be able to shed light on this question. They will show that the sacrifice is in fact a good one, unless the food is so far away that the swarm can't reach it at all.

Other metrics that could be of interest include robustness to robot failure, tolerance to sensor noise, and energy usage. Although robustness is not explicitly tested, it is captured by the metrics used in this study. If a robot failure causes the failure of the entire algorithm, that will be captured by the success metric. Similarly, if a robot failure delays the swarm in finding the food, or causes it to return less, that will also be measured. Energy efficiency is not studied here because the robots on which the hardware model is based are not significantly energy constrained. When working with robots which have significant energy constraints, efficiency would clearly be an

important metric.

3.1 Test Parameters and Metrics

There are several test–related parameters which must be chosen, such as world size, food placement, run time, and communication radius. These are chosen by comparing them to the physical robots which will ultimately be used. The world setup, including positions of nest and food, is shown in Figure 3.3a. (See section 3.3 for results in worlds B, C, and D.)

- **world size** — World size can be non–dimensionalized by dividing by robot body length. The robot model assumes a body diameter of about 8 cm. A square world in which each side is 50 times the body length would be 4m×4m, which is what is used in these tests. Worlds with and without obstacles are studied.

- **robot density** — The number of robots in the swarm divided by the total world size gives the average robot density. This chapter explores the effect of this parameter, presenting results as a function of both absolute number of robots and robot density. The absolute number of robots varies between 10 and 110.

- **nest–food separation** — This can be non–dimensionalized by dividing by world size. The nest–food separation is varied from 10% to 65% of the world size. In these experiments, there is only one food pile. It never runs out of food so that the foraging algorithms operate continuously.

- **run time** — To non–dimensionalize the run time, the analysis uses the number of direct nest–food traversals that are theoretically possible during the run (assuming the robots could travel straight to the food and back) for the largest nest–food separation. Each simulation is run long enough for a robot to theoretically traverse the nest–food distance 50 times. Given the implementation and time resolution of the simulator, that yields a run time of 15000 time steps.

- **communication radius** — The non–dimensional parameter in this case is communication radius divided by body length. Given available hardware (see Chapter 65), a ratio of 10 is conservative. This analysis assumes a communication radius equal to 10 times the body length, or 80 cm.

This analysis focuses primarily on the effects of the number of robots and the nest–food separation. These are the two most interesting and relevant parameters. Number of robots, world size, and robot density are all simply related, so only the number of robots is varied. Similarly, nest–food separation and communication radius are also

related. Increasing the communication radius has a parallel effect to reducing the nest–food separation. In this case, the nest–food separation is the parameter which is varied, because the analysis is concerned with how far a given swarm can reach.

Three metrics are used to measure the performance of the algorithms.

- **fraction of runs which returned food** — A basic success metric, answers the simple question "how often does the algorithm work?"
- **time at which the food was first found** — A measure of the speed of the algorithm. This does not strictly relate to *returning* food to the nest, but rather measures the time step during the simulation at which the swarm located the food in the environment. This is particularly relevant when considering foraging as a type of search.
- **total food returned** — The total amount of food that has been returned to the nest by the entire swarm. Note that in all experiments, food piles have an infinite amount of food. This is to make the analysis more general; the specific amount of food is application dependent, and using an unlimited food size allows general measurements. A specific arbitrary food size would lose data from algorithms which could return more.

The performance of the algorithm as measured by these metrics will be compared with a simple `randomWalk` algorithm as a lower bound comparison and a `GPScomparison` algorithm as an upper bound comparison.

`randomWalk` was chosen as a lower comparison algorithm because it has the smallest requirements for communication and sensor hardware – none. It represents a minimum use of coordination among robots. Furthermore, there is no guarantee that using the robots–as–beacons scheme (`VP`, `gradient`) will be beneficial. Those algorithms sacrifices robots to be beacons, which are then no longer free to walk around the world and return food to the nest. This sacrifice is made to improve swarm communication, but in some cases could significantly impact performance. In `randomWalk`, all robots are always free to move. The `randomWalk` algorithm is simple: robots walk straight until they detect an obstacle (or another robot) in front of them, at which point they turn a random amount and walk straight again.

While `randomWalk` serves as a lower bound hardware comparison, `GPScomparison` is an upper bound hardware comparison. If the robots had the same hardware, with the addition of a global position measurement in a shared localization space, how well could they perform? Note that this is not an extreme upper comparison. The robots still do not have global communication, perfect locomotion, excellent odometry, a-priori knowledge of the food and obstacle positions, or extensive memory capability. With the sole addition of GPS, the `GPScomparison` algorithm works as follows. Robots begin by randomly exploring the world, because they have no knowledge of where the food is. Although they know where they have been, they can not

avoid area that other robots have already explored because they can not communicate their entire history globally, and even if they could, an entire world map is too big to transmit and too much for the robots to remember. However, once a robot has found the food, it can transmit a message locally with the simple GPS coordinates of the food. Robots in the communication range of this robot receive the message and retransmit it. All robots that have this information continuously retransmit it so that soon every robot knows the food location. Thereafter, the robots move in a straight path back and forth between the nest and the food. Obstacle avoidance works the same way as in `VP`, `gradient`, and `sweeper` – when a robot detects an obstacle, it turns to the left until it no longer detects the obstacle, moves forward a small amount, and then resumes its original navigation. If the obstacle is still there, it will repeat the move-left procedure until it has walked around the obstacle.

The algorithms were written and tested in a hand-designed simulator, implemented in Java®. This simulator, used throughout the dissertation, was built from the start as a multi–robot simulator. It is capable of modeling motion, collisions, local communication, and food transport in a continuous world, and it can be easily extended to simulate other effects such as the virtual forces used in the `sweeper` algorithm. A continuous world model was chosen over a gridded world environment so that the algorithms would face real problems such as collisions and congestion. Screenshots of the simulator's graphical output option were used to demonstrate the behavior of the algorithms in the previous chapter (see page 23).

3.2 Performance in an Obstacle–Free World

As stated earlier, two parameters will be varied in this analysis: number of robots and nest–food separation. First, the number of robots will be varied from 10 to 110 while holding the nest–food separation at a constant 1.6m, or 40% of the world size. Second, the nest–food separation will be varied from 10% to 65% of the world size while holding the number of robots constant at 30. Each of these five algorithms (`randomWalk`, `VP`, `gradient`, `sweeper`, and `GPScomparison`) will be run for 15000 simulation time steps in a world with no obstacles. Results from varying the number of robots are shown in Figure 3.1 and results from varying the nest–food separation are shown in Figure 3.2.

3.2.1 Effect of the number of robots

As can be seen in the upper plot in Figure 3.1, each algorithm requires a certain number of robots to work at all. `VP` and `gradient` both begin to work around 30 robots, achieving consistent success at 70 robots. `sweeper` works with many fewer robots. With 20 robots, for example, `VP` and `gradient` do not work at all, but `sweeper` returns food approximately 85% of the time. `randomWalk` does have a consistent

100% success rate, meaning it always returns food, but this does not mean that it does so quickly or returns a lot. When letting robots run free in the world for 15000 time steps, it is almost assured that at least one of them will happen to run into the food and wander back to the nest.

The next plot (in the middle) shows the time at which the swarm first found the food in the environment. `sweeper` takes an order of magnitude longer than any of the other algorithms. The robots need a lot of time to respond to the virtual forces. For example, if the pullers are trying to form the line, the robots several steps away will feel the pullers influence only weakly because that influence needs to be conducted through all the other robots. It takes time to form the line in the first place, and time to slowly sweep the world. The other three algorithms (`VP`, `gradient`, and `randomWalk`) perform similarly to each other on this metric. `randomWalk` is slightly faster, but this just measures the first time a robot randomly picks up food. All three algorithms expand, but `randomWalk` and `GPScomparison` are not concerned with keeping a connected beacon network as the other two are, so they can expand faster.

Finally, the lower plot shows the total amount of food returned by the swarm. Obviously, `GPScomparison` outperforms all the others because soon after one robot finds the food, they all know where it and the nest are without the need for a beacon network, so they return a lot of food. (Recall that in `GPScomparison`, the robots have global positioning but not global communication. When a robot finds the food, it uses local communication to share the food location with its neighbors, they retransmit it, and the information spreads through the swarm.) `VP` has very poor performance until a large number of robots are used. Even then, it only returns a small amount. `gradient` does better. It returns food with fewer robots and always returns more than `VP`.

`randomWalk` shows an interesting trend. As the number of robots increases, the performance also increases because there are more robots available to transport food. However, when there are more than about 30 robots, it is so crowded that they have trouble moving and the performance again decreases. With more and more robots in the field, robot collisions become dominant and the world is too crowded. This congestion effect is clearly visible in the `randomWalk` data. With 110 robots, the world is so congested that the robots spend most of their time avoiding each other.

The performance of `sweeper` is fairly consistent across all numbers of robots. As long as there are enough robots to form a chain long enough to reach the food, `sweeper` will find it and return it well.

The fact that `VP` and `gradient` perform better than `randomWalk` at all shows that some coordination is being achieved between the robots using the virtual pheromones and gradient values. These algorithms sacrifice some robots to be immobile beacons, and these robots are no longer picking up and dropping of food. It could be the case that this sacrifice outweighs the benefit derived from the virtual pheromone or gradient. If that were so, using the `VP` or `gradient` algorithms at all would be

harmful and `randomWalk` would be better. In fact, they often outperform `randomWalk`, which shows that the sacrifice of some robots to be beacons confers a net benefit on the algorithm. Furthermore, although each algorithm uses robots as beacons to achieve a coordination benefit, `gradient` derives more benefit from these beacons. This benefit is reduced and eventually eliminated in both cases at small numbers of robots, however, because in that case the robots are mostly used as beacons so there are few walkers left.

Both `gradient` and `VP` exploit the ant–inspired idea of marking the environment, but the success of the `gradient` algorithm over `VP` shows that there are more effective ways to lay marks than to simply copy the ants' method. `VP` is mostly a transcription of ant–inspired foraging into robot hardware, with necessary modifications to enable it to work (such as using robots as pheromone markers). This may not be the best way to spread information, though, and the success of `gradient` is an example of a move beyond direct biological mimicry.

3.2.2 Effect of nest–food separation

Results of varying the nest–food separation are shown in Figure 3.2. The upper plot shows the simple success rate. As expected, when the nest and food are close together, the algorithms perform well, and as the separation increases, the success rate decreases. `VP` and `gradient` follow similar trends, with `VP` slightly worse. At a separation greater than about 1.7m, neither of these algorithms is useful at all because they can only expand so far. Once the swarm has expanded as far as it can and all robots have become beacons, it can not go any further and it will fail. `sweeper` also has a maximum distance at which it can find food, around 2.5m. If the line that `sweeper` forms is not long enough to reach the food, then the line will just keep sweeping and the algorithm will never find food.

The middle plot show the effect that the nest–food separation has on the time at which each algorithm first finds the food. As expected, placing the food further away generally causes the algorithms to take longer to find it. `sweeper` is an exception. `VP` and `gradient` expand outward from the nest. When they have found the food, they mostly stop expanding and start returning it. If the food is further away, they must spend more time expanding to find it. `sweeper`, on the other hand, forms a constant length line and sweeps, so the distance of the food from the nest has no effect on how long it takes to find. Of course, if the food is extremely close to the nest, all of the algorithms find it quickly. If the food is only 40cm from the nest and the robots have an 8cm diameter, then with 30 robots in the world, there is a robot near the food almost at the first time step. `randomWalk` and `GPScomparison` have nearly identical performance because they both perform a random walk to find the food the first time.

Total amount of food returned vs nest–food separation is shown in the lower plot. `VP` and `gradient` fall off quickly, returning almost nothing after about 1.7m. `randomWalk` degrades more smoothly, as expected. If the food is further away, the robots have

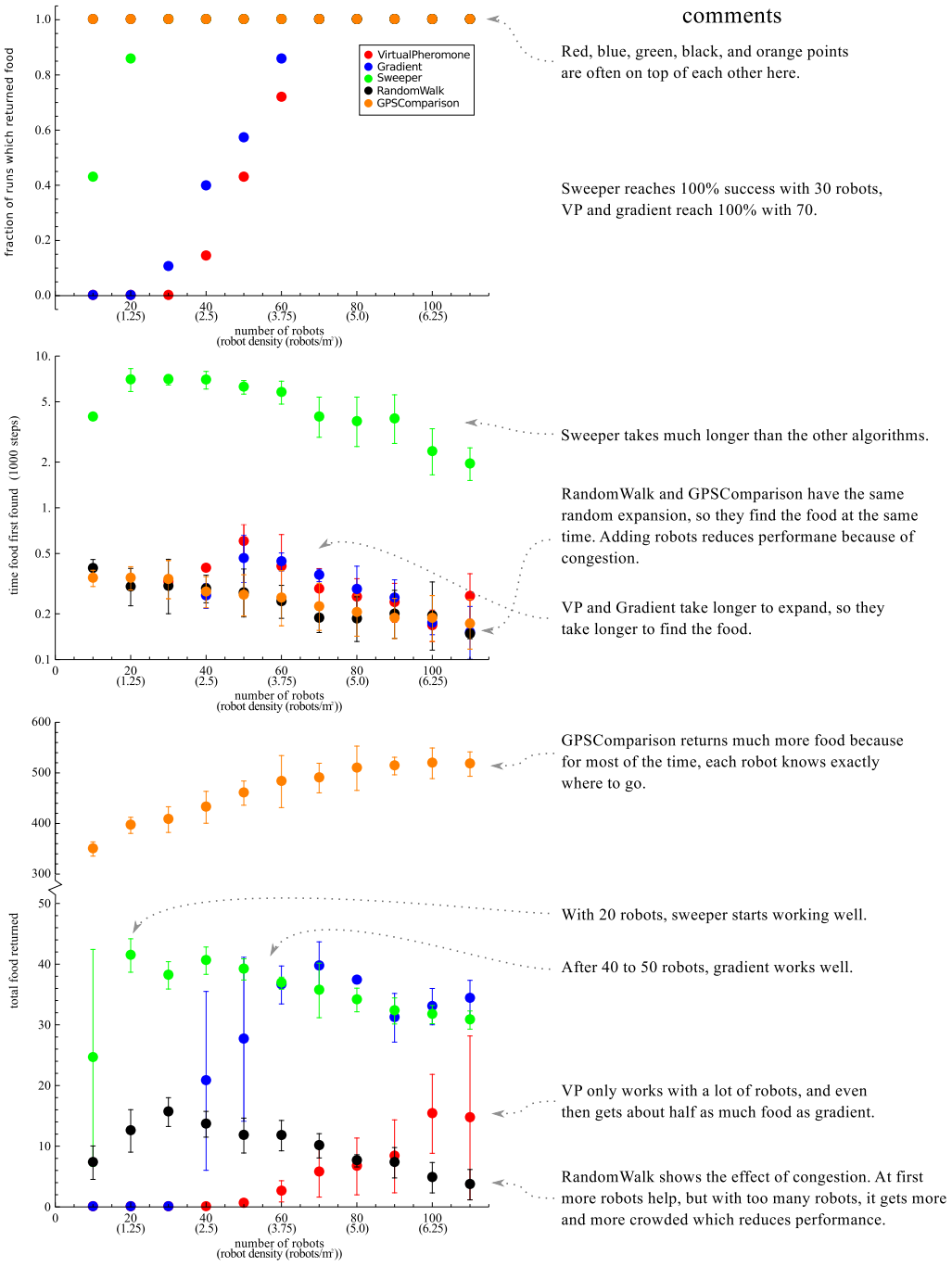


Figure 3.1: These plots show the performance of `VP`, `gradient`, `sweeper`, and `randomWalk` as measured by all three metrics – success rate (top plot), time at which food was found (middle plot), and total food returned to the nest (bottom plot). These plots show the effect of the number of robots. Each point is an average of 30 runs, with standard deviation error bars. Note the log scale on the vertical axis of the middle plot and the broken vertical axis on the bottom plot.

a longer distance to travel and the probability of successfully traversing it by chance decreases. `sweeper`, by contrast, returns a constant amount of food regardless of the food's distance. This continues until the maximum distance, at which point it rapidly falls off. Because `sweeper` forms a line of constant length regardless of where the food is, it does not return a smoothly decreasing amount of food with increasing nest–food distance. `GPScomparison` again far outperforms the other algorithms, and congestion is again visible. When the nest and food are very close together, all the robots remain crowded around and it is difficult for the robots to move. More separation gives the robots more room to maneuver and increases performance.

Overall, again, `sweeper` returns the most food but takes the most time to find it. `gradient` consistently outperforms `VP`. There is a range inside which any algorithm works, a range beyond which nothing but `randomWalk` works, and ranges in the middle in which other algorithms perform best. These ranges, and the possibility of choosing the best algorithm for the given range, will be explored in the next chapter.

3.3 Tests and Results in Worlds with Obstacles

When deployed in the real world, swarms will not likely operate in a perfect obstacle–free world. It is more likely that there will be unknown obstacles that restrict their movement and communication. The algorithms should be tolerant to obstacles, and this section measures their ability to function in such environments.

Obstacles present a challenge to distributed foraging algorithms because they make search and navigation more difficult. They do this by changing the shortest path between the nest and the food from a straight line into a more complex path requiring turns. In the obstacle–free world used above, the shortest path involved zero turns. This section studies obstacle configurations requiring one and two turns. These configurations are shown in Figure 3.3. In configuration B, a simple obstacle blocks the most direct path, such that the swarm must find a one–turn path around the obstacle. Configuration C has two obstacles that require the robots to take a more complex two–turn path to get to the food and back.

`VP`, `gradient`, and `sweeper` do obstacle avoidance in nearly the same manner. Each walker has a direction it wants to go, determined either by virtual pheromones, gradient values, or virtual forces. For each of these three algorithms, when a robot encounters an obstacle, it attempts to avoid it by simply turning left and moving forward. After that, it turns back in the direction it wants to go and if the obstacle is still there, it repeats the obstacle avoidance maneuver.

The performance of the algorithms in the presence of obstacles is analyzed using the same three metrics used to analyze the no–obstacle case, described in section 3.1. For each obstacle field, the number of robots is varied over the same range as before. Varying the nest–food distance is not as meaningful here; it is the obstacles that change the difficulty of the environment, not the placement of the food. Results from

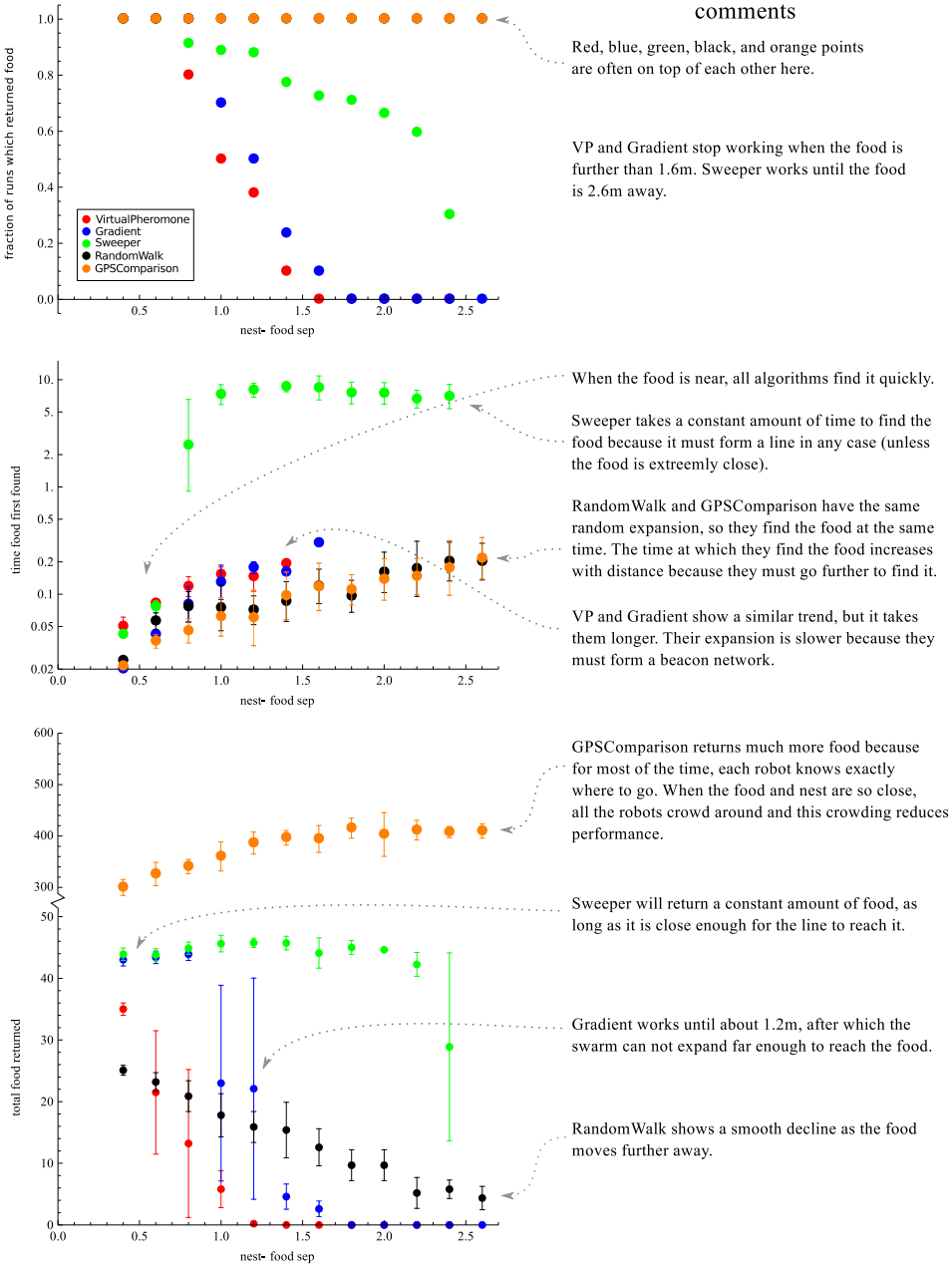


Figure 3.2: These plots show the performance of `VP`, `gradient`, `sweeper`, and `randomWalk` as measured by all three metrics – success rate (top plot), time at which food was found (middle plot), and total food returned to the nest (bottom plot). These plots show the effect of the nest–food separation. Each point is an average of 30 runs, with standard deviation error bars. Note the log scale on the vertical axis of the middle plot and the broken vertical axis on the bottom plot.

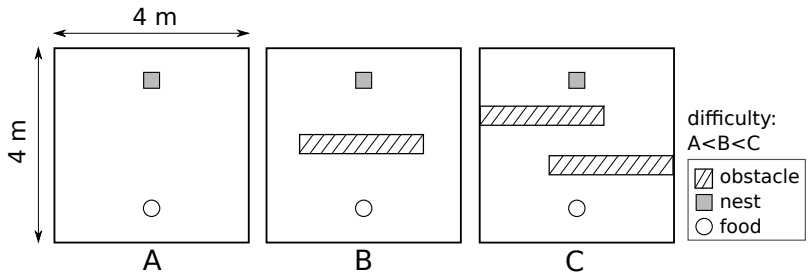


Figure 3.3: There are three world configurations. Results from configuration A are discussed in section 3.2, and results from configurations B, and C are discussed in section 3.3.

the single obstacle tests are shown in Figure 3.4 and results from the double obstacle tests are shown in Figure 3.5.

For the single obstacle case, similar trends to the no–obstacle case are seen in the success of the algorithms, as shown in the upper plot in Figure 3.4. There is a minimum number of robots which each of the algorithms needs to be successful, this minimum is larger for `VP` and `gradient`, and lower for `sweeper`. Given enough robots, though, each of the algorithms can reach 100% success. The double obstacle case, however, is much more difficult. The `sweeper` algorithm never works at all, and `gradient` needs 110 robots just to achieve 30% success.

The single obstacle slows `VP` and `gradient` down a lot, as shown in the middle plots. When they work (large numbers of robots), they take a long time. The swarm must expand from the nest and curve around the obstacle to reach the food. This takes a similar amount of time for `VP` and `gradient` because their expansion algorithm is the same. Again, the double obstacle case is much harder. Only at very large numbers of robots does it work at all, and it is very slow. This time, the expansion must flow around two obstacles, making multiple turns. It does work, but only if there are a lot of robots and a lot of time. `sweeper` has no chance of forming a line and sweeping the world, because the line can't make the multiple turns around obstacles that would be required. In both cases, `randomWalk` is able to find the food relatively quickly because it does not have to maintain coordination, although it does take longer in the two–obstacle case.

For the total amount returned in the single obstacle case, `gradient` and `sweeper` are able to perform well with large numbers of robots. `randomWalk` is the only algorithm which returns any food for small numbers of robots. In the double obstacle case, even `randomWalk` performs badly, and `gradient` manages a few food units with the most robots. `GPScomparison` does not work at all because it is not capable of making the two turns required to navigate both obstacles. `GPScomparison` always walks directly toward the food or nest, turning left when it finds obstacles. The two–obstacle case, though, also requires right turns which `GPScomparison` can not make. `GPScomparison` always makes left turns, causing it to get pinned by the obstacles.

`gradient` has the same obstacle avoidance method, but robots follow the shortest path created by the beacons, which is capable of having right turns.

3.4 Summary

In this chapter, the performance of the `VP`, `gradient`, and `sweeper` algorithms was measured and analyzed. Overall, `gradient` outperforms `VP` in nearly all situations. `gradient` performs best at finding food quickly when it is near the nest. `sweeper` can find food further away, but is slower in finding it. In difficult obstacle configurations or with small numbers of robots, `randomWalk` is still the best algorithm.

The `gradient` and `sweeper` algorithms are tolerant to a single obstacle, but they need more robots to function well. None of the algorithms are capable of performing well in the double obstacle case, but `gradient` is probably the best hope. With obstacle configurations requiring two or more turns in the shortest path, a large number of robots and a lot of time will be required for the swarm to expand and fill all the areas of the world. In these cases, the tradeoff between `randomWalk` and `gradient` is not so clear.

As discussed in the previous chapter, the `gradient` algorithm is fairly parameter free, suggesting that it is a good choice to be adapted to other scenarios. `sweeper`, by contrast, is tightly tuned to this hardware model. The force function was carefully chosen so that lines of robots would form and not break. If adapting these foraging algorithms for a different task or situation, `gradient` would be easy because of its lack of parameters whereas `sweeper` would require a careful retuning.

Even after being tuned for these experiments, the line in the `sweeper` algorithm does occasionally snap. In this case, the pullers can recognize that the line snapped because their nest gradient value goes to zero. If this happens, they can reverse their pulling force by 180° and try to push the crowd of robots backward and repair the break. This sometimes works and does reduce the failure rate, but unrecoverable line snaps still sometimes occur. The `gradient` algorithm also has built in failure recovery mechanisms, such as the constant recalculation of gradient values. If a beacon were to fail or be pushed out of position by a walker, the gradient field on the surrounding robots would need to be updated. Otherwise, walkers would get incorrect navigation information which could lead them permanently astray. Such failures do not happen in the simulator, but they do in the real robots (as will be seen in Chapter 5), and this constant updating of gradient values helps keep the gradients correct.

The contributions discussed in this chapter are:

- If it can reach the food, the `gradient` algorithm is capable of outperforming `randomWalk`, showing that it derives a coordination benefit from the beacons. In other words, the sacrifice of some robots to be stationary beacons is a worthwhile sacrifice. There are fewer robots available to do the actual work of carrying

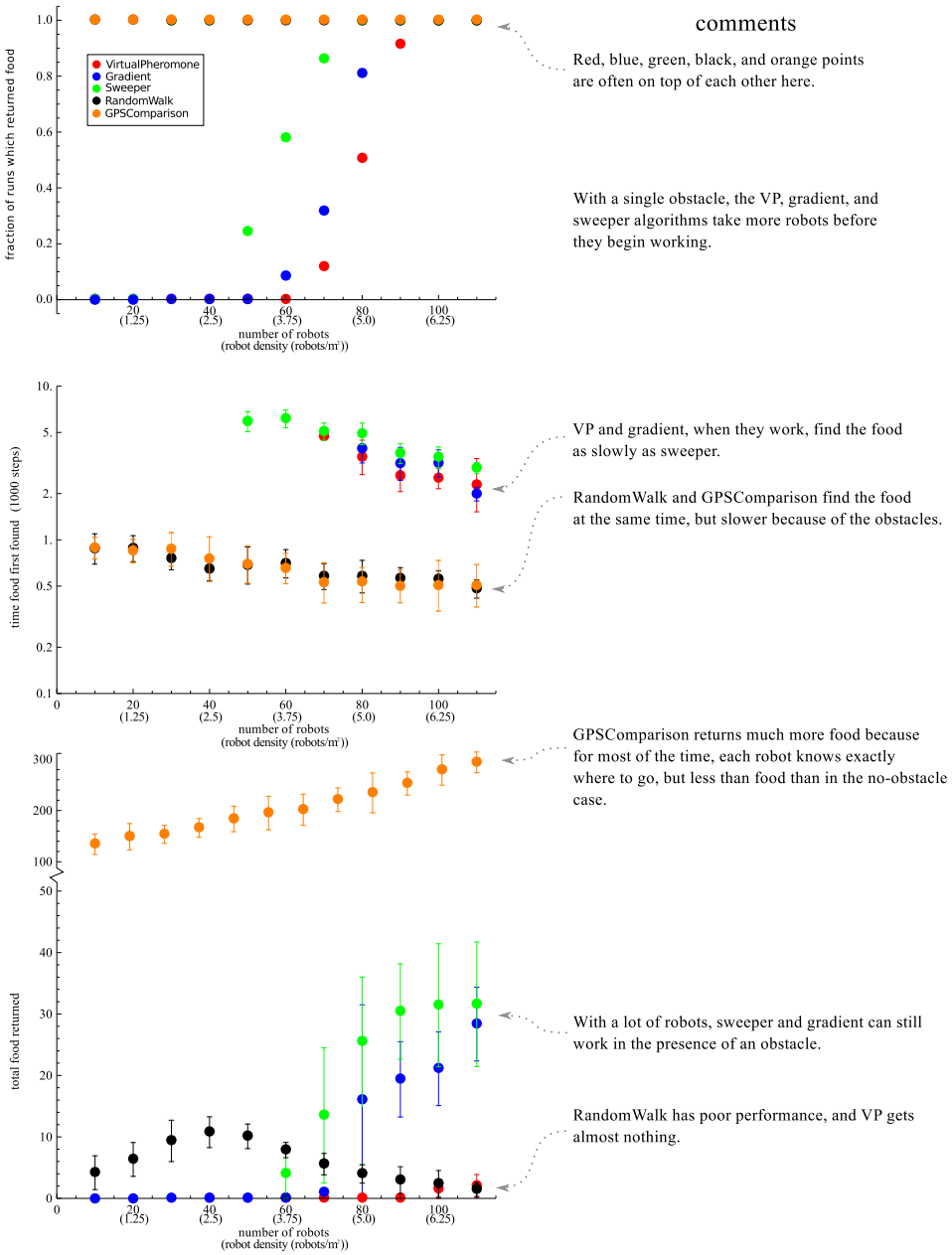


Figure 3.4: These plots show the performance of `VP`, `gradient`, `sweeper`, `random-Walk`, and `GPScomparison` as measured by all three metrics – success rate (top plot), time at which food was found (middle plot), and total food returned to the nest (bottom plot). These are results from the single obstacle case. Each point is an average of 30 runs, with standard deviation error bars. Note the log scale on the vertical axes of the middle plot and the split axis on the bottom plot.

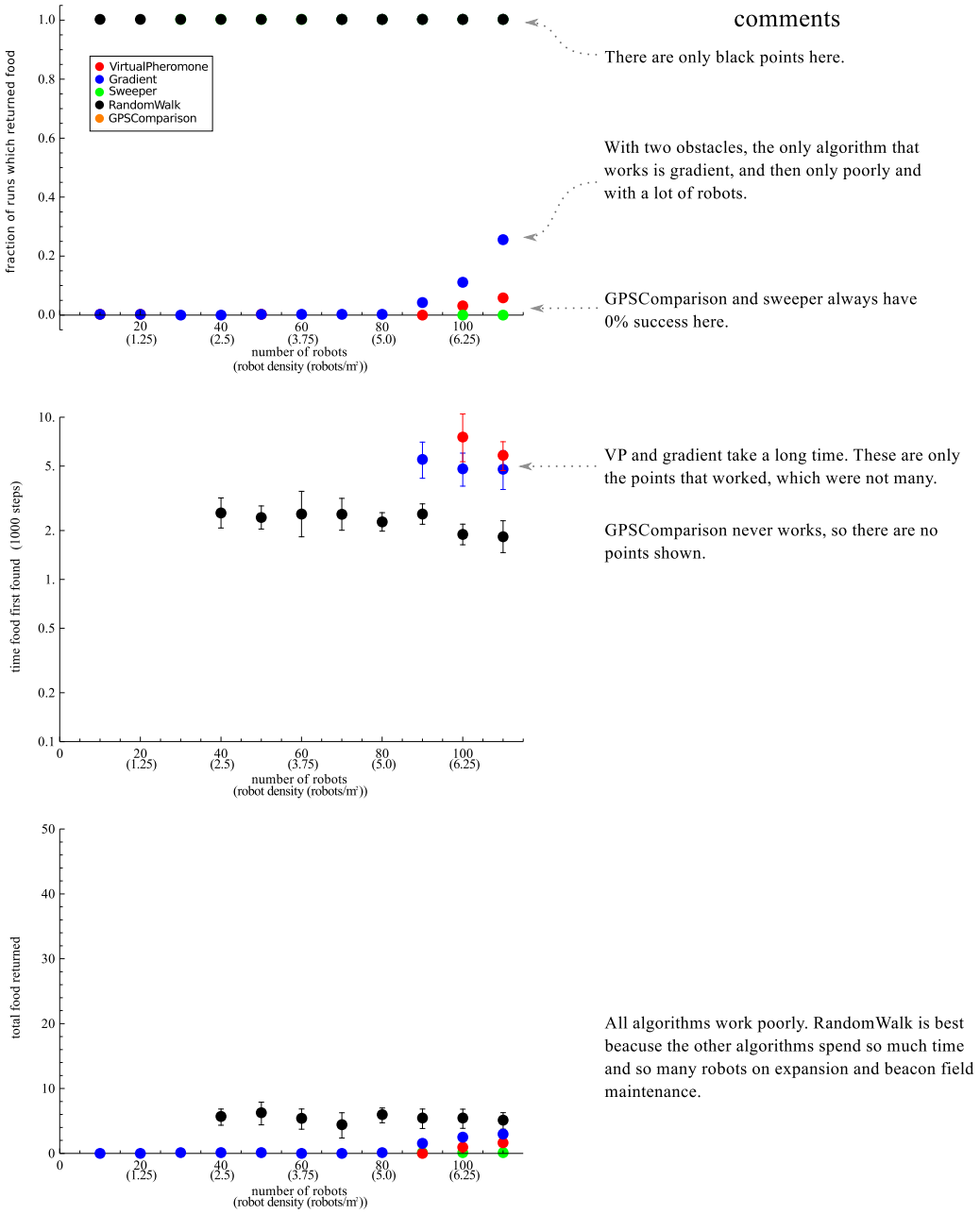


Figure 3.5: These plots show the performance of `VP`, `gradient`, `sweeper`, `random-Walk`, and `GPScomparison` as measured by all three metrics – success rate (top plot), time at which food was found (middle plot), and total food returned to the nest (bottom plot). These are results from the double obstacle case. Each point is an average of 30 runs, with standard deviation error bars. Note the log scale on the vertical axes of the middle plot and the split axis on the bottom plot.

food back to the nest, but their contribution to gathering and disseminating information more than makes up for this.

- The ant-inspired idea of marking the environment is an effective method of distributed coordination, but direct biological mimicry is not necessarily the best way to implement this marking. Both `VP` and `gradient` are biologically inspired algorithms, but `VP` nearly copies the behavior of the ants while `gradient` moves beyond it, finding a better method.

- The `sweeper` algorithm is also capable of performing the foraging task, but is not directly comparable to `gradient`. In some cases, it is better than `gradient`, and in some cases worse. This tradeoff will be more fully explored in the next chapter.

The analysis in this chapter has shown that `gradient`, `sweeper`, and `randomWalk` are each the best algorithm to choose for different values of nest–food separation. The next chapter will develop a method for the swarm as a whole to choose its algorithm to suit the given situation.

Chapter 4

Swarm–Level Algorithm Switching

The performance of a robot swarm algorithm is affected by the environment in which the swarm operates. In a search task for example, the location and number of search targets and presence or absence of obstacles could affect the performance of the swarm. One algorithm may be more efficient at finding nearby targets while another may be better at finding distant targets. Because the specifics of the environment are generally not known before the swarm begins its execution, the swarm itself should be able to intelligently choose its own algorithm based on the observed performance. An interesting challenge is whether a robot swarm, as a whole, can assess the success or failure of an algorithm and, as a whole, switch algorithms to increase its success.

Colony–level algorithm switching is difficult for two reasons. First, the information on which the colony will base its decision is distributed throughout the environment and not detectable by any one robot. Therefore, the environment detection must be distributed. Second, if a swarm algorithm relies on strong coordination, then switches must be nearly unanimous and synchronized. In this case, changing algorithms will not help unless all individuals change to the same algorithm at the same time. Both of these actions—environment sensing and algorithm switching—must truly take place on the colony level, as opposed to the individual level.

This chapter builds on the `gradient` and `sweeper` algorithms and presents a third algorithm, termed the `adaptive` algorithm. All of these are based on a few core "sub–algorithms" which the swarm intelligently switches between in various combinations and at various times. In the `adaptive` algorithm, the colony cooperates to detect when one algorithm is failing and switches to another, increasing performance.

To evaluate the algorithms, food is placed at varying distances from the nest. The `gradient` algorithm finds the food quickly but only in a short range, and the `sweeper` algorithm is slower but has a larger range. If the food is very far away, neither of these algorithms work and `randomWalk` is the only option left. The `adaptive` algorithm is capable of high–level switching, and the swarm is able to choose the algorithm best suited to the current food location.

4.1 Related Work

Algorithm switching and colony–level decisions have been used in the field of swarm algorithms in the past. The two most relevant areas of related work are in task allocation and quorum sensing.

4.1.1 Task Allocation

Task allocation is useful when different individuals in a swarm need to perform different tasks [28]. In a termite mound model, for example, the colony may need to clean the mound, defend the mound, and forage for food. The colony needs to perform each of these subtasks simultaneously, and because there is no central leader, each individual needs to decide which subtask it should perform. These decisions need to be made such that enough individuals are working on each task, and the distribution may need to be adjusted over time. A common way to accomplish this is by establishing thresholds for each task [3, 11]. For example, while an individual is doing its task, it will measure the amount of clutter or dead bodies in the nest. If the amount that the individual detects is above a specific threshold, the individual will become a cleaner. Similarly, if the amount of food in the nest goes below a specific threshold, it will become a forager. This strategy allows the task distribution to change dynamically. After some individuals have been cleaning the nest for some time, there will be less clutter, and some individuals will switch tasks because some other threshold will take their attention.

Task allocation is different from the algorithm adaptation under consideration in this chapter because task switching is an individual decision whereas the `adaptive` algorithm requires colony–level decisions. In task switching, each individual can made a decision about whether to switch tasks based on the information it can sense. This is not always the case with the `adaptive` algorithm. A threshold system will not work for the `adaptive` algorithm because such a system does not spread information throughout the world. Because individuals sense locally, they evaluate thresholds locally, and make decisions locally. The `adaptive` algorithm requires integration of information which could be anywhere in the world. The `adaptive` algorithm needs the whole swarm to change algorithms, but not every individual in the swarm can sense the information that would cause the change, so decisions can not be purely local.

4.1.2 Quorum Sensing

Quorum sensing is a system by which a collection of robots with local communication can come to a consensus [37, 38, 39]. It is a distributed voting scheme in which each robot votes for its choice and tallies votes from other robots as it encounters them. If it encounters enough votes for a specific choice, it commits to the choice.

Quorum sensing could be applied to the algorithm switching problem by instructing the robots to come to a consensus about which algorithm they should run. This could be useful in many situations, but two drawbacks prevent it from being useful for the `adaptive` algorithm.

First, vote collection relies on the robots being able to encounter each other. A robot will collect votes from the other robots it encounters as it travels. This is a problem because in the `gradient` algorithm, many of the robots act as stationary beacons and do not move at all. Quorum sensing relies on each of the robots being able to eventually encounter a large number of other robots, which is not the case with the algorithms used here.

A second related problem is that the information that would cause the swarm to want to change algorithms may only be detectable by one robot. This robot would vote to change algorithms, but all the other robots would still vote for the previous algorithm, thinking that it is still working. A simple voting scheme is insufficient because it can ignore small but significant pieces of information.

4.2 Adaptive Algorithm Description

This section describes the `adaptive` algorithm and the means by which algorithm switches are made at the individual– and colony–level. Figure 4.1 diagrams the relationship between the algorithms and their parts. See section 2.2.3 on page 17 for a description of the `gradient` algorithm and section 2.2.4 on page 19 for a description of the `sweeper` algorithm.

The `gradient` and `sweeper` algorithms each have strengths and weaknesses. `gradient` operates in a short range but finds the food quickly, while `sweeper` has a longer range but is slower. (This is quantified in section 4.3.) The goal is to use each algorithm when it is most appropriate. The `adaptive` algorithm combines the benefits of these two algorithms, along with `randomWalk`, by trying each one in sequence and choosing the best one for a given situation. It first tries `gradient`, which would work well if the food is near enough to use it. If not, it switches to `sweeper` to get food further away. If it still doesn't find the food, it switches to the last resort – `randomWalk`. Switches between these three algorithms are made in a distributed manner at the colony level. To accomplish this, a third gradient is included.

4.2.1 Local Description

Robots begin with an algorithm very similar to the `gradient` algorithm described earlier. They are split between walker and beacon robots as before, but they maintain three gradients as opposed to two. The third one measures how far each beacon is from any walker robot. This requires all walker robots to transmit a single bit of information indicating their presence and identity as a walker. The beacons

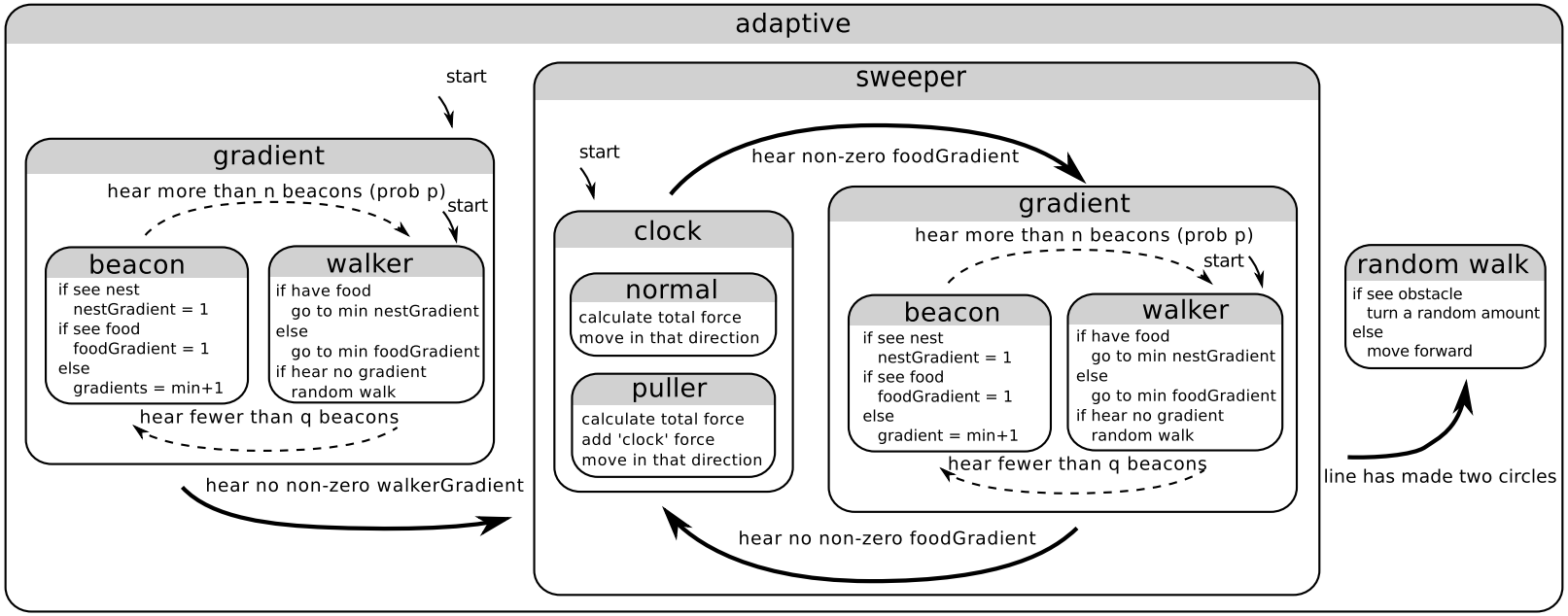


Figure 4.1: This figure describes the relationships between the `gradient`, `sweeper`, and `adaptive` algorithms, and their switches. The "start" arrows indicate how each algorithm begins. Bold arrows indicate colony–level switches and dotted arrows indicate individual–level switches.

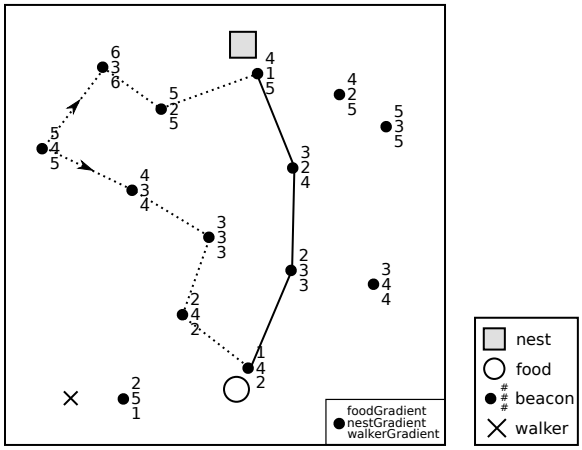


Figure 4.2: This is similar to Figure 2.2b on page 17, except a third number indicating the `walkerGradient` of each beacon has been added. If there were just a single walker remaining, indicated by a cross in the lower left, the beacons would update their `walkerGradient` values as shown.

then transmit a 1 for the `walkerGradient` if they can see a walker, and $min + 1$ if they can not see a walker. Figure 4.2 shows a diagram for the `gradient` algorithm with the `walkerGradient` added. Other than maintaining the `walkerGradient`, the robots execute the `gradient` algorithm exactly as described above. Maintaining this `walkerGradient` is something they do in addition, but it is irrelevant to the execution of the `gradient` algorithm.

To implement adaptive foraging, robots need to explicitly detect when to change algorithms. If a robot does not see any (non-zero) `walkerGradient` for several time steps in a row, it will completely switch algorithms from `gradient` to `sweeper`. Thereafter, if the line of robots has swept the world twice and still has not found food (as evaluated by the pullers), the pullers send a signal through the beacon network causing every robot to again switch algorithms to `randomWalk`. These transitions are shown in Figure 52.

4.2.2 Global Behavior

The swarm will begin executing the `gradient` algorithm. There are many walkers at the beginning of the execution, so the values for `walkerGradient` are all fairly low. If the swarm finds the food, it returns it as usual, and the `walkerGradient` remains irrelevant. If, however, the swarm expands to the point that all robots have become beacons and the swarm has still not found the food, then there will be no walkers left. When the last walker becomes a beacon, suddenly no beacon anywhere in the swarm will have information on which to broadcast a `walkerGradient`, so all `walkerGradient` values will revert to 0. A short time later, they all decide nearly–

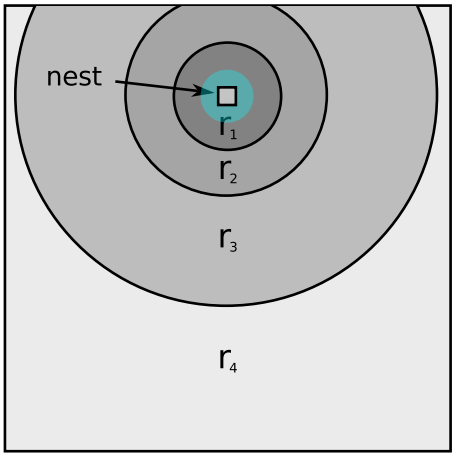


Figure 4.3: This shows the regions described in section 4.3.1. The regions indicate where the food could be found. The light blue area around the nest shows the approximate starting area of the robots.

simultaneously to switch algorithms and begin the `sweeper` algorithm. From this point on, the swarm proceeds as normal as if it had just begun the `sweeper` algorithm, pulling out a line of robots and sweeping the world. Once this line has swept around the world twice without finding food, the swarm switches to `randomWalk`, which is the only option left. `randomWalk` does not involve coordination, which relieves the robots of the requirement of staying near each other. This is the only way to get food so far away.

4.3 Performance

The `gradient`, `sweeper`, and `adaptive` algorithms were tested in the continuous-world multi-agent Java® simulator. An unlimited food source was placed at varying distances from the nest, with a swarm of 20 robots trying to find and retrieve it. As before, simulations were run for 15000 time steps.

The performance of the algorithms is assessed using the same three metrics used in the previous chapter:

- whether or not the swarm found the food,
- how quickly it found the food, and
- the total amount of food it returned.

The results are shown in Figure 4.4. Each data point represents an average of 30 runs.

4.3.1 Region–Based Analysis

Based on observed performance (Figure 4.4), the world is divided into four distinct regions, diagrammed in Figure 4.3.

If the food is inside r_1, it is so close to the nest that any reasonable algorithm will find it. As shown in the top plot, in the region inside r_1, all algorithms find the food almost all the time. The middle plot shows that in this region, the food is also found quickly regardless of which algorithm is chosen, and the bottom shows that all algorithms return a lot of food.

When the food is in r_2, `gradient` works most of the time, and when the food is further than r_2, it fails most or all of the time. In r_2, `gradient` finds the food, finds it quickly, and returns a lot of it. Outside of r_2, `gradient` works poorly, eventually never finding the food. As the robots expand and form a beacon field, eventually the swarm will expand to its maximum size and there will be no walkers left to continue the expansion. If the food is beyond this critical radius, there is no way for the `gradient` method to get it.

The `sweeper` algorithm is also capable of finding the food in r_2 and returning a lot once it has found it, but as seen in the time plot, takes much more time to find it.

In r_3, the `gradient` algorithm is essentially useless, and the `sweeper` algorithm performs well, forming a line and sweeping the world. The plots show that in r_3, `sweeper` finds the food roughly 70% of the time and returns a lot of it, although it takes a long time to locate it.

Outside r_3, even the `sweeper` algorithm fails because the line of robots can not reach that far. At this point, the food is so far away that only `randomWalk` can work. Although it has a high success rate and finds the food fairly quickly, it returns a very small amount because the food is so far away.

region	description
r_1	any algorithm works
r_2	coordination needed, `gradient` works well
r_3	too far for `gradient`, `sweeper` works well
r_4	too far for `sweeper`, only `randomWalk` works

In different regions, different algorithms perform best. In r_2, `gradient` is best because it finds the food and finds it faster than `sweeper`. In r_3, `sweeper` is best because `gradient` doesn't work any more. In r_4, `randomWalk` is the only option left. In every region, the `adaptive` algorithm is able to choose the most appropriate foraging method. In r_2 it runs the `gradient` method, in r_3 it runs the `sweeper` method, and in r_4 it runs `randomWalk`. In r_4, `adaptive` does not achieve the 100% success rate of `randomWalk` because it has spent most of its time trying previous algorithms. As the time plot shows, `adaptive` spends up to about 10000 time steps running the previous algorithms before it switches to `randomWalk`, so it has only about 5000 time steps left to use `randomWalk` to search for the food. This is often

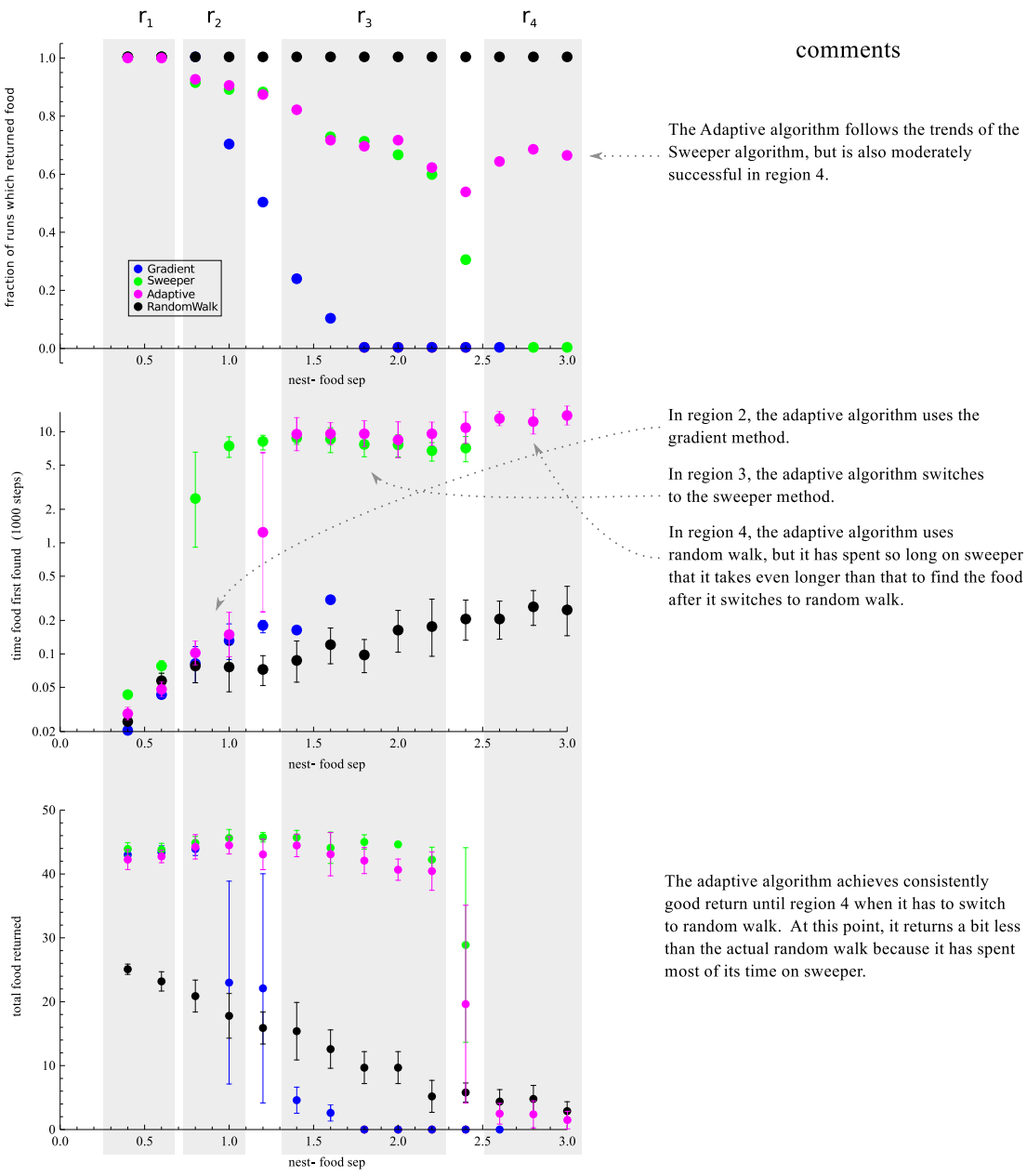


Figure 4.4: These plots show the success rate of each algorithm, the time at which food was first found, and the total amount of food returned by each algorithm. Each point represents an average of 30 runs, and the error bars indicate one standard deviation. Note the log scale on the vertical axis of the middle plot.

Table 4.1: Overall performance assessment.

	r_1, r_2, or r_3		whole world	
algorithm	success rate	time food found	success rate	time food found
`gradient`	31%	69 ± 31	19%	69 ± 31
`sweeper`	83%	4500 ± 3800	34%	5300 ± 3400
`adaptive`	85%	3300 ± 3500	68%	9200 ± 6100

enough time (about 65%, as shown in the upper plot), but not always. Because of this time penalty, `adaptive` also returns less food than straight `randomWalk` in r_4.

4.3.2 Overall Assessment

As an overall assessment of each algorithm, the food can be placed in an unknown random location. In the Table 4.1, "r_1, r_2, or r_3" indicates that the food is placed randomly anywhere in those three regions, and "whole world" indicates that the food is randomly placed anywhere. The numbers are averages over all placements. For example, if the food is randomly placed anywhere and the `sweeper` algorithm is running, the swarm can be expected to find it 34% of the time, after an average of 5300 time steps with a standard deviation of 3400 time steps.

In r_1, r_2, or r_3, the `adaptive` algorithm finds the food almost as often as the `sweeper` algorithm, but does so faster. This is because it is able to take advantage of the speed of the `gradient` method when the food is nearby. When very distant food locations are included (whole world), all algorithms suffer lower success, but the `adaptive` algorithm is able to use `randomWalk` to at least achieve some success in a very long time and does not suffer the decrease in success rate that `sweeper` does.

4.4 Algorithm Switching Generalizations

4.4.1 Framework

To generalize the concept of algorithm switching, consider a swarm which may use a set of algorithms $\mathbb{A} = \{A_1, A_2, A_3, \cdots, A_n\}$. These could be the `gradient`, `sweeper`, and `randomWalk` algorithms from this chapter, for example, although more possibilities are presented below. Under some circumstances when running algorithm A_i, the swarm may want to switch to a different algorithm A_j. The `gradient` to `sweeper` transition is an example, although loops and more complex switching arrangements are possible. If all the algorithms require large scale coordination, the swarm must be nearly unanimous and synchronized in its switching decisions (this requirement can also be relaxed, as discussed below). Then, for each $A_i \rightarrow A_j$ transition the swarm wants to make, the programmer must design triggers that each robot

can detect. When a robot running A_i detects an $A_i \rightarrow A_j$ transition trigger, it will switch to A_j. Because individuals can only switch based on what they can sense and communicate, these triggers must be designed to be detectable and evaluateable by each robot locally, although the trigger may be transmitted through robot-to-robot communication. Examples of this are given later in this section. (Note that the trigger causing an $A_i \rightarrow A_j$ transition can not necessarily by the same as the one causing $A_j \rightarrow A_i$.)

These $A_i \rightarrow A_j$ algorithm switches are colony–level switches. Colony–level switches are ones which (a) require information which is not detectable by every robot, and (b) need to be agreed on by every robot in the colony. There are also individual–level switches. The main differences between the two types are:

- First, individual switches are made by a particular robot based on the information it can perceive, whereas colony–level switches require information which is distributed throughout the environment and not directly perceivable by each robot making the decision.
- Second, colony–level switches must be nearly synchronized, whereas individual switches need not be explicitly coordinated.

4.4.2 Requirements

To achieve a colony–level algorithm switch, information must be shared throughout the swarm. For example, when the `adaptive` algorithm switches from `gradient` to `sweeper`, each robot must be aware that no walkers are left, but no single robot is capable of perceiving this. This global information is detected and shared through the beacon network. Because global information is required for colony–level switches, maintaining the connectedness of the beacon network is critical for these switches.

Three types of information can be identified:

information type	example
directly perceivable	sense a beacon ahead
directly perceivable by another robot	someone found food
only perceivable by swarm as a whole	swarm has expanded to max size

The first type of information can be detected by an individual robot using its sensors. The second type can be detected by an individual robot, but must be transmitted to be useful to others. There is no robot in the swarm with a sensor capable of detecting the third type of information; it can only be detected by the swarm as a whole through cooperation. A connected network is critical for detecting and transmitting this information. Individual switches can be made solely based on information of the first type. Colony switches require the second and third types, which requires a beacon network.

For example, if the colony is executing the `gradient` algorithm, but the robots have expanded as far as possible and the food still has not been found, there is no sensor on any robot which can detect that. It is only by sharing information about the number of walkers seen, and then reasoning about that information in a distributed manner, that the situation can be detected. Again, a connected communication network is critical.

4.4.3 Examples of Incremental Extensions

Consider a situation in which an algorithm similar to `gradient` had to operate in a world that may contain obstacles. As shown in Chapter 3, `gradient` does not perform well in complex obstacle fields and `randomWalk` may be better. So perhaps the swarm would want to run `gradient`, but switch to `randomWalk` if it detected an obstacle. In this case, as soon as a robot detected an obstacle, it would transmit a special message to its neighbors, who would then retransmit it to their neighbors. This message would serve as the `gradient`→`randomWalk` trigger, and when each robot heard the trigger, it would switch to `randomWalk`. This is a simple way for a swarm to "give up" if it needs to.

As another example, consider a foraging situation in which energy efficiency is a concern. Data in Chapter 3 showed that `sweeper` takes longer to find the food than `gradient`, although it can reach further. Because it takes more time, `sweeper` probably would also require more energy. If the colony needs to conserve energy, it may only want to run `sweeper` if its food supplies are very low. Otherwise, it will just run `gradient`, and if that doesn't find any food, the colony will stop foraging instead of spending the extra energy required to run `sweeper`. In this case, the nest would need to sense the amount of food it had and transmit that information to the robots. The nest could transmit the food amount to the nearest robot, who would retransmit it, etc. (These transmissions could be stamped with an increasing integer to prevent old data from overwriting new data.) The robots could then run the same `adaptive` algorithm as before, except the `gradient`→`sweeper` transition would have another condition checking if the amount of food at the nest (as transmitted throughout the swarm) is lower than a pre–defined threshold. This condition can be evaluated locally by each robot, making it a valid trigger.

The algorithm switching system described here can be extended to similar situations beyond ant–like foraging. Consider a swarm of flying surveillance robots with solar panels. Normally, they fly around performing their normal mission, but sometimes they need to charge. They need to find non–cloudy areas in which to charge, but if there are no sunny areas, the swarm should switch algorithms, and revert to a power–saving mode or return to a base. Because no robot can see the entire area, detecting the lack of sunny areas anywhere probably requires distributed detection and swarm–level algorithm switching. (Of course, another option would be to solve the problem with centralization or expensive hardware, but both of these solutions

eliminate the scalability of the distributed system.)

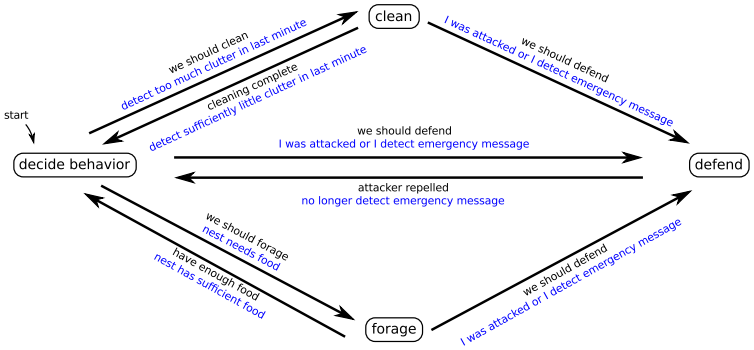
4.4.4 Examples of Broad Extensions

Algorithm switching is a general concept, useful throughout swarm algorithms. Figure 4.5 shows how several different swarm behaviors could be understood in the context of algorithm switching.

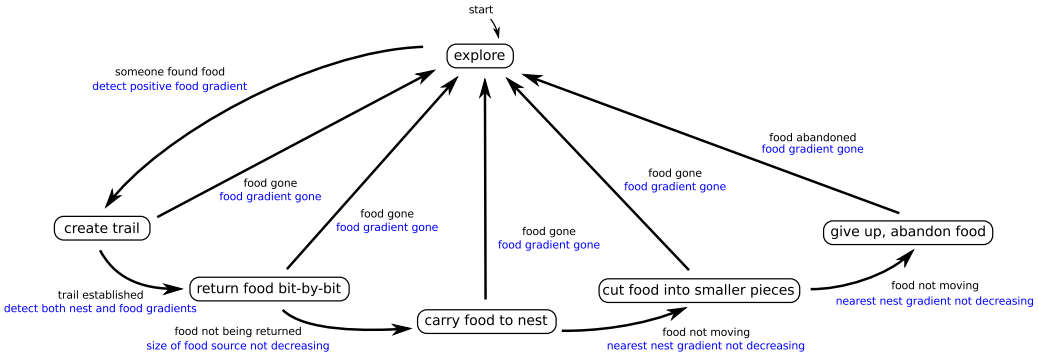
Overall Colony Behavior: In a real colony, foraging of the type described in this dissertation would likely be just one behavior among many that the swarm needed to perform. Figure 4.5a shows a system in which foraging is integrated into an overall colony algorithm which includes other tasks such as cleaning and defense. The "foraging" node in the diagram could contain the entire switching foraging behavior described in this chapter, integrated into a broader framework. Each switch requires a trigger by which each individual robot will decide to make the switch. These triggers are indicated in blue on the diagrams in Figure 4.5. In some of these cases, sensors beyond those included in the standard robot model are required. For example, in order for the swarm to decide that it must forage, the robots need sensors which detect the amount of food the nest has. Depending on the particular hardware implementation of the nest, this could just be a voltage sensor or could be much more complex. To switch to the defense task, the robots must be able to either detect an attack directly or transmit information about such an attack (an "emergency message"). As there often is, there is a trade–off between sensors and communication. Each robot could have a sensor to detect an attack anywhere in the colony, or it could just have a local sensor and use communication to spread the information. These options trade off sensors for communication.

Collective Transport: Figure 4.5b gives an example of collective transport. In this scenario, the swarm must search for food, but it is possible that the food will be too big to return to the nest bit–by–bit, as the previous algorithms assume. The swarm begins by exploring, creating a trail to the food, and trying to return it incrementally. If that doesn't work it tries carrying the food to the nest, if that doesn't work it tries slicing it up into smaller bits that can be carried, and if that doesn't work, it gives up. Again, additional sensors are required to perform the algorithm switches. In this case, the robots must be able to sense if the amount of food in a food pile is decreasing. This is how they know if the current method is working. When the robots are trying to carry the food back to the nest, a method is required for the robots to know if they are making progress toward the nest. Doing this without GPS or high quality optometry is difficult. If the swarm used a nest gradient similar to the one in the `gradient` algorithm, they could detect if they are making progress toward the nest by measuring if the nest gradient values are decreasing. If they are not, the the food is not moving and the robots are not making progress.

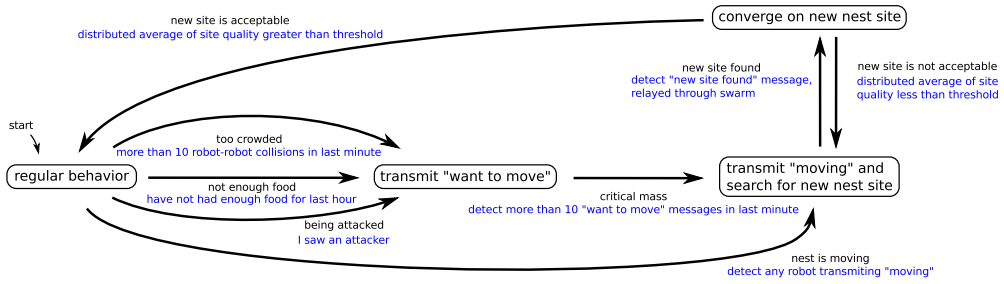
Nest Relocation: A third example is given in Figure 4.5c, describing nest relocation. This occurs when a swarm must decide for some reason that its current



(a) Colony behaviors



(b) Collective transport



(c) Nest site switching

Figure 4.5: Possible algorithm switching diagrams for three different swarm applications. Figure (a) shows an overall colony switching algorithm in which foraging is just one component, figure (b) shows an example for collective transport, and figure (c) shows how a swarm could decide to move its nest to a new location. The black annotations near the arrows describe the switch and the blue annotations are the triggers for each individual robot.

nest is no longer suitable and the whole swarm must move. In this case, it is especially important for the swarm to remain together and make its decisions in a unified manner. In the diagram, three possible reasons are given for why the swarm would want to move nest sites. The current nest could be too small, it could be in an area with insufficient available food, or it could be under dire attack. When any of these is detected by a robot, it transmits a "want to move" message. When enough robots are transmitting this message, one of them detects this and changes its message to "moving". When any robot transmits "moving" the entire colony decides to move. The colony searches for a new nest site, and when one has been found, the robots must assess the quality of the new candidate site. Different applications would lead to different metrics for nest site quality, and it is assumed that each robot can assess some local aspect of this quality. When a candidate nest site is under consideration, each robot would transmit its assessment of the quality. By repeatedly averaging its quality assessment with the quality assessments of the robots nearby, the swarm will reach an estimate of the average quality assessment. This is the swarms' estimation of the quality of the new nest site. If it is above some threshold, the new site is accepted, if not, a new search begins.

4.4.5 Multiple Concurrent Algorithms

In the examples discussed here, every robot in the swarm executes the same algorithm. When the swarm switches algorithms, the entire swarm switches to the new algorithm. A broader and possibly more useful framework would allow different robots to execute different algorithms in a nonetheless coordinated manner. For example, it would probably not make sense for different robots to be running different foraging algorithms in the same place, but it could make sense for some robots to be foraging while others are cleaning. This involves the larger problem of task allocation [36]. Task allocation can be done in a low–communication manner (if a robot sees too much clutter, clean it up, otherwise forage) but that only results in individual decisions, not explicit swarm coordination [3]. Swarm level coordination requires communication, and doing swarm level task allocation would require both more communication and probably more sensors than assumed here. Specifically, in order to make the transitions indicated by the arrows in Figure 4.5, the robots evaluate the triggers (text in blue). In pure algorithm switching, this is enough, but for swarm level task allocation (in which different robots can be doing different tasks), the robots must not only evaluate the trigger but also decide if they as an individual should make the switch. This decision must be made such that there are an appropriate number of each robots on each task, that the robots in the right locations are doing the right tasks, etc. In other words, it requires more communication and sensing.

4.4.6 Relaxing the Connectedness Requirement

Overall, in all of these examples, each individual robot must decide on its own to switch algorithms, but these individual decisions must be coordinated to produce swarm-level behavior. The information that causes each individual to switch must be detected and communicated in a distributed manner (unless it can be directly detected by each robot with its sensors, which is unusual). Distributed sensing and communication is critical, and requires the robots to maintain a connected network of communication links.

The communications model could be further generalized, however, to possibly relax this connectedness requirement. In some situations, it may be possible for the robots to be eventually connected, as opposed to fully connected. In a fully connected situation, there is a series of communication links connecting every robot to every other robot. This is usually what is required. In eventually connected communication, however, every robot will at some point in the future have a series of communication links to each other robot, but not necessarily at the same time. In other words, for every pair of robots, there will eventually be a series of communication links that connects them.

Consider an example in which the robots return periodically to their nest, and while in the nest, they can all communicate with each other. This could be accomplished with special hardware facilitating communication in the nest. Outside the nest, however, they can only communicate locally with neighbors. For some algorithms, it may be sufficient for the robots to save messages that they wish to relay to the swarm until they have returned to the nest. For example, when the swarm needs to relocate its nest site because there is insufficient food in the vicinity, it is probably fine for the communication that mediates this decision to occur only at the nest, and not out in the environment. The eventually connected communications model would work for algorithm switches that do not need to happen in a short amount of time. A local version of eventually connected communication would work, for example, in the collective transport case. When the robots need to switch from returning the food bit–by–bit to carrying it back at once, it is not critical for all robots to make that switch at exactly the same time. It is acceptable if different robots get the message that an algorithm switch is required at slightly different times, and eventually, there are enough of them to start carrying the food.

4.5 Summary

After the previous chapter presented several distributed foraging algorithms which perform best under different food locations, this chapter presented another method in which the swarm as a whole can choose the best algorithm for the given situation. The `gradient` algorithm can find nearby food quickly, and the `sweeper` algorithm can find food further away but takes longer. The `adaptive` algorithm uses the `grad-`

`ient`, `sweeper`, and `randomWalk` methods, detecting in a distributed manner if one has failed and switching to the next. For food in any region of the world, the `adaptive` method is able to choose the most appropriate algorithm. Colony–level algorithm switching requires communication, but can combine benefits of multiple algorithms and improve overall performance.

The central contribution discussed in this chapter is:

- Design of an adaptive foraging method in which the swarm makes colony–level decisions based on distributed information, choosing the algorithm best suited to the given food location. This colony–level decision making requires no additional hardware and only a slight increase to the required local communication capabilities of the robots. Extending this method to switch between other algorithms may require additional sensors depending on the particular triggers.

Chapter 5

Hardware

Algorithms for robot swarms may be developed initially in simulation, but they must eventually prove themselves on real robots. This chapter explores the connections between algorithms and hardware.

First, it explores the tradeoff between hardware quality and swarm performance. Swarm robots depend on communication and sensing capabilities to coordinate and perform their tasks, but when constructing such robots, there is pressure to design these capabilities to be as simple as possible. Simpler, fewer, or lower-quality sensors and actuators will reduce the cost and complexity of the robot, which will reduce the cost for the overall system or allow more robots to be built. However, reducing the quality of the sensors and actuators is expected to reduce the performance of the swarm, so there is a tradeoff. This chapter seeks a design point, compromising between hardware quality and algorithm performance.

To explore this tradeoff, several common swarm primitives are run on a collection of physical robots which match the original hardware model (see section 2.1 on page 12). Three common swarm algorithm primitives are chosen—trail following, expansion, and line formation—and each of them is run both on a small swarm of E-Pucks and in simulation. After measuring their performance with close-to-perfect hardware, the hardware is artificially degraded and the performance is measured to assess the effect of this degradation and search for a design point.

The most complex piece of the hardware model used in this dissertation is the directional range-and-bearing communication. The second part of this chapter investigates the consequences of removing the bearing measurement completely, as opposed to just reducing its resolution. To study the effects of removing the bearing measurement, the `gradient` algorithm is modified for robots without bearing measurement (and also lacking some other capabilities), and then run on a swarm of them. Even given the reduced hardware capabilities, it is still possible to run a modified version of the `gradient` algorithm, but the performance is reduced because of the required modifications to the algorithm.

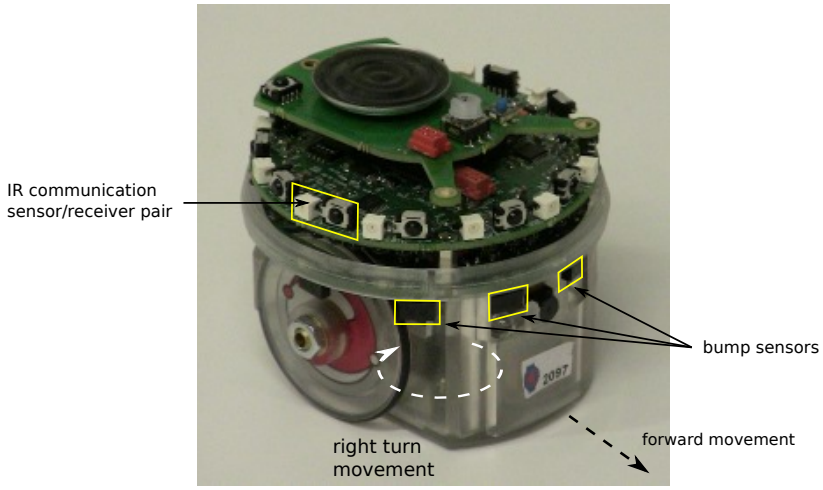


Figure 5.1: An E-Puck robot.

5.1 Hardware Quality vs Algorithm Performance

The quality of the sensors and actuators on a robot will affect the performance of the swarm. It is expected that a swarm of robots with highly accurate motors would perform better than a swarm of robots with noisy ones, for example, but high quality hardware is expensive and sometimes complex. This section will explore several points in the design space, investigating the tradeoff between hardware quality and swarm performance, using E-Pucks (shown in Figure 5.1) as a testbed.

E-Pucks are small research robots with tank steering, an array of bump sensors, range–and–bearing communication between robots, and an easily programmable microprocessor. They also have other capabilities which are not used in this work, such as a speaker, accelerometer, and camera. These robots conform to the hardware model described in section 2.1 on page 12.

The swarm algorithms in the literature tend to be composed of a few common primitive behaviors, such as trail following, expansion, and shape formation [11, 33, 6]. A foraging algorithm, for example, could combine expansion (to find the food) with trail following (to return it to the nest). A flocking algorithm could use shape formation to form and maintain a flock, and homing (a special case of trail following) to move the flock. A coverage algorithm could use expansion on its own, combined with appropriate communication. Instead of measuring the effect of hardware quality on a few specific swarm algorithms, this chapter measures the effect on these three primitives:

- trail following
- expansion
- line formation

These primitives, described in section 5.1.1, are used in the `gradient` and `sweeper` algorithms described earlier.

From experience building small robots, two specific types of hardware capability were chosen for study. The first is the accuracy with which bearing measurement on incoming communications can be measured. Robots need to communicate and often need to measure the range and bearing to their neighbors. Measuring bearing usually requires multiple receivers arrayed around the robot. Using fewer sensors would yield savings in space, complexity, and power usage, but would yield a less accurate bearing measurement. The second type of hardware degradation is movement accuracy. With small robots, the locomotion tends to have a significant random component. Making robots which can accurately move and turn requires tight joint and leg tolerances, accurate motors and precision timing, or expensive feedback sensors which eat up processing time, power, and weight. Noisy open–loop locomotion would allow rough construction and would eliminate the feedback sensors. These two hardware quality variables are described in section 5.1.2.

5.1.1 Task Descriptions

This section describes each of the tasks to be tested along with the metrics by which their performance will be measured. Each of these tasks selected for study—trail following, expansion, and line formation—all have straightforward implementations. Pseudocode for each is shown in Figure 5.3 on page 70.

Trail Following Task

The goal of the trail following task is for a robot to detect a trail marked by other robots, and repeatedly traverse the trail from one endpoint to the other. A set of robots is positioned to create a trail, remaining stationary. The first robot sends a signal indicating it is the start of the trail (perhaps the 'nest' in a foraging task, or the base station in a search–and–rescue task), and the last robot indicates that it is the end (the 'food' or 'victim'). The other robots that make up the trail broadcast integers representing their hop–count from each end of the trail. The trail, along with hop–count annotations, is shown in Figure 5.2a. By listening to the broadcasts from the stationary beacon robots, a 'walker' robot can navigate. When traversing the trail from start to end, it moves toward the beacon with the smallest hop–count leading to the end. The hop–count trails are hard–coded into the robots in these tests, so that the results focus solely on the trail–following ability of the walker. Distributed algorithms for developing the gradient are well known in general, and in this case the one from the `gradient` algorithm described chapter 2 is used.

When the walker robot senses an obstacle or another robot in front of it (the bump sensors can not distinguish obstacles from robots), it simply turns left until it can no longer sense the obstruction, moves forward a short distance, then resumes its

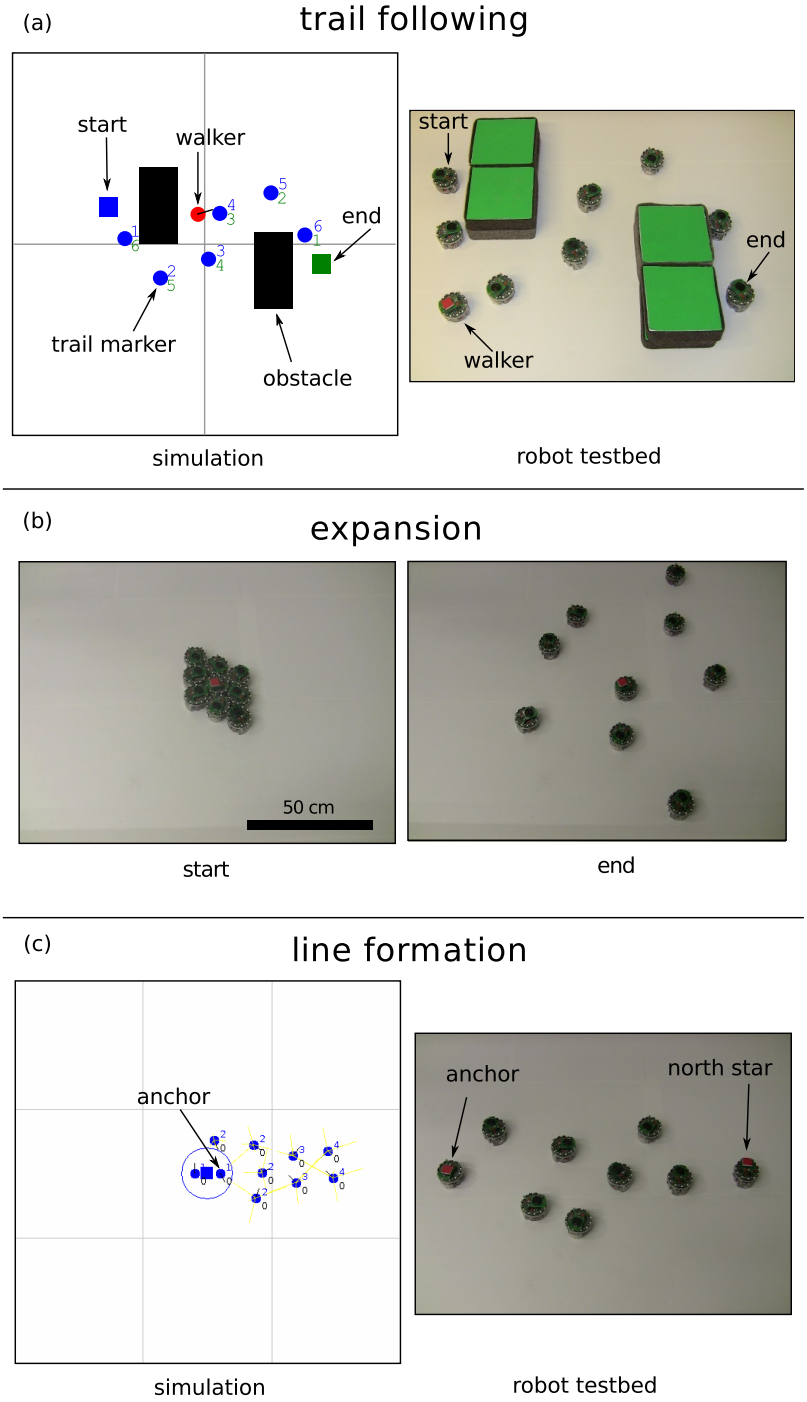


Figure 5.2: Figure (a) shows the trail following task mid–execution. Figure (b) shows start and end configurations in the expansion task. Figure (c) shows the final state of the line formation task in simulation and on the robot testbed.

previous navigation. This results in the walker walking around obstacles to the left. More complex obstacle avoidance methods are possible, but this is simple and works well.

The success metric for the trail following task will be the number of complete traversals the walker is able to make in 10 minutes, capturing both correctness and speed. Once the walker sees the robot marking the end, it turns around and navigates to the start, and repeats. Moving from one side of the trail to the other counts as a single traversal.

Expansion Task

In the expansion task, the swarm must expand to cover as much of the world as possible while maintaining a connected network of robot–to–robot communication. Example starting and final configurations are shown in Figure 5.2b. The expansion algorithm is simple: if a robot can hear more than 3 neighbors, it moves randomly, otherwise it remains stationary. Eventually, the robots will have expanded from the starting point such that each robot has 3 neighbors. If a search target were within this covered area, the swarm would be able to find it, so we are interested in how much area is covered by the swarm, and how fast the swarm reaches this level of coverage.

Line Formation Task

In this task, the robots start in a clump and must form a line. This is done using a virtual–forces algorithm similar to the `sweeper` algorithm. Each robot measures the range and bearing to each of its neighbors, calculates a virtual force as if there were a spring between itself and the neighbor, sums the forces from all robots in its communication range, and moves in that direction. Robots which are close together will be forced apart, and robots too far apart will be pulled together. One robot does not move regardless of the virtual forces acting on it, thus acting as an anchor. Each robot also feels an additional small force which is always directed in the same global direction (arbitrarily called "north"). This could be achieved by placing a compass or sun sensor on the robots. E-Pucks do not have compasses, so in these tests it was achieved by placing a stationary robot at the edge of the field, transmitting a special message at full power, acting as the "north star". All robots placed an additional force on themselves pointing toward the north star robot.

After some amount of time, a line of robots is formed (example shown in Figure 5.2c), reaching from the anchor robot and stretching to the north. We measure how often this virtual–forces method succeeds in creating a line, and when it does, how long it takes the swarm to form the line.

trail following:

```
if outbound
  detect all outbound gradient values in range
  if detect endpoint
    begin inbound, return
  if pointing in direction of smallest value
    if facing obstacle
      turn left until no obstacle, then move forward
    else
      move forward
  else
    turn toward direction of smallest value
else
  (similar procedure to move inbound)
```

expansion:

```
n = number of receptions in comm range
if n > 3
  move randomly, do not transmit
else
  stand still and transmit
```

line formation:

```
detect all robots in comm range
for each reception
  calcualte virtual spring force
vector sum virtual spring forces
add virtual north force
if pointing in the direction of overall force
  if facing obstacle
      turn left until no obstacle, then move forward
  else
      move forward
else
  turn toward direction of overall force
```

Figure 5.3: Pseudocode for each task.

5.1.2 Hardware Variables

Two types of hardware quality are tested: bearing quantization and motor accuracy.

Bearing Quantization

The robots receive signals from other robots and can measure the range and bearing to each transmitter. This bearing measurement would likely be achieved by placing a ring of receivers around the robot. When a transmission comes in, it would probably be received by multiple sensors. One could interpolate between the received intensities at each receiver to calculate a continuous bearing to the receiver. The E-Pucks calculate a continuous bearing measurement in this manner (with 12 sensors arranged around the robot). This requires high quality receivers and significant computation to do the interpolation. If there were only 8 sensors, bearing would be obtained by simply knowing which sensor received the signal, and would only be known to an accuracy of 45°. In other words, there would only be 8 quantized possibilities for the bearing measurement. This would be a significantly simpler design, requiring fewer sensors, less wiring, less power usage and weight, and fewer possibilities for failure.

Four possibilities for this bearing quantization are tested: none (continuous bearing measurement), 8 sensors, 4 sensors, and 2 sensors. With only two sensors, the robot only knows if the transmitter is in front of or behind it. These possibilities for bearing quantization are diagrammed in Figure 5.4. To achieve this on the actual robots, sensors are artificially degraded in software on–board.

Motor Accuracy

The model assumes that control of motion is implemented by velocity control of two motors. Uniformly distributed noise is artificially added to the intended velocity to measure the effect of motor quality. Four amounts of noise are tested: $\pm 0\%$, $\pm 5\%$, $\pm 20\%$, and $\pm 100\%$. With $\pm 100\%$ noise, each wheel could rotate at a speed anywhere between 0 and twice the commanded rate. Notice that because the noise of each wheel is independent, this also means that when a robot intends to go straight, it could actually veer off course.

An illustration is provided in Figure 5.5. As a simple illustration of this noise model, a robot in simulation is commanded to walk in a large circle ten times. This is repeated with various amounts of motor error. With $\pm 1\%$, the circles drift slowly, and with $\pm 100\%$, it resembles random walk.

Building robots with small movement error is difficult because it requires accurate actuators, feedback sensors (which require mass, power, and computation), or tight tolerances on construction (precise wheel diameter, leg length, power regulation). An

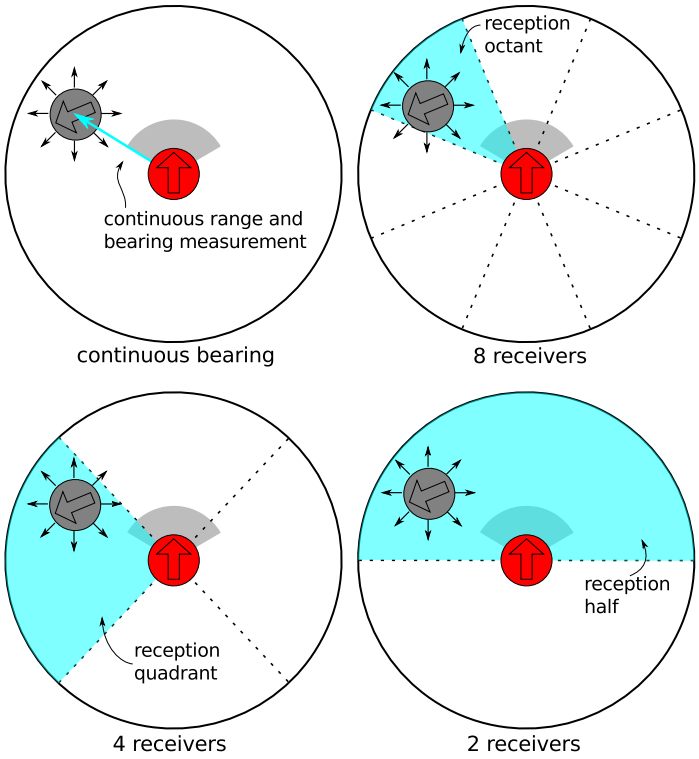


Figure 5.4: With continuous bearing measurement, the receiver can calculate the bearing to the transmitter (to within the error of the sensors). With quantization of 4, for example, the receiver only knows which quadrant the transmitter is in. This is done by reducing (in software) the number of IR sensor/receiver pairs on the communication ring (see Figure 5.1).

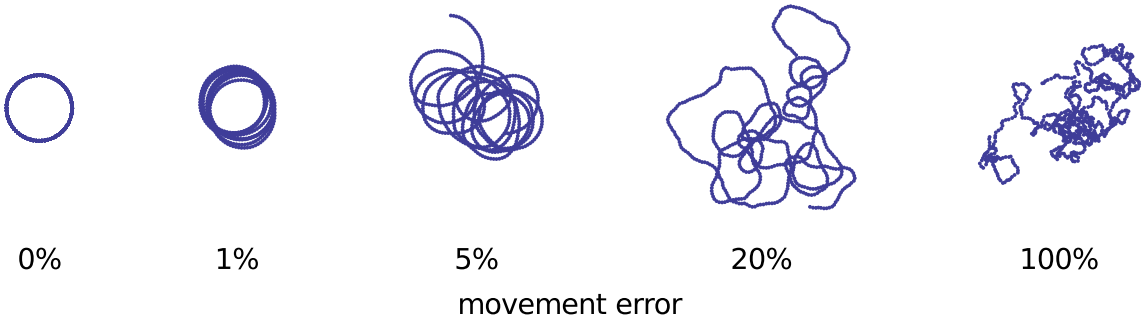


Figure 5.5: In simulation, a single robot is commanded to move in a 2m–diameter circle ten times, with varying amounts of movement error. With ±20% error, there is still approximate looping behavior, and with ±100% error, it looks like completely random movement.

algorithm which can perform well on robots with poor locomotion will be more useful because the robots will be easier to build, less costly, and simpler.

Error of ±0% is possible to achieve in the simulator, but the physical robots will, of course, have some minimum noise. Informal measurements indicate that this error is less than 1% for the motors on the E-Pucks. Strictly, movement error is added to the error already present in the system, which for the simulator is 0% to within the accuracy of mathematical operations on a Java® `double`, and for the robots is less than 1%.

5.1.3 Tests and Results

This analysis measures the effect that bearing quantization and motor accuracy have on the metrics described above: number of complete path traversals, amount of area coverage, speed of coverage, line formation success rate, and speed of line formation. Each test is done both in simulation and on physical robots. The data is shown in Figures 5.6, 5.7, 5.8.

The parameters of the simulation are set to match the physical robots. Robot size, movement speed, communication radius, and obstacle sensing range are all matched between simulation and hardware. Dropped communications do occur in the real robots but are not modeled in the simulator, sometimes causing a difference between the robot performance and the simulator performance. This effect is discussed in detail after the data is presented.

Trail Following Experiment

Number of Complete Path Traversals With no bearing quantization, the robots appear to make 3 to 4 traversals in 10 minutes (Figure 5.6a). With only 8 or even

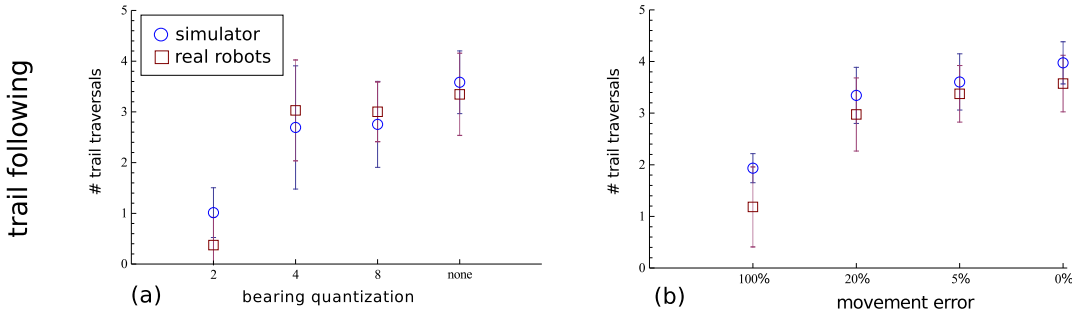


Figure 5.6: Trail following performance. Simulation data points represent the average of 100 runs, physical robot data points represent the average of 5 runs, and error bars show one standard deviation. On the horizontal axes, hardware performance increases to the right. Bearing quantization must decrease to 2 until an appreciable decrease in performance is observed. Similarly, movement error must reach ±100% before performance decreases.

4 sensing regions, the performance does not substantially fall. There is a substantial drop with only 2 sensors.

Movement error shows a similar trend. ±5% or even ±20% show no substantial drop in performance from ±0%. Only with ±100% error does the performance fall. For both bearing quantization and movement error, the simulation and physical robots have comparable performance.

Expansion Experiments

Total Area Coverage Neither bearing quantization nor movement error have an effect on area coverage, as seen in Figures 5.7a and 5.7b. This makes sense because the bearing to a transmitter has no effect on robot movement during this procedure; the only quantity that matters is the number of other robots in the sensing range. The robots simply move randomly when they can hear more than 3 other robots and stand still when they can hear 3 or fewer, regardless of where those other robots are. It also makes sense that the movement error has no effect because this metric measures the area covered when the expansion is finished, regardless of how long it took. So, with ±100% error, one might expect that coverage takes longer, but it still covers the same area in the end.

In the case of bearing quantization, the physical robot performance is noticeably worse than the simulation performance. This is likely because of intermittent communications failures. In reality, the edge of the communication radius is not sharply defined. Sometimes communications are just lost, and sometimes they travel farther than normal. This can cause robots to wander too far away and get lost, and therefore not be counted.

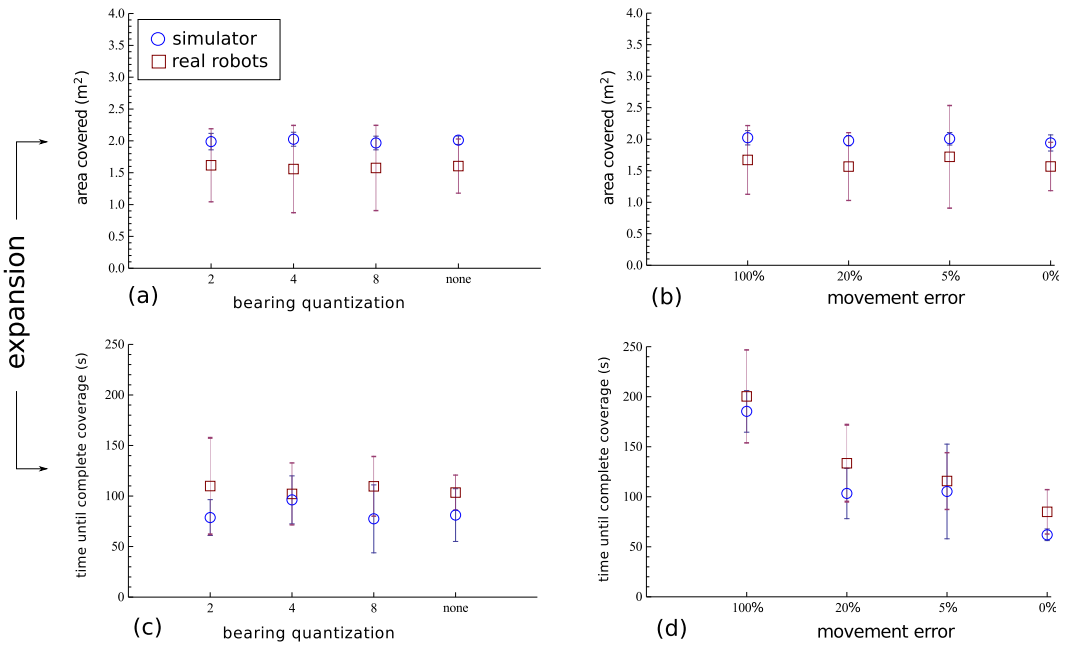


Figure 5.7: Expansion performance. Simulation data points represent the average of 100 runs, physical robot data points represent the average of 5 runs, and error bars show one standard deviation. Simulation data points On the horizontal axes, hardware performance increases to the right. Bearing quantization has no effect on the total area covered nor the time it takes to reach coverage. Movement error does not affect total area coverage either, and must reach ±100% before it degrades the time until complete coverage.

Time Until Coverage As expected, with ±100% error the swarm takes longer to reach its final coverage than with ±0% (Figure 5.7d). ±5% or ±20% show mostly non-degraded performance. Bearing quantization has no effect (Figure 5.7c), which makes sense for the same reason as above — the location of the receptions is not used in the expansion algorithm. The same effects of communications failures can also be seen here, with the results from the robots being consistently worse than the results from the simulation.

Line Formation Experiments

Line Formation Success Rate In simulation, the robot swarm is almost always able to form the line, as seen in Figures 5.8a and 5.8b. With physical robots, however, failures are more common. Because of dropped communications, some of the virtual forces can be temporarily lost causing robots to move in the wrong directions, or even get completely lost. With only two reception directions (Figure 5.8a), the situation is even worse. In this case, the robots often fail, partly because of communications

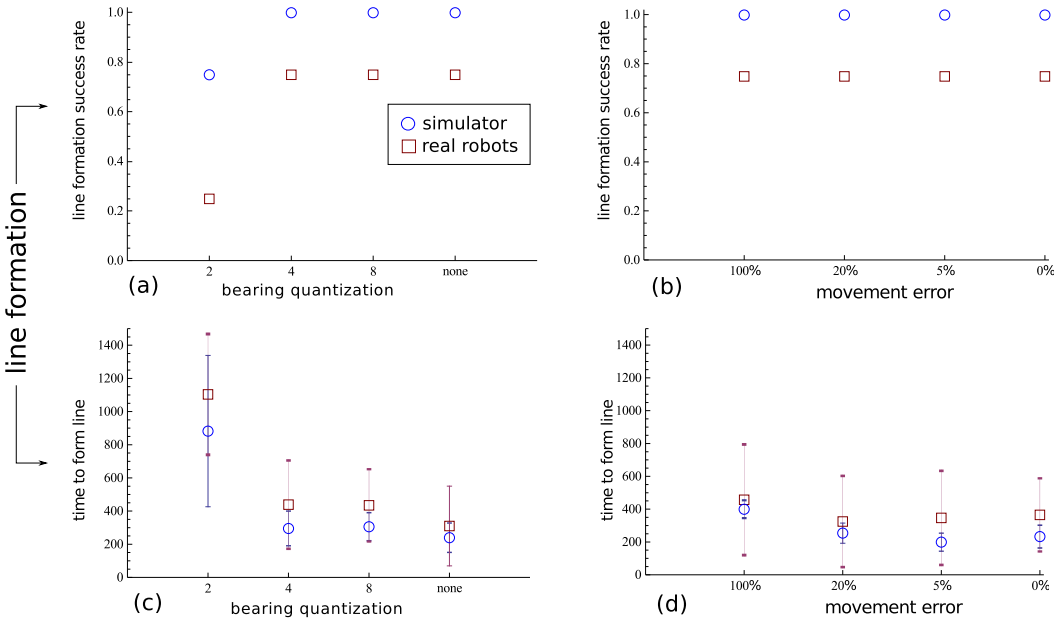


Figure 5.8: Line formation performance. Simulation data points represent the average of 10 runs, physical robot data points represent the average of 5 runs, and error bars show one standard deviation. On the horizontal axes, hardware performance increases to the right. The swarm is capable of forming the line most of the time, except under a bearing quantization of 2, when it also takes much longer to form the line.

failures, but partly because they don't have enough information on which to calculate their movements. The virtual force is always pointing either directly ahead or directly behind. If it's directly behind, they will turn until it is directly ahead (not necessarily a full 180° turn) and move. With so little information, the communications failures overwhelm them.

Line Formation Time The time required for the swarm to form a line (when it does so successfully) is roughly the same with no bearing quantization as with a quantization level or 8 or 4. When only two sensors are present (bearing quantization of 2) it takes twice as long or more, as seen in Figure 5.8c. Surprisingly, the amount of movement error seems to have no significant effect on the line formation time. Even with ±100% error, it takes approximately as long as with no error. During the process of forming the line, the robots 'jostle' around as they are moved by the virtual forces. Most of their movement appears to be this jostle, and over time they drift to the correct final positions. Apparently, adding even ±100% error just adds to the jostle, but the drift rate is unaffected and robots are still able to reach the final positions in roughly the same time.

Differences Between Simulator and Physical Robot Performance

In several cases, the performance of the physical robots was consistently worse than the simulator model. This is particularly evident in the area covered in the expansion task (top of Figure 5.7) and the line formation success rate in the line formation task (top of Figure 5.8). These differences are likely caused by communication failures in the robots, which are not modeled in the simulator.

The robots sometimes drop communications. Even with just two robots communicating with each other, communications will occasionally be dropped. The rate of communication failure increases when more robots are overlapping. A simple TDMA scheme was developed for the robots to mitigate these failures, but this did not completely eliminate the failures. Communication failures are especially problematic for the line formation task, where they are interpreted as a loss of the virtual force between two robots. If two robots are at the edge of their communication ranges and briefly lose communication, they could move further away under the influence of other virtual forces, possibly causing the line of robots to snap. This is the likely explanation for the difference in performance in the line formation task.

The second communication failure mode which is not modeled in the simulator is that robots themselves can block communications. If one robot is positioned between two others, for example, the two outer robots will have a high failure rate in their communication. This is especially problematic for the expansion task, which depends on robots being able to count their neighbors. A robot could calculate that it has fewer neighbors than it really has because some of them are hidden behind other robots. Underestimating the number of neighbors could cause a robot to become a beacon when it should in fact remain a walker. This causes the density of the beacon field to be too high, which reduces the amount of area the swarm can cover.

Communication Strategies

The robots in a multi-robot swarm need to communicate, but this communication could be implemented in multiple ways. It need not be synchronous, it needs to gracefully handle transmission losses, and it must scale well with the size of the swarm.

For the experiments with the E-Pucks, the communication used TDMA. This requires an external synchronization pulse, which was provided for these experiments. TDMA is not a good solution for large swarms because it does not scale well. For large swarms, a randomized backoff protocol would be more appropriate. In such a communication scheme, each robot will first sample the communication channel to determine if it is safe to transmit. If so, it transmits, and if not, it waits a random amount of time before trying again. This solution scales easily to large numbers of robots, but sacrifices communication reliability. It could happen, for example, that there are so many robots wanting to transmit that one robot keeps delaying until its

message is so old that it is deleted.

The communication unreliability of the random backoff can be partially mitigated by logging previous communications. A robot would log every communication it received in the last one second, for example. If a robot is receiving communications from another robot somewhat regularly and the stream of messages is briefly interrupted, the receiving robot could assume that the transmitter is still present by examining the log. This would be useful in the `sweeper` algorithm, for example. If communication is briefly lost, or the communications get delayed because of repeated random backoffs, two robots could deduce from their history that the other is still present, and still calculate a virtual force between themselves.

Asynchronous communication using a random backoff system with logging is a good solution for robot swarms. Unfortunately, the logic governing the use of the log is algorithm dependent. `sweeper` would use the log to reconstruct lost transmissions differently from the way `gradient` would use it. Perhaps a sophisticated communication layer could be written between the low-level communication protocol and the algorithm to abstract these difficulties away.

5.1.4 Results Summary

One goal of this section is to answer the hypothetical question from an engineer building a robot swarm: "how good do the robots need to be?". The data provide a partial answer. For the trail following task, it appears that building a communication system which can only distinguish four possible reception directions and a movement system with ±20% error is good enough, meaning that it will achieve performance comparable to a robot with continuous bearing resolution. ±20% seems to also be sufficient for the expansion task. Lastly, for the line formation task, quantization of 4 offers similar performance to continuous resolution, and even ±100% movement error is acceptable.

Overall, robotic hardware offering ±20% movement error and bearing quantization of 4 yield mostly un-degraded performance, suggesting those are good design points for swarm robots. Slight gains may be achieved by selecting ±0% over ±20% movement error, or from selecting a continuous resolution over a bearing quantization of 4, but these are only worth paying for if the cost of those higher–quality sensors and actuators is commensurate with the small gain.

5.2 Robots without Bearing Measurement

The most complex piece of hardware assumed in the hardware model in section 2.1 is the range–and–bearing communication. Range–only communication would be much simpler, but the algorithms were developed assuming bearing would be available. In the analysis above, the number of bearing sensors was reduced to two, but this

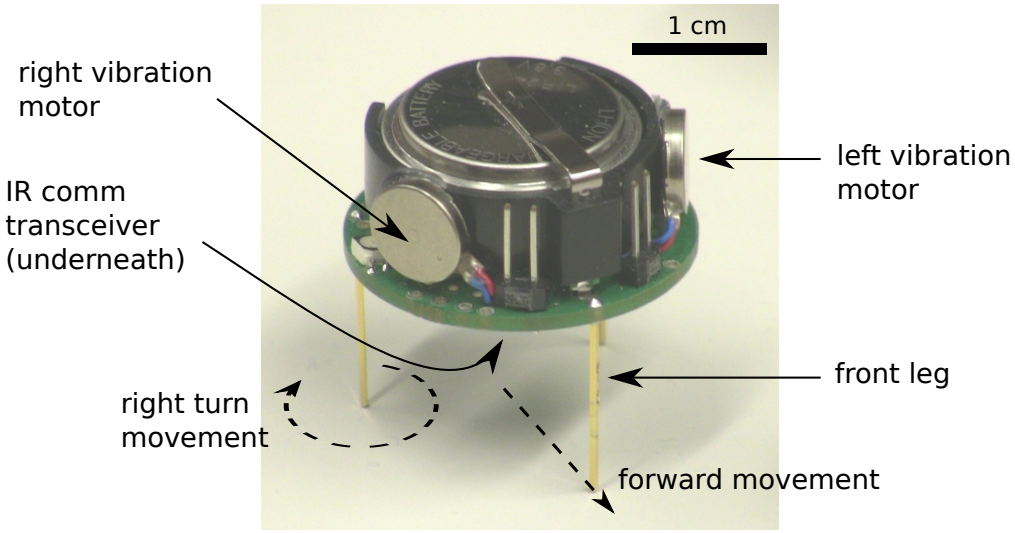


Figure 5.9: A kilobot.

section explores the consequences of removing bearing measurement completely and using range–only communication. This section introduces a robot that has range–only communication, describes an adaptation of the `gradient` algorithm for such robots, and presents results from an implementation.

5.2.1 Kilobot description

The kilobot project, lead by Michael Rubenstein, aims to design and build a robot which is simple and inexpensive enough to be produced in large numbers (hundreds to thousands) but has rudimentary sensing, processing, communication, and motion capabilities. The robot, called "kilobot", is shown in Figure 5.9. Kilobots have a diameter of roughly 2.5cm and stand roughly 2cm tall. They have three legs and move using two vibrating motors. By differentially activating the two vibrating motors, the robot is capable of moving straight or turning left or right. Using a single IR transmitter/receiver pair, the robots are able to communicate with and measure distance to their neighbors. They are controlled with a basic onboard microprocessor. See [50] for more information on the kilobot design, usage, and sample algorithm implementations.

There are several important differences between the kilobot hardware and the hardware model used to develop the algorithms in this dissertation:

- The kilobots do not have bearing sensors. When receiving a communication from a neighbor, they can measure the distance to the neighbor, but not the bearing.

- The kilobots do not have bump sensors. There is no direct way for them to

know if they are bumping into an obstacle or another robot.

- The ratio of communication radius to body size is smaller for the kilobots. The communication radius to body length ratio is about 6 for the kilobots, but was assumed to be 10 in the parameters described earlier (section 3.1 on page 36).
- The kilobots can turn left and right, but they do not turn around their center. Instead, they rotate around one of their back legs. When the kilobot turns right, for example, its right leg remains stationary and the robot rotates around it, as shown in Figure 5.9.

The most significant of these differences is the first one, the lack of bearing information.

5.2.2 Gradient Algorithm Adaptation

The `gradient` algorithm had to be modified from its form as presented in chapter 2 because the kilobots do not have bearing sensors. They can receive communications from their neighbors and can measure how far each communicating neighbor is, but can not measure the bearing. The critical capability that the bearing information provided to the `gradient` algorithm was the ability to navigate to any transmitting robot (usually the robot with the lowest gradient value) in its communication range. In order to replicate that capability without a bearing measurement, a modified orbiting behavior was used.

In the orbiting behavior, one robot stands still and transmits. Another robot, the orbiting robot, receives the transmission and measures the distance to the transmitter. The orbiting robot is moving forward, but does not know if it is moving toward the transmitter, away from it, or orbiting it (such that the distance would not change). The orbiting robot continues measuring the distance to the transmitter as it moves, and calculates whether this distance is increasing or decreasing. If, over time, the distance is increasing, the orbiting robot bears right, if the distance is decreasing it bears left, and if the distance is not changing, it moves forward. This causes the orbiting robot to orbit the transmitting robot in a clockwise direction.

This orbiting behavior can be used as a method of following a gradient. In a field of beacons, all of which are stationary and transmitting their gradient values, a walker robot could begin by orbiting the robot with the lowest gradient value it heard. As the walker moved, it would come into the communication range of a robot with a lower gradient value, then begin orbiting that robot. The diagram in Figure 5.10 shows an example of how this would be used to follow a gradient.

Another significant difference in the hardware capabilities of the kilobots is that they do not have bump sensors. If there were an obstacle in the world, there would be no way for a kilobot to sense it. If the obstacle prevented the robot from moving, it would have no direct way to know that it was being stopped. The `gradient` algorithm

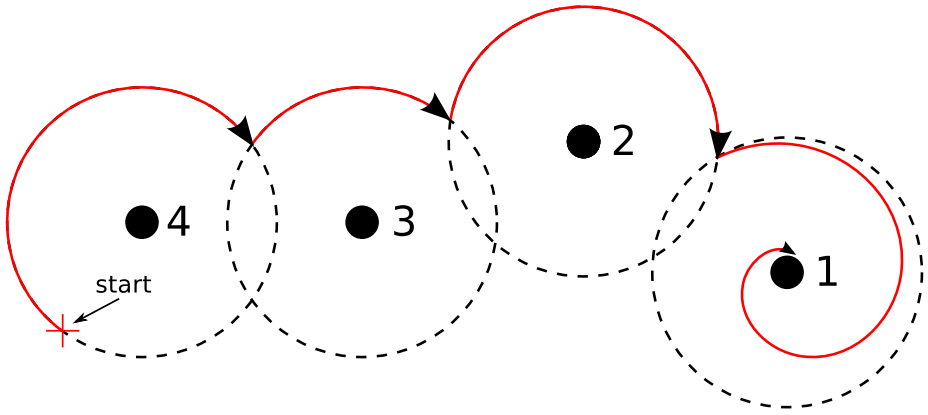


Figure 5.10: A robot starting at the start point could use the successive orbiting behavior, possibly combined with a spiral at the end, to reach its destination. The dotted circles are not communication radii, they are simply drawn at the radius the robot happens to start at relative to the first beacon it sees. The numbers next to each beacon indicate possible gradient values that the walker is attempting to follow.

requires a bump sensor, however, so a virtual bump sensor was developed by defining a virtual bump radius around each robot. In the kilobot tests, the only obstacles were other robots, so if a robot detected another robot within the bump radius, and that other robot was getting closer over time, the virtual bump sensor would trigger. It is not sufficient to simply trigger a bump every time another robot is detected inside the bump radius because the potential obstacle could be behind the robot. This virtual bump sensor requires keeping track of the distances of all robots within the communication range and recording that information over time, so that if any of them enter the virtual bump radius, it is known whether they are getting closer (and so probably in front) or not (probably behind). Note that this bump sensor is not perfect and can misfire, registering a virtual bump when the robot is actually not in danger of bumping. This could occur, for example, when a robot sees another robot approaching slowly from the side.

With these two modifications—using the orbiting behavior to track gradients and using the virtual bump sensor to avoid collisions—the `gradient` algorithm could be implemented on the kilobots. These modifications have a negative impact on the performance of the swarm, and to assess this impact, the modified gradient algorithm is compared against the "real" `gradient` algorithm in simulation with parameters changed to reflect the dimensions of the kilobots.

While it is possible to adapt the `gradient` algorithm for use on the kilobots, it is not possible to adapt the `sweeper` algorithm. Both algorithms require bearing information, but they use the bearing information in different ways. `gradient` needs bearing information in order to navigate to a desired beacon in order to follow the gradient trail. If there were another way to follow the gradient trail, `gradient` could

operate without a bearing sensor. This is exactly what the adaptation does; it substitutes a different method for following the gradient, eliminating the need for bearing sensors. `sweeper`, on the other hand, uses bearing information to perform vector addition of virtual forces in order to calculate the virtual force on itself. This requires knowing the bearing to the neighbors and there is no easy substitute, so `sweeper` can not be implemented on robots without bearing sensors[1].

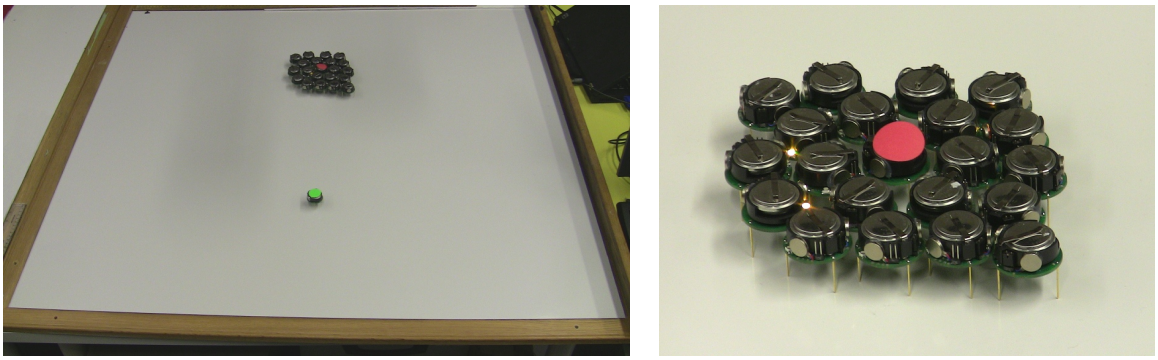
5.2.3 Results

Figure 5.11 shows the test setup. A collection of kilobots are clustered around the nest marker (shown in red), and the food marker (green) is placed a specific distance away. Figure 5.12 shows snapshots of the swarm of kilobots as they run the modified `gradient` algorithm. The swarm follows a similar pattern to the previous `gradient` algorithm—they expand, find the food, and return it bit–by–bit. One important behavior difference is that the walker robots usually do not move between two beacons, instead they go around the perimeter. This behavior is shown in Figure 5.13, which shows the beacon field of a sample run with the path of a single walker superimposed. Because of the lack of a forward facing bump sensor, the robots must use a virtual bump sensor which is less precise and sometimes triggers when it shouldn't. In order for a walker to be able to walk between two beacons, it must be very close to the centerline dividing the beacons so that it doesn't trigger a bump from either of them. This is rare, so usually the bump sensor triggers causing the walker to avoid both beacons and go around.

The test conditions used to evaluate the performance of the modified `gradient` algorithm on the kilobots are slightly different from those used earlier, because of the reduced hardware capabilities. The communication radius of the kilobots is approximately 10cm. To test the effect of the nest–food separation, the food is placed between 1 and 3 communication radii away from the nest and the amount of food that the robots return to the nest in 20 minutes is measured. These results are compared against the `gradient` algorithm in simulation, with the robot size and communication radius changed to match the parameters of the kilobots.

Data from these experiments is shown in Figure 5.14. When the food is very close, the kilobots always find it, as expected. As the food gets further away, the success rate goes down, until the food is 3 communication radii from the nest, at which point the swarm can never find it. The real `gradient` algorithm has a similar trend. The

[1]It is actually possible to estimate the bearing to a neighbor without using bearing sensors. It can be done by taking multiple measurements of the distance at different locations, assuming no other robots are moving, and trigonometrically calculating the relative location to the target. This requires stationary neighbors, accurate moving and turning, and time to complete the calculations. It is too complex, brittle, and inaccurate to be useful in this situation. In another scheme, a global coordinate system could be theoretically established from distance–only measurements if all robots were participating. This is a topic of current research.



(a) (b)

Figure 5.11: The starting configuration for the kilobot tests. The test area, shown on the left, is approximately 1.3m^2. The detail on the right shows the cluster of robots before the run begins. The robot marked in red marks the nest, and the one in green marks the food.

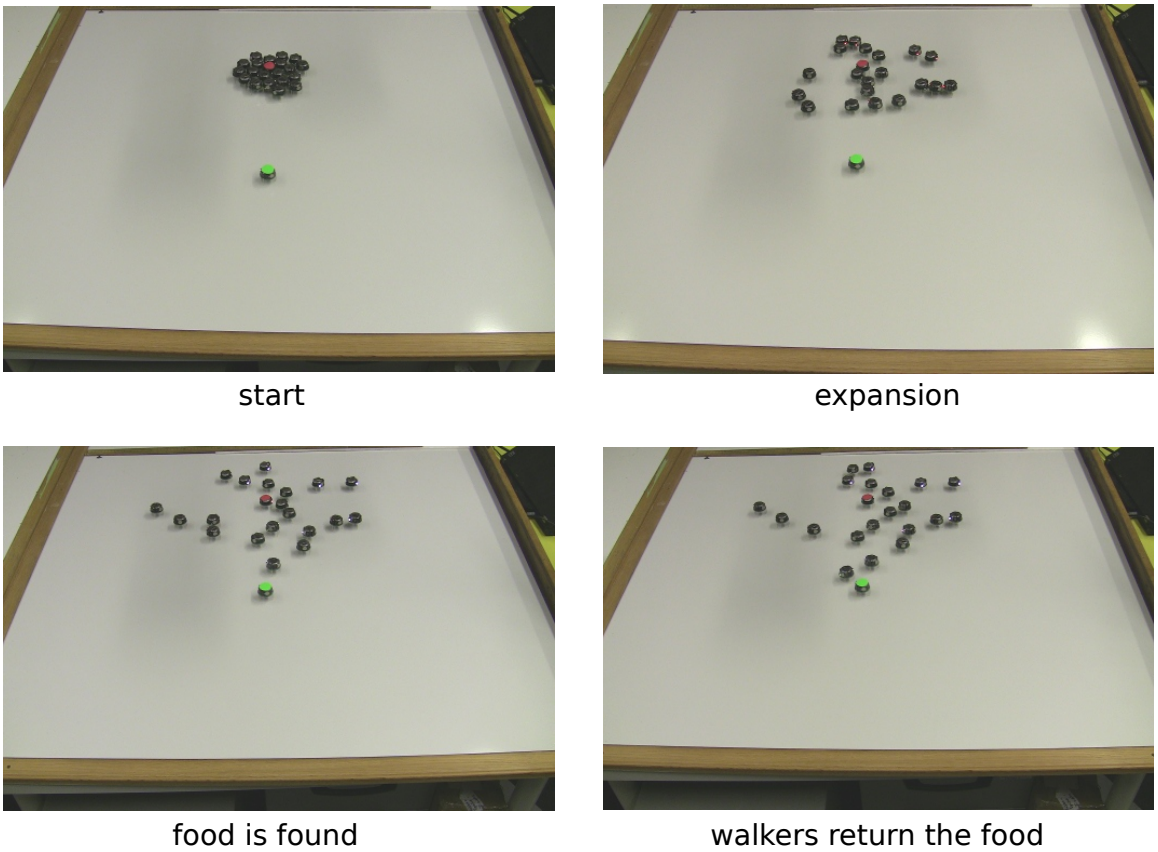


Figure 5.12: Several snapshots of the modified `gradient` algorithm running on the kilobots. The path that walkers take when returning food can not be seen in the lower right image, but is shown in Figure 5.13.

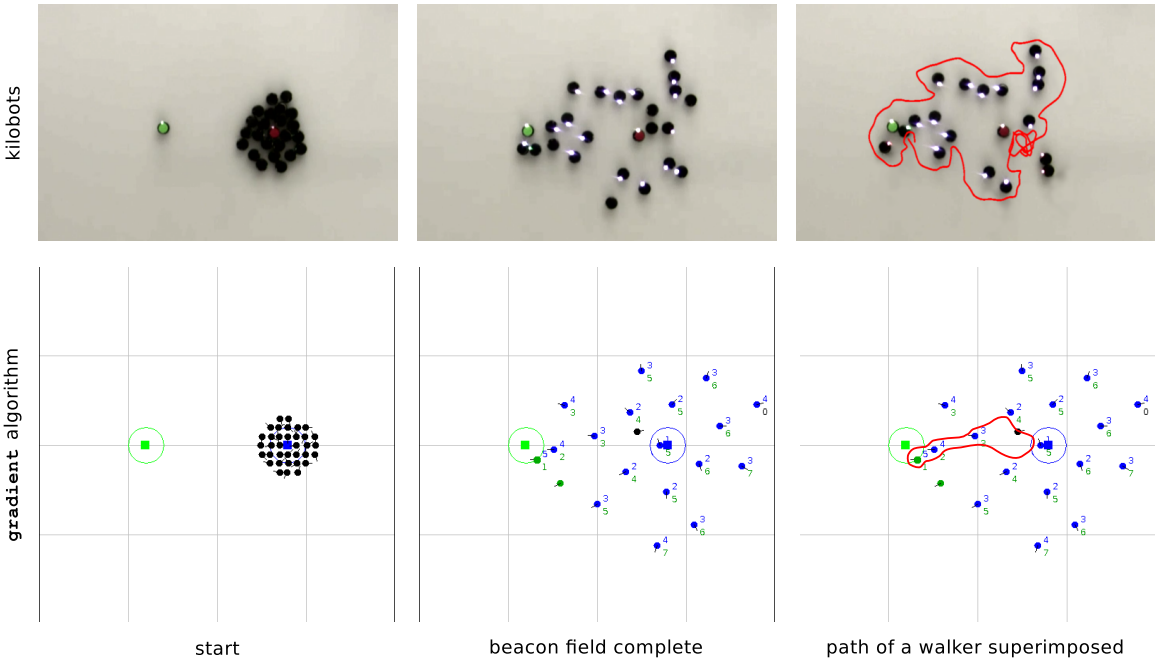
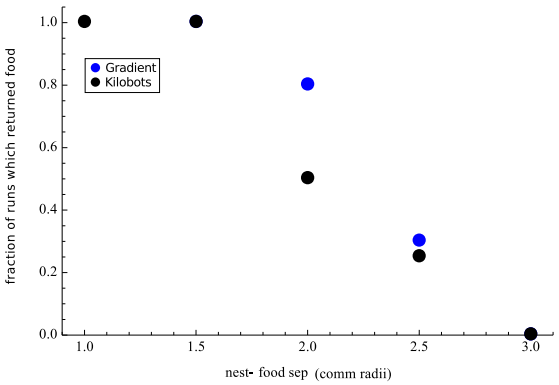
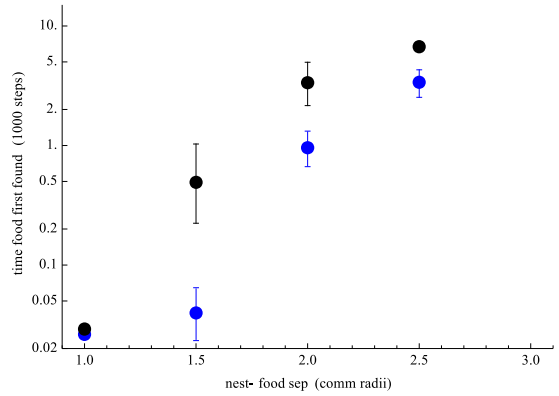


Figure 5.13: The middle images shows beacon fields. In the images on the right, all walker robots have been digitally removed from the image except one, and the path of that single walker robot has been superimposed. This shows the preference of the kilobot walkers to go around the beacons rather than threading through the field. For comparison, the `gradient` algorithm is able to cut a more straight path because of its better bump sensors.

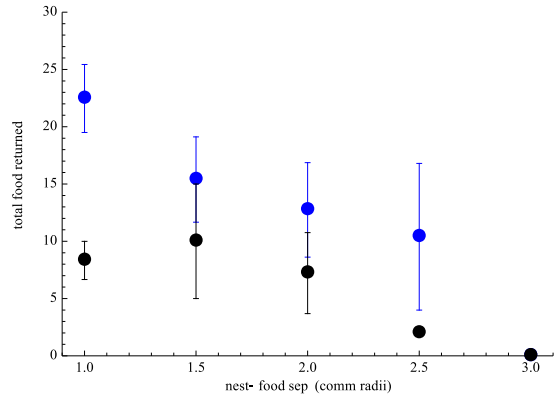


comments

Kilobots and the gradient algorithm show similar trends



Kilobots are always slower than the gradient algorithm in simulation.



Again, similar trends are seen. In the real kilobots, there is a performance penalty when the food is very near the nest because there are so many robots packed around that the walkers spend most of their time doing obstacle avoidance. Because of the real bump sensor in the gradient algorithm, the robots can do better obstacle avoidance.

Figure 5.14: Plots showing results from the kilobots (no bearing, virtual bump sensor), compared against the `gradient` algorithm (in simulation with full bearing and bump sensing). Each kilobot point represents 4 runs and each `gradient` point represents 30 runs. To compare the times at which the food was first found, a rough equivalence of 1 second $\approx$ 4 simulator time steps was used.

kilobots also exhibit the expected trend when measuring the time the food was first found by the swarm. When the food is within a communication radius of the nest, it is found almost immediately. As the food gets further away, it takes much longer. Lastly, the total amount of food returned also shows the expected trend with one exception. When the food is very close to the nest, the kilobots have slightly reduced performance. This is because of congestion, combined with the poor virtual bump sensor. With so many robots, the walkers spend nearly all of their time avoiding each other. This problem is aggravated further by the fact that the virtual bump sensor triggers too often, especially with so many robots. The `gradient` algorithm would also suffer a bit from the congestion problem when the food is so close, but because of the real bump sensor and better homing capabilities (using real bearing measurements), the robots can more accurately avoid their neighbors and traverse the very short path.

In each case, the kilobot gradient algorithm underperforms the real `gradient` algorithm in the simulation. This can be interpreted as a rough estimate of the performance impact of excluding bearing information and a bump sensor from the sensor array. The lack of these two sensors forced modifications to the algorithm which reduce performance. The reduction is significant. Without these sensors, the robots return roughly half of the food and take longer to find it.

5.3 Summary

This chapter explored the connection between algorithms and hardware. First, the relationship between hardware quality and algorithm performance was studied. On a swarm of E-Pucks running common swarm primitives, the quality of the communication and locomotion was degraded and the performance was measured. Reductions in hardware quality lead to reductions in swarm performance, but only when the hardware quality reduction is severe. Next, a modified `gradient` algorithm was run on a swarm of kilobots. Kilobots do not have the capability to measure bearing on incoming communications, nor do they have bump sensors. The reduced hardware capabilities of the kilobots led to reduced performance of the swarm, but it was still able to function decently.

The relationship between hardware and swarm performance is important to understand as robotics moves toward swarms of smaller and more hardware–constrained robots. Eventually, the aim is to implement swarm algorithms on very small robots [1] on which sensor and actuator precision is quite expensive. Algorithms which can work on robots with low quality sensors and actuators will be more widely useful.

The contributions discussed in this chapter are:

- A reduction in sensor and actuator capability, at least for the sensors, actuators, and algorithms tested here, causes a reduction in performance. However,

this reduction is only observed when the hardware quality has been severely degraded.

- The results suggest that robotic hardware offering ±20% movement error and bearing quantization of 4 is a good design point. The hardware is far from perfect which makes construction and maintenance easier and less expensive, but the algorithm performance is not strongly degraded.
- The `gradient` algorithm is not tied to the robot model for which it was developed; it can be adapted for the different hardware capabilities of the kilobots, and runs successfully.

Chapter 6

Conclusion

This dissertation is concerned with distributed multi–robot foraging in the simple robotic hardware regime, and the connections between the hardware capabilities of the robots and the swarm algorithms that they run. This chapter will briefly summarize what has been presented and will provide some thoughts on possible directions for future work.

6.1 Summary

Multi–robot swarm systems can be designed using a variety of types of robots, and this dissertation is concerned with robots with simple hardware capabilities. The robots are assumed not to have global positioning and communication capabilities, and instead must rely on local communication and sensing in order to coordinate. The robots communicate using local range–and–bearing communication, although the consequences of removing the bearing measurement are also explored. The algorithmic focus is on distributed foraging algorithms for swarms of simple robots.

This dissertation can be roughly broken down into three parts:

- Chapters 2 and 3 developed and analyzed three distributed multi–robot foraging algorithms, called the `VP`, `gradient`, and `sweeper` algorithms. `VP` is inspired fairly directly from the foraging behavior of ants, with the exception that robots themselves act as pheromone locations ("beacons") and these virtual pheromones are laid and detected through direct local robot–robot communication. This change was necessary because the robots can not deposit a chemical pheromone. The `gradient` algorithm is a similar but purified version of the `VP` algorithm, replacing real–valued decaying virtual pheromones with integer–valued gradient levels. This is a simpler algorithm and it generally performed better than `VP`. Finally, the `sweeper` algorithm uses the range–and–bearing communication to create virtual spring–like forces to form a line of robots. This

line forms a search front and sweeps the world like the hand of a clock searching for food. If food is found, it returns the food to the nest along the search front.

- Analysis in chapters 2 and 3 showed that the `gradient` and `sweeper` algorithms each worked best with different nest–food locations, which led to chapter 4, in which a method is devised to allow the swarm as a whole to recognize which algorithm would work best for the given situation and switch to it. The resulting `adaptive` algorithm outperformed the others because it was able to choose the most appropriate algorithm for the given nest–food separation.

- Chapter 5 explores hardware implications from two perspectives. First, it explored the design tradeoff between hardware quality and algorithm performance. Lower quality hardware will make the robots less expensive and easier to construct, but may reduce the performance of the swarm. To measure this tradeoff, several common swarm algorithm primitives were run a collection of robots and their movement precision and bearing measurement precision were reduced in several steps. At most tasks, the swarm can function well with significantly reduced hardware quality. The range–and–bearing communication is the most restrictive part of the hardware model used in this dissertation, and the next part of the chapter explores the consequences of removing the bearing measurement completely. It discusses results from implementing the `gradient` algorithm on a swarm of robots with no bearing measurement. These robots have less hardware capability than the `gradient` algorithm was designed for, which required modifications to the algorithm and penalized the performance. The `sweeper` algorithm could not be adapted for these robots at all because of their lack of a bearing sensor, which is critical for that algorithm.

6.2 Future Work

There are many interesting possible extensions of the work presented here, as well as applications to other areas of swarm robotics.

One obvious extension of the `gradient` algorithm would be to develop a method by which useless beacons can be reclaimed. If the food is near the edge of the expansion radius of the swarm, then there will only be a few robots left as walkers to return the food and most of the robots will be beacons. Most of these beacon robots, however, will be useless. They will be far away from both the food and the nest and no walker robots will ever use them. It would be useful if these beacons could realize that they are useless and become walkers again. There are two basic problems: beacons need a way to know if they are useless, and if they are useless they need to know if it is safe for them to walk away without leaving other beacons stranded. It would be best to solve each of these problems without adding additional communication. A simple algorithm can be designed for beacons to detect if they are at the edge of

the field[1], but this alone is not enough for the beacon to decide that it is useless because a walker could still be using the beacon to navigate. Solving these problems with minimal additional communication is a good area for future work, and would probably increase performance in some situations.

An interesting area of future work for the algorithm switching behavior would be to implement the framework in a more general setting involving dynamic environments. The swarm tries algorithms A, B, and C in sequence, and after it has decided that A is not the best algorithm, it does not return. In a dynamic world, though, food could appear in a position for which A is best. In this case, the swarm would need to continually sense the environment and potentially switch from any algorithm to any other. Implementing a more general algorithm switching behavior capable of operating in dynamic worlds is a good area for future work.

Dynamic worlds in general are a good topic to be explored because they are much closer to reality. A swarm of microrobots deployed in a real application will almost surely find a dynamic working environment. They must be tolerant to this, and be capable of responding to new situations.

Of course, the area of future work that this dissertation ultimately drives at is to implement swarm foraging algorithms on a very large swarm of real robots. The kilobots used in Chapter 5 are promising, as are the many microrobots under current development. However, when considering robots which can potentially operate on uneven terrain, walk up walls, or even fly, some modification to the communication or foraging strategies may be required. In the work in this dissertation, the robots were assumed to be operating in a flat 2D environment. In such an environment, the straight–line distance between two robots (which is what the range–and–bearing communication would measure) is roughly equal to the distance one robot would actually have to travel to reach the other. This is not the case in more complex environments. Figure 6.1 shows some examples. If robots can climb up walls, for example, a walker on the ground can not traverse the straight–line distance to a beacon on the wall, even though the operating environment is still topologically 2–dimensional. With flying robots, the situation is even more complex. It is likely that with flying robots, beacons would decide to land, rather than hover in the air transmitting. In this case, there would be a beacon field on the ground, with walkers (which should now be called "flyers") operating above them. Again, the straight–line distance from a flyer to a beacon is not the path the flyer would actually take to reach that beacon in the course of following a gradient. The flyer would probably maintain some constant altitude, using the beacons as waypoints.

---

[1]For example, the beacon could add both of its gradient values (call this `mySum`), add both of the gradient values from each of the other beacons it can hear (call this list `theirSums`), find the largest value in `theirSums` (call this `theirMaxSum`), and test if `mySum > theirMaxSum`. If so, the beacon is on the perimeter. Interestingly, a similar algorithm can be used for a beacon to decide if it is on the shortest path between the nest and the food. This is the case if `mySum < theirMinSum`. Notice

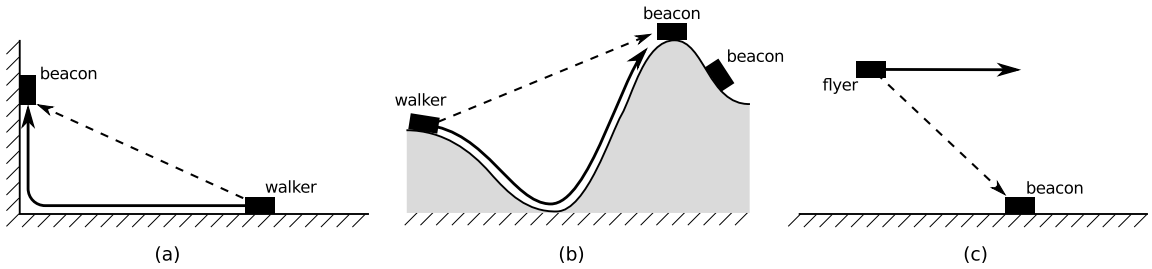


Figure 6.1: Examples of several environments in which the straight–line distance between a walker and a beacon is different from the path that the walker must actually take to follow the beacons. (a) shows the case of robots which can walk and perch on walls. (b) shows a complex terrain which distorts path lengths and can hide beacons. The straight–line distance is not always shorter than the path they walker would take, as shown in (c).

It is possible that the walkers could still use the same algorithm, ignoring these effects. The purpose of the beacon field is not to estimate the distance between the walker and any other point. Instead, the beacon field provides a navigational aid. It tells the walkers which direction to move in, not necessarily how far they will have to go. With this in mind, the walkers should just move in the direction of the beacon they want to move to, and hopefully they will eventually reach it. If the walker does not reach the intended beacon for some reason, then it will be in the range of other beacons or it will become a beacon itself. The `gradient` algorithm, at least, should be tolerant to the effects of non–flat environments. The `sweeper` algorithm, however, will not be. That algorithm relies explicitly on the straight–line distance between robots, so it will behave incorrectly in the presence of walls and hills.

There are other concerns that robots will face in real environments that they did not in the simulations and experiments in this dissertation. Perhaps the most important one is energy management. Robots should be able to measure the amount of energy remaining in their battery and recharge when necessary. If recharging requires returning to the base, then the algorithm needs to be tolerant to robots suddenly leaving the field and going home. They could be dynamically replaced by fresh robots, but this needs to be handled. If power can be transferred between robots, then perhaps a new role should be defined in addition to walker and beacon. Robots in this new role could pick up energy at the nest, bring it to the robots in the field, and return for more. If robots can simply harvest energy from the environment (with solar panels, for example), they must be sure to do that periodically (which may not be so simple, if they need to find a sunny area, for example). In any case, with real robots, energy management needs to be built into the algorithm.

---

that both of these decisions can be made without any additional communication.

Bibliography

[1] A. Baisch, R. Wood. Design and Fabrication of the Harvard Ambulatory Micro-Robot. *International Symposium on Robotics Research*, 2009.

[2] A. Gutierrez et. al. Open E-Puck Range & Bearing Miniaturized Board for Local Communication in Swarm Robotics. *International Conference on Robotics and Automation*, 2009.

[3] Ronald Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[4] B. Hölldobler and E.O. Wilson. *The Ants*. Springer-Verlag, 1990.

[5] B. Werger and M. J. Mataric. Robotic food chains: Externalization of state and program for minimal-agent foraging. *P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S. W. Wilson, editors, Proceedings of the International Conference on Simulation of Adaptive Behavior*, pages 625–634, 1996.

[6] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous space programs for robotic swarms. *Neural Computing and Applications, Special Issue on Swarms*, 19(6):825–847, 2010.

[7] Tucker Balch and Maria Hybinette. Social potentials for scalable multi-robot formations. In *International Conference on Robotics and Automation (ICRA)*, pages 73–80, 2000.

[8] Eric Bonabeau, Guy Théraulaz, Jean-Louis Deneuborg, Nigel Franks, Olivere Rafelsberger, Jean-Louis Joly, and Stephane Blanco. A model for the emergence of pillars, walls, and royal chambers in termite tests. *Philosophical Transactions of the Royal Society of London B*, 353:1561–1576, 1998.

[9] Rodney Brooks and Anita Flynn. Fast, cheap, and out of control: A robot invasion of the solar system. *Journal of The British Interplanetary Society*, 42:478–485, 1989.

[10] Gilles Caprari. *Autonomous Micro-Robots: Applications and Limitations*. Ph.D. thesis, EPFL, 2003.

[11] Chris Jones and Maja Mataric. Behavior-Based Coordination in Multi-Robot Systems. *Autonomous Mobile Robots: Sensing, Control, Decision-Making, and Applications*, 2005.

[12] D. Lambrinos, R. Möller, R. Labhart, R. Pfeifer, and R. Wehner. A mobile robot employing insect strategies for navigation. *Robotics and Autonomous Systems*, 30:39–64, 2000.

[13] D. Payton, M. Daily, R. Estkowski, M. Howard, and C. Lee. Pheromone Robotics. *Autonomous Robots*, 11(3):319–324, 2001.

[14] Brian Decleene, Neil Immerman, Jim Kurose, and Don Towsley. Leader election algorithms for wireless ad hoc networks. In *in Proceedings DARPA Information Survivability Conference and Exposition*, pages 22–24, 2003.

[15] Marco Dorigo, Elio Tuci, Roderich Groß, Vito Trianni, Thomas Halva Labella, Shervin Nouyan, Christos Ampatzis, Jean-Louis Deneubourg, Gianluca Baldassarre, Stefano Nolfi, Francesco Mondada, Dario Floreano, and Luca Maria Gambardella. The swarm-bots project. In Erol Sahin and William M. Spears, editors, *Swarm Robotics*, volume 3342 of *Lecture Notes in Computer Science*, pages 31–44. Springer Berlin / Heidelberg, 2005.

[16] E. Barth. A dynamic programming approach to robotic swarm navigation using relay markers. *In Proceedings of the 2003 American Control Conference*, 6:5264–5269, 2003.

[17] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence*. Oxford University Press, 1999.

[18] F. Adler and D. Gordon. Information collection and spread by networks of patrolling ants. *The American Naturalist*, 140(3):373–400, 1992.

[19] J. Gautrais, C. Jost, and G. Theraulaz. Key behavioural factors in a self-organised fish school model. *ANNALES ZOOLOGICI FENNICI*, 45:415–428, 2008.

[20] Markus Hannebauer, Jan Wendler, Enrico Pagello, Luca Iocchi, Daniele Nardi, and Massimiliano Salerno. Reactivity and deliberation: A survey on multi-robot systems. In *Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

[21] A. Howard, M. Mataric, and G. Sukhatme. Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem. *Sixth International Symposium on Distributed Autonomous Robotics Systems*, pages 299–308, 2002.

[22] B. Hrolenok, S. Luke, K. Sullivan, and C. Vo. Collaborative foraging using beacons. In *Proc. of 9th Int. Conf. on Autonomous Agents and Multi-agent Systems (AAMAS 2010)*, 2010.

[23] S. Ichikawa and F. Hara. Experimental characteristics of multiple-robots behaviors in communication network expansion and object-fetching. *Distributed Autonomous Robotic Systems*, pages 183–194, 1996.

[24] J. Deneubourg, S. Aron, S. Goss, and J. Pasteels. The self-organising exploratory pattern of the argentine ant. *Journal of Insect Behaviour*, 3(2):159–168, 1990.

[25] J. Svennebring and S. Koenig. Building Terrain-Covering Ant Robots. *Autonomous Robots*, 16(3):313–332, 2004.

[26] J. Werfel. Building Blocks for Multi-Agent Construction. *Distributed Autunomous Robotic Systems*, 2004.

[27] K. O'Hara, D. Walker, and T. Balch. The GNATs Low-cost Embedded Networks for Supporting Mobile Robots. pages 277–282, 2005.

[28] M. Krieger, J-B. Billeter, and L. Keller. Ant-like task allocation and recruitement in cooperative robots. *Nature*, 2000.

[29] M. Nakamura and K. Kurumatani. Formation mechanism of pheromone pattern and control of foraging behavior in an ant colony model. *In C. G. Langton and K. Shimohara, editors, Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*, pages 67–76, 1997.

[30] Mamei et al. Spreading Pheromones in Everyday Environments via RFID Technologies. *2nd IEEE Symposium on Swarm Intelligence*, 2005.

[31] J. McLurkin. Measuring the accuracy of distributed algorithms on Multi-Robot systems with dynamic network topologies. *9th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, 2008.

[32] J. McLurkin and D. Yamins. Dynamic task assignment in robot swarms. *Proceedings of Robotics: Science and Systems, June*, 8, 2005.

[33] James McLurkin. *Stupid Robot Tricks: A Behavior-Based Distributed Algorithm Library for Programming Swarms of Robots*. S.M. thesis, Massachusetts Institute of Technology, 2004.

[34] James McLurkin. *Analysis and Implementation of Distributed Algorithms for Multi-Robot Systems*. Ph.D. thesis, Massachusetts Institute of Technology, 2008.

[35] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, Floreano D., and A. Martinoli. The e-puck, a robot designed for education in engineering. *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, 1(1):59–65, 2009.

[36] S. Nouyan, R. Groß, M. Bonani, F. Mondada, and M. Dorigo. Teamwork in self–organized robot colonies. *IEEE Transactions on Evolutionary Computation*, 13(4):695–711, 2009.

[37] C.A.C. Parker, Hong Zhang, and C.R Kube. Blind bulldozing: multiple robot nest construction. In *IEEE Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 2010–2015, 2003.

[38] Chris A. C. Parker and Hong Zhang. Consensus-based task sequencing in decentralized multiple-robot systems using local communication. *IEEE International Conference on Inelligent Robots and Systems (IROS)*, 2008.

[39] Chris A. C. Parker and Hong Zhang. Collective unary decision-making by decentralized multiple-robot systems applied to the task-sequencing problem. *Swarm Intelligence*, 2009.

[40] J. Parrish, S. Viscido, and D. Grunbaum. Self-organized fish schools: An examination of emergent properties. *Biological Bulletin*, 202:296–305, 2002.

[41] J. Pugh and A. Martinoli. The cost of reality: Effects of real–world factors on multi-robot search. *Proceedings of the IEEE International Conference on Robotics and Automation*, 2007.

[42] R. Beckers and J.L. Deneubourg. Trails and U-turns in the Selection of a Path by the Ant *Lasius Niger*. *Journal of Theoretical Biology*, 159:397–415, 1992.

[43] R. Beckers, J.L. Deneubourg, and S. Goss. Modulation of trail laying in the ant *Lasius Niger* and its role in the collective selection of a food source. *Journal of Insect Behaviour*, 6(6):751–759, 1993.

[44] R. Russell. Heat trails as short-lived navigational markers for mobile robots. *In Proceedings of the International Conference on Robotics and Automation*, pages 3534–3539, 1997.

[45] R. Russell. *Odour Sensing for Mobile Robots*. World Scientific, 1999.

[46] R. Sharpe and B. Webb. Simulated and situated models of chemical trail following in ants. *In Proceedings of the International Conference on Simulation of Adaptive Behavior*, pages 195–204, 1998.

[47] R. T. Vaughan, K. Stoy, G. S. Sukhatme, and M. J. Mataric. Blazing a trail: insect-inspired resource transportation by a robot team. *Proceedings of the International Symposium on Distributed Autonomous Robot Systems*, 2000.

[48] R. T. Vaughan, K. Stoy, G. S. Sukhatme, and M. J. Mataric. Whistling in the dark: Cooperative trail following in uncertain localization space. *In C. Sierra, M. Gini, and J. S. Rosenschein, editors, Proceedings of the Fourth International Conference on Autonomous Agents*, pages 187–194, 2000.

[49] R. Vaughan, K. Stoy, G. Sukhatme, and M. Mataric. LOST: Localization-space trails for robot teams. *IEEE Transactions on Robotics and Automation*, 18(5):796–812, 2002.

[50] Mike Rubenstein and Radhika Nagpal. Kilobot: A robotic module for demonstrating behaviors in a large scale (2^{10} units) collective. *Modular Robotics Workshop, IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2010.

[51] S. Rutishauser, N. Correll, and A. Martinoli. Collaborative coverage using a swarm of networked miniature robots. *Robotics and Automated Systems*, 2009.

[52] S. Goss and J. L. Deneubourg. Harvesting by a group of robots. In *First European Conference on Artificial Life*, pages 195–204. MIT Press, 1992.

[53] T. Schmickl and K. Crailsheim. Trophallaxis within a robot swarm: Bio-inspired communication among robots in a swarm. *Autonomous Robots*, 25:171–188, 2008.

[54] M. Schwager, J. McLurkin, J. J. E. Slotine, and D. Rus. From theory to practice: Distributed coverage control experiments with groups of robots. In *Proceedings of International Symposium on Experimental Robotics*, Athens, Greece, jul 2008.

[55] William M. Spears, Diana F. Spears, Jerry C. Hamann, and Rodney Heil. Distributed, physics-based control of swarms of vehicles. *Autonomous Robots*, 17(2–3):137–162, 2004.

[56] K. Sugawara, T. Kazama, and T. Watanabe. Foraging behavior of interacting robots with virtual pheromone. *IROS 2004*, 3:3074–3079, 2004.

[57] James Traniello. Foraging strategies of ants. *Annual Review of Entomology*, 34:191–210, 1989.

[58] S. Yoon, O. Soysal, M. Demirbas, and C. Qiao. Coordinated locomotion of mobile sensor networks. *5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, pages 126–134, 2008.

[59] Chih-Han Yu. *Biologically–Inspired Control for Self–Adaptive Multiagent Systems*. PhD thesis, Harvard University, 2010.