Bar-Ilan University
Department of Computer Science

OF ROBOT ANTS AND ELEPHANTS

by

Asaf Shiloni

Advisor: Prof. Gal Kaminka

Submitted in partial fulfillment of the requirements for the Master's degree
in the department of Computer Science

Ramat-Gan, Israel
June 2010

This work was carried out under the supervision of

Prof. Gal A. Kaminka

Department of Computer Science, Bar-Ilan University.

Abstract

Investigations of multi-robot systems often make implicit assumptions concerning the computational capabilities of the robots. Despite the lack of explicit attention to the computational capabilities of robots, two *computational classes* of robots emerge as focal points of recent research: Robot *Ants* and robot *Elephants*. Ants have poor memory and communication capabilities, but are able to communicate using pheromones, in effect turning their work area into a shared memory. By comparison, elephants are computationally stronger, have large memory, and are equipped with strong sensing and communication capabilities. Unfortunately, not much is known about the relation between the capabilities of these models in terms of the tasks they can address. In the first part of this thesis, we present formal models of both ants and elephants, and investigate if one dominates the other. We present two algorithms: AntEater, which allows elephant robots to execute ant algorithms; and ElephantGun, which converts elephant algorithms—specified as Turing machines—into ant algorithms. By exploring the computational capabilities of these algorithms, we reach interesting conclusions regarding the computational power of both models.

In the second part of this thesis, we investigate a specific problem involving two ants. In order to create a cooperative intelligent behavior, ants may need to get together; however, they may not know the locations of other ants. Hence, we focus on an ant variant of the *rendezvous problem*, in which two ants are to be brought to the same location in finite time. We introduce three algorithms that solve this problem for two ants by simulating a bidirectional search in different environment settings. Two algorithms for an environment with no obstacles and a general algorithm that handles all types of obstacles. We provide detailed discussion on the different attributes, size of pheromone required, and the performance of these algorithms.

Acknowledgments

To Prof. Gal Kaminka, for giving me the opportunity, for believing in me, for arguing with me, for guiding me, for being a role model, for arguing again until the point is made, for trusting me, for being a good friend.
To Alon Levy, for working with me around the clock and for his insightful ideas - the ultimate study buddy and co-writer.
To Noa Agmon, Yehuda Elmaliach, Ariel Felner, and Meir Kalech, for being wonderful research partners and for imporving my research abilities, each in their own way.
To the MAVERICK lab, for being my friends and making me feel like home.
To Prof. Manuela Veloso, Prof. Milind Tamnbe, and Prof. Yonatan Aumann, for their feedback and good advice.
To my family, for supporting me through this journey in any way possible.
To Ella, my one and only, for being my constant source of energy.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Investigations of multi-robot systems, from a computational perspective, often focus on algorithms for specific tasks and applications [15, 11]. Such algorithms make explicit their assumptions concerning the sensing and actuation morphologies of the robots. However, more often than not, assumptions as to the computational capabilities of the robots are left implicit. They can be determined by examining the requirements of the algorithms, and the basic set of atomic actions they utilize.

The first part of this thesis defines two important computational classes of robots: *robot ants* and *robot elephants*. We examine the computability of these classes (see Section 1.1). We find that on one hand, one robot ant is computationally equivalent to one robot elephant. On the other hand, multiple robot elephants strictly dominate multiple robot ants. These are the first computability results in this area.

In the second part of the thesis (described briefly in Section 1.2), we present a family of algorithms for robot ants, all focus on a single canonical problem (rendezvous). Solving this problem is key to enabling ants to carry out complex multi-robot tasks, and also stands as the basis for transforming algorithms for more computationally-capable robots, into algorithms for robot ants.

1.1 Ants and Elephants

Despite the lack of explicit attention to the formal computational capabilities of robots, two *computational classes* of realistic robots emerge as marking extreme points in recent research: Robot *ants*, which have restricted computational and communication capabilities, but can utilize pheromones to read/write messages in their environment, and robot *elephants*, which have strong computation and communication capabilities, but no pheromones. Other computational classes lie somewhere in between these extremes, e.g., many types of swarm robotics models [9, 36] which often share some computation restrictions with ants, but similarly to elephants, do not have pheromones. We focus here on ants and elephants.

Robot *ants* [33, 47, 45, 48] are usually memory-less (or have severe memory limitations) and have relatively weak sensing abilities, if any [19, 46]. However, they can communicate through the environment, by leaving behind pheromones which essentially turn the environment into a shared memory. Robot ants have been shown to be able to carry out impressive robotic tasks, such as terrain coverage [19, 46, 35] and foraging [33, 23].

Robot *elephants* seem—by comparison—significantly stronger from a computational perspective. These have a large amount of memory (as large as needed, for instance, to hold a full map of the work area), and are equipped with strong sensing, computation, and communication machinery. Robot elephants have similarly been shown to work in the same tasks as above (e.g., [15, 11]).

However, while researchers of ant-robotics and elephant-robotics have tackled similar tasks, the actual computational capabilities and limitations of the two models remain open questions. Empirical comparisons between solutions are difficult and rare, in part due to the different metrics and experiment designs in each community. Moreover, most robots in practice are more limited than the prototypical elephants described above, but less restricted than the robot ants; this makes distinguishing the underlying computational advantages and disadvantages of different robot models even more challenging.

In this thesis, we seek to theoretically distinguish the two extreme models and their basic computability capabilities. We present formal models of both ants and elephants in a grid environment, and investigate if one dominates the other,

they are equivalent, or rather each has its own advantage over the other and thus, they are incomparable. We present two algorithms: AntEater, which allows elephant robots to execute ant algorithms; and ElephantGun, which converts elephant algorithms—specified as Turing machines—into ant algorithms.

By exploring the computational capabilities of these algorithms, we reach interesting conclusions regarding the computational power of both models. We find that a group of elephant robots can easily simulate every ant algorithm run by a group of ant robots. Moreover, we find that a single ant robot can fully simulate a single elephant robot, given infinite space. However, we show that there exist problems, which multiple elephants can solve and ants cannot. This is the first general computability result in this area.

1.2 Ants Rendezvous Problem

As mentioned before, many canonical problems in robotics, such as terrain coverage and foraging, were shown to be successfully solved with ants. Furthermore, ants are believed to be practical in future real-world applications. For example, the advantage of ants over conventional robots may be found in the micron-scale environments such as the blood stream and other parts of the human body. The robots acting in these environments cannot be equipped with abundant memory or high-end sensors, nor with high computational performance capabilities. Thus, only robots with light hardware capabilities must be used [17, 16].

Algorithms for ant robots often require cooperation between multiple ants. These algorithms usually assume that the ants start executing the algorithm while they are located at the same location, or in a close proximity, and would not work otherwise. For example, in the row straightening technique [44] the assumption is that the ants are close enough to each other in order to align themselves using local interactions. Similarly, in self-assembly problems [5, 42] the assumption is that the ants are close enough to cooperate and even physically attach themselves to each other in order to create a tree structure of ants.

If the ants are scattered in the environment, they first need to meet in order to execute these algorithms. Furthermore, ant algorithms with multiple ants usually terminate when the mission is accomplished. In many cases, after termination the

ants are scattered around the environment. For example, in the cleaning protocol [46] the algorithm halts when the entire environment is clean, but each ant ends up in a different location. Similarly, this happens in the coverage algorithm [28]: the ants end up in different locations. In many cases, the ants should be gathered for future tasks.

Therefore, in the second part of this thesis, we address the *ant rendezvous problem* which is the problem of bringing **two** ants from arbitrary positions to a common position in finite time[1].

A naive protocol for solving the rendezvous problem will instruct one ant to cover the whole area (using any of the existing coverage algorithms [28, 15, 46]) and the other ant to wait, thus alleviating the need for coordinating the meeting. However, this algorithm has three drawbacks. First, it requires the ants to decide in advance, which ant searches and which stays. Unfortunately, the ants do not have any direct communication nor do they know the other ant's unique id in advance. Second, in this protocol only one ant makes the search. If this ants fails the entire process will fail because the search is not distributed among the ants. Third, in this protocol one of the ants does all the work while the other remains idle. Clearly, if the search is distributed, time can be reduced too.

In this thesis, we address the rendezvous problem of two ants and present three algorithms for various environment settings:

- An algorithm for an environment with no obstacles, the *No-Obstacles Algorithm* (NOA).
- An attempt at an algorithm for an environment with rectangular obstacles of finite size, the *Rectangular-Obstacles Algorithm* (ROA).
- A general algorithm that handles all types of obstacles, the *General-Obstacles Algorithm* (GOA).

In these algorithms, ants communicate by leaving pheromone tracks in the environment. To solve the rendezvous problem, each ant runs the same algorithm separately and tries to find the other ant by the pheromone tracks it leaves. The

[1] Generalizing this to more than two ants is not always trivial and is left for future work

main idea of these algorithms is to simulate a breadth-first search from their initial position. Thus, the algorithms guarantee convergence and thus, the meeting of the ants.

In the first algorithm (NOA), the ants move in a spiral until sensing each other pheromones. In the second algorithm (ROA), the ants also move in spiral, but are also bypassing rectangular obstacles of finite size. As we later show, this attempt to extend NOA for a relatively simple environment fails, and this algorithm is left as an alternative for NOA in an environment with no obstacles. Nevertheless, the development of the ROA leads to important insights. These, in turn, lead to the development of the last algorithm (GOA), where the ants move in an iterative deepening manner. This guarantees that the area surrounding them will be covered until they finally meet.

1.3 List of Publications

The work reported in this thesis has been published in the following conferences:

- A. Shiloni, N. Agmon, and G. A. Kaminka. On ants and elephants. In Proceedings of the AAMAS-08 Workshop on Formal Models and Methods for Multi-Robot Systems, 2008.
- A. Shiloni, N. Agmon, and G. A. Kaminka. Of robot ants and elephants. In AAMAS, 2009 (Full paper).
- A. Shiloni, A. Levy, A. Felner, and M. Kalech. Ants meeting in an unknown environment. In MABS '09: The 10th International Workshop on Multi-Agent-Based Simulation, 2009.
- A. Shiloni, A. Levy, A. Felner, and M. Kalech. Ants meeting algorithms. In AAMAS, 2010, In press.

Chapter 2

Background and Related Work

Ant robots are usually described as memoryless or more formally as finite state machines [46, 45] i.e., having only a constant amount of internal memory, the size of which is independent of the problem size. Furthermore, they typically have limited sensing capabilities [19, 47]. Common to all previous work is the assumption that the ants are not able to use conventional planning methods [47]. What distinguishes the ants from other simple mobile robots is the usage of pheromones to communicate with each other. These pheromones are basically pieces of information that can take any physical form such as chemicals [33], heat [32], markings [19], RFID [24], etc., and are sometimes evaporative [45, 32].

This usage of pheromones as an indirect communication through the modification of the environment is the very basis of stigmergy [41]. This concept of using stigmergic behavior repeatedly in order to evolve self-organization is one of the strengths of the ant model, as it enables a group of ants to create probabilistic algorithms that solve problems in dynamic environments [18, 41]. Nevertheless, in this thesis we only use basic active stigmergic concepts, since we are interested in non-probabilistic solutions.

Bruckstein and Wagner have shown algorithms for area coverage by a team of ants, using evaporative [45] and non-evaporative [28] markings. While some of these pheromones are laid by the robots themselves [28], others are a part of the given workspace [46]. They considered simple robots with a bounded amount of memory [46, 48] for their model of ant robots. Their works and additional works

by Koenig et al. [20] produced upper bounds for the time it takes to complete a single or a repeated coverage by a swarm of ants. However, none of the works above prove any concrete boundaries on the ant model abilities in general.

It is important to differentiate between ants and other types of swarms robots. All swarm robot models are decentralized and have very limited sensorial and computational abilities [9, 36]. However, ants have the usage of pheromones that can be placed and sensed, and by that transform their environment into a shared memory. Other swarm robot models exist which do not use pheromones, and yet do not have the unbounded communications of robot elephants [9, 36]. We do not investigate these in this thesis.

Unfortunately, model comparisons of robots had not been often discussed. There is, indeed, an extensive theory of computation, which includes a hierarchy of calculating machines from finite state machines to Turing machines [37]. However, to the best of our knowledge, we are the first to utilize these computability results to analyze the computational power of robots. O'Kane and LaValle [27] produced a model for comparing the power of robots based on sensory abilities, but did not address computational and memory differences.

Several papers investigated classes of semi-synchronous [40] and asynchronous [29, 30] mobile robots that have all powerful sensing abilities, such as taking a snapshot of the world, in contrast to their weak memory functionality, no localization, and no sense of direction. Some interesting boundaries to these robots' abilities were found, yet we do not know if those limits stand when these robots are equipped with pheromones.

The rendezvous problem was first described in [34] and has countless variations, such as: heterogeneous or homogeneous agents, known or unknown environments, networks or planes, the agents can communicate freely, only in close proximity, or not at all, agents are synchronized or move asynchronously, known point of meeting or any location, and so on [10, 4, 3, 31, 22, 49, 8]. This thesis focuses on two homogeneous ants, both run a protocol to ensure meeting in a finite time within infinite grids.

Previous work in ant robotics (e.g., [20, 28]) address two problems close to the rendezvous problem: (1) the area coverage problem where the entire area should be visited by the ants and (2) the search problem where a certain unknown

location (e.g., a location that contains a treasure) should be found. They allow a non-evaporative, unbounded integer pheromone in any one unit of space and focus on the continuous domain coverage problem. By contrast, our problem does not assume a stationary target and we bound the size of the pheromone.

Spiral searches were also used to allow a robot finding a target object in an unknown environment with obstacles [6]. But, common robots with large memory were assumed and the algorithm presented is not applicable given the memory constraints of ants. Moreover, a stationary target was assumed while in the variant of the rendezvous problem presented in this thesis any location can serve as a meeting place.

The gathering problem is another similar multi-agent problem, in which multiple agents gather into a point or a small region, within finite or expected time [26]. As in the rendezvous problem, the gathering problem has many variations such as: unlimited or limited visibility, shared or no directionality (common compass), synchronous or asynchronous agents, etc. [14, 7, 39, 2, 38]. However, all of these works assume the agents are stationed initially in close proximity.

Chapter 3

Elephants Imitating Ants

In this section, we provide formal definitions of the ant and elephant models used throughout our work (Section 3.1). We then begin to compare between the computational power of the models, using a first algorithm (Section 3.2) that allows multiple elephants to execute an algorithm for multiple ants. Lastly, we show an algorithm for a group of a more restricted variant of elephants, which achieves the same property (Section 3.3).

3.1 Definitions

For simplicity, we will use a grid as the environment in which the ants and elephants interact. Nonetheless, we note that some of the proofs ahead are valid even on continuous domains.

Definition 1. *World*
The world is an infinite two dimensional regular square grid. Each cell can be either blocked *(with an obstacle, even if only partly) or* free*. Pheromones may only be placed in* free *cells. In this thesis, we handle three types of grids: (1)* ClearGrid *- An infinite grid with no obstacles. (2)* RectangleGrid *- An infinite grid with bounded sized rectangular obstacles. (3)* ObstacleGrid *- An infinite grid with unbounded-sized any-shape obstacles.*

Grid decomposition of the environment is a well known approximation for

problems solving of this kind [46, 11]. The cell unit should be at least as large as the smallest square that can surround the ant.

We define the ant model as having a representative subset of properties from the models discussed above:

Definition 2. *Ant*
An ant *is a robot that has the following attributes and abilities:*
Attributes:

- Homogeneity: *Ants are homogenous; they all have the same capabilities, and run the same algorithm.*
- Localization: *The ants share the same grid alignment but cannot recognize their location (in Section 6 we show how to eliminate the shared grid alignment requirement).*
- Directionality: *Ants do not have a notion of a global "north", but can be aware of their directionality relative to the direction they started at. We also explore (in Section 5.1) a variant of ants that have a global sense of directionality, called D-Ants.*
- Communication: *No direct communication is allowed between ants. They communicate only using pheromones (Definition 3).*
- Computational power: *From a computational point of view, ants are finite state machines. They cannot manipulate variables and cannot use recursions.*
- Anonymity: *Ants are anonymous and cannot identify each other. We also explore a variant of ants that have a unique* id*, called NA-Ants (Section 5.2).*

Actions:

- Move: *in four directions,* north, south, east, west*, all relative to their initial pose (can be arbitrarily chosen to be "north".*

- Sense: *Ants can sense the content of cells which are distanced up to a given radius. The outcome of a sense reveals the content of that cell which is either blocked, free, contains a pheromone, or contains another ant. In this thesis, we assume that the sense radius of the ant is one unit. That is, it can sense the content of its current cell and any of its eight neighboring cells.*[1]

- Write: *(or change) pheromones in its current cell. There is no limit on the number of cells that an ant can write a pheromone in (if it is located there), i.e., they have unlimited "ink".*

Communication in ants is done using pheromones as defined below:

Definition 3. *Pheromone*
A pheromone is a symbol that can be read/writen by ants in cells. Each cell can contain at most one pheromone. When a pheromone is encoded, it is divided into a finite number of fields. Each field can have a finite set of different values. Therefore a pheromone has a finite set of symbols. Pheromones do not evaporate by themselves but can be erased and rewritten by ants.

To focus the comparison between the ant and the elephant models on issues rather than sensing (already handled by [27]), we assume that the elephants have the same sensing capabilities as the ants.

Definition 4. *Elephant*
An elephant *is a robot that has the following attributes and abilities. We use emphasized text to denote differences with ants:*

Attributes:

- Homogeneity: *Elephants are homogenous in the sense that they all have the same capabilities and run the same algorithm.*

- Localization: *Elephants can typically* perfectly localize themselves on a shared coordinate system. *We call these* LF-Ants *(the* L *stands for localized). We also explore a variant of elephants that cannot localize within a global grid, called NF-Ants (Section 3.3).*

[1] In section 6.1 we show how to modify our algorithms to a sense radius of zero, where an ant must physically move to a cell in order to learn about its content.

- Directionality: *Elephants have a notion of a* global "north".
- Communication: *Elephants have* reliable, instantaneous communications *among each other.*
- Computational power: *Elephants have* unbounded *memory. From a computational point of view, elephants are Turing machines.*
- Anonymity: *Elephants have* distinct identities *and all know of each other.*

Actions:

- move: *in four directions,* north, south, east, west*, all relative to their initial pose (can be arbitrarily chosen to be "north").*
- sense: *Elephants can sense the content of cells which are distanced up to a given radius.*

The difference in computational ability between models is measured by the ability to solve different classes of problems. We define computational *dominance* similarly to the definition in [27]. Dominance is defined as follows:

Definition 5. *let A_N and B_M be models of N and M mobile robots, respectively. Then:*

- *We say that A_N* dominates *B_M and notate it $A_N \trianglerighteq B_M$ if the computational ability of A_N is at least as powerful as those of B_M, i.e., if every problem solvable by B_M is also solvable by A_N.*
- *We say that A_N* strictly dominates *B_M and notate it $A_N \triangleright B_M$ if $A_N \trianglerighteq B_M$ is true, and in addition there exists at least one problem solvable by A_N, but unsolvable by B_M.*
- *We say that A_N is* equivalent *to B_M and notate it $A_N \equiv B_M$ if $A_N \trianglerighteq B_M$ and $B_M \trianglerighteq A_N$.*
- *We say that A_N is* incomparable *to B_M and notate it $A_N \bowtie B_M$ if there exists at least one problem solvable by A_N, but unsolvable by B_M and at least one problem solvable by B_N, but unsolvable by A_M.*

3.2 The Anteater

In this section, we show that N LF-Ants (elephants with localization) computationally dominate N ants in the sense that N LF-Ants can simulate N ants, where $N \geq 1$. To do this, we use an algorithm AntEater, that is executed by the LF-Ants and simulates the behavior of the ants. We prove that this algorithm transforms the ants' algorithm, while keeping the characteristics of the original algorithm.

Algorithm 1 AntEater (Ant algorithm $\mathcal{A}$, list of robots R, map M)

```
1:  Initialize pointer p to point to first instruction in A
2:  while A has not stopped do
3:    if step in p is to write pheromone l in location (x, y) then
4:      write l in M(x, y)
5:    else if step in p is read pheromone l from location (x, y) then
6:      read value l from M(x, y)
7:    else if step in p is sense location (x, y) then
8:      Sense location (x, y) in space
9:    else if step in p is calculate values (z_0, ..., z_n) then
10:     Simulate calculation of (z_0, ..., z_n)
11:   Broadcast M to all r ∈ R
12:   if step in p is move to (x, y) then
13:     Move to location (x, y) in space
14:   Set p to point to the next instruction in A
```

The underlying idea in AntEater is to execute exactly the same movements as the ant algorithm $\mathcal{A}$, but distribute the shared memory created by the use of pheromones. The elephant receives a map M, large enough to contain the work area, with current position from localization device. Whenever $\mathcal{A}$ writes a pheromone value in the environment, AntEater writes it in the internal map kept by each LF-Ant robot. And whenever $\mathcal{A}$ reads a pheromone value, the map is accessed in memory to retrieve the value stored. The LF-Ant robots continuously communicate their map information to each other, thus making sure that their maps are identical—therefore simulated pheromones written in one LF-Ant robot's memory are readily available to all others for reading. We formally show this in Theorem 1.

Theorem 1. *Let $\mathcal{B}$ be a problem that can be solved by a group of N ants running*

algorithm $\mathcal{A}$ on an ObstacleGrid. *Then, $\mathcal{B}$ can be solved by a group of N LF-Ants running procedure* AntEater *on an* ObstacleGrid *in polynomial time while preserving $\mathcal{A}$'s robustness.*

Proof. **Task completion:** Assume that the solution for problem $\mathcal{B}$ is a collection of paths and that this collection is achieved by the ant algorithm $\mathcal{A}$ at a certain time t. Therefore, since AntEater performs the same movements as the original ant algorithm $\mathcal{A}$ and simulates its calculations and pheromones in space, the LF-Ants will perform the same collection of paths and thus, will solve the given problem.

Time complexity: Let $\mathcal{O}(n)$ be the time complexity of the original ant algorithm $\mathcal{A}$, such that n is the number of steps taken by the ant. Since in every step AntEater is going over exactly the step that would have been taken by $\mathcal{A}$, its time complexity will be $nc = \mathcal{O}(n)$, where c is the cost of broadcasting the robot's map and thus, is still a function of the number of robots. This can be achieved because AntEater does not perform any extra actions per step.

Robustness: AntEater preserves $\mathcal{A}$'s original robustness, for they eventually behave exactly the same. Lastly, as it emerges from line 2, AntEater assures termination in case the original ant algorithm itself terminates. □

We will use a coverage algorithm for ant robots called Mark-Ant-Walk, proposed by Osherovich et. al. [28], in order to exemplify the above theorem. The Mark-Ant-Walk algorithm is intended for one or more memoryless robots who use pheromones as indirect communication to perform a coverage task of an area. As advertised, Mark-Ant-Walk guaranties full coverage of a continuous area within $n\left\lceil \frac{d}{r} \right\rceil + 1$ steps, where n is the number of cells in the domain, d is the diameter of the domain, and r is the radius of the robot effector (although, the above algorithm does not know when to stop). Also, it promises immunity to noise and robustness to robot death: As long as at least one robot is alive, the complete area will be covered.

The Mark-Ant-Walk algorithm is given below (Algorithm 2). This algorithm is called continuously by each of the ant robots, with p given as the current location (whose coordinates are unknown to the robot). $R(r, 2r, p)$ denotes the robot's ability to sense pheromone level at its current position p and in a closed ring of radii r and $2r$ around p. $D(r, p)$ denotes the open disk radius r around the robot

in which it can set the pheromone level, and $\sigma(a)$ denotes the pheromone level at point a:

Algorithm 2 Mark-Ant-Walk (current location p)

1: Let $x \leftarrow \text{argmin}_{q \in R(r,2r,p)} \sigma(q)$
2: **if** $\sigma(p) \leq \sigma(x)$ **then**
3: **for all** $u \in D(r,p)$ **do**
4: $\sigma(u) \leftarrow \sigma(x) + 1$ {We mark open disk of radius r around p}
5: Move to x

Therefore, if we run AntEater with Mark-Ant-Walk as an input on LF-Ants with the same sensing capability yet with direct communication instead of the ability to read and write pheromones, it will behave as follows: First, the LF-Ant will initialize a map with its own location on it and keep updating that map all along its run time with information it receives from other robots. This can be done since LF-Ants have enough memory to create such a map. Then, in each step the LF-Ant will move exactly as the ant would have, use its effector just as the ant would have, but instead of placing pheromones, it will update their value in its own map. Also, instead of sensing for pheromones it will fetch the pheromone level from its own map. Eventually, after completing a step, it will broadcast all other robots the changes it made to the map, in case there are any. Based on Theorem 1, we maintain the original upper bound of Mark-An-Walk.

Moreover, we claim that not only does the AntEater preserve the original ant algorithm, but with some additions which are built specifically for a certain ant algorithm, we can improve its run time, efficiency, and/or robustness. As an example, the above Mark-Ant-Walk algorithm does not know when to stop. This is due to its bounded memory, which is not a function of the problem size and thus, cannot count steps to know to stop after $n \left\lceil \frac{d}{r} \right\rceil + 1$ steps, when it is assured that the area is covered. However, our LF-Ant's memory is not bounded and therefore, an addition to the algorithm of counting steps and a condition to stop after $n \left\lceil \frac{d}{r} \right\rceil + 1$ improves the original algorithm.

Indeed, we show (Theorem 2) that a group of N LF-Ants computationally dominates a group of N ants:

Theorem 2. *Let* ANT$_N$ *and* LF-ANT$_N$ *be the models presented in Subsection 3.1,*

where N is the number of robots, then LF-ANT$_N \trianglerighteq$ ANT$_N$ *for* $N \geq 1$.

Proof. Following Theorem 1, every algorithm executed by ants can be executed by LF-Ants, while completing the same goal in at most the same computational complexity and while maintaining the same characteristics. Therefore the computational ability of N LF-Ants is at least as strong as the computational ability of N ants. □

3.3 LF-Ants and NF-Ants

The LF-Ants above use a shared coordinated system thanks to their localization devices. This localization within a shared coordinate system is a key component in their dominance over ants. However, localization is not a trivial capability.

We therefore introduce the NF-Ant, which is a weaker version of the LF-Ant model. The NF-Ant model is identical to the LF-Ant model except it does not have localization and thus, two or more NF-Ants do not necessarily share the same coordinate system.

Hence, we provide a way for NF-Ants to simulate ants, of course, without localizing themselves on a shared coordinated system. This is done by an algorithm called NFantSpiral.

The main idea in the above NFantSpiral algorithm is for one robot to search for all other robots, update the new origin of their coordinate systems as its own origin, and then return to its own starting point to run the previous AntEater algorithm.

To do so, all robots will receive a map large enough to contain the work area and then will elect the robot with the lowest id as the leader ($id = 0$ w.l.o.g). The leader will then start moving in a spiral around its original position until it finds another robot. It will then send the difference between its own origin and the robot position as the robot's new origin. The leader will continue searching for other robots and will stop only if the group size equals the size of the list of robots given as input, when it will then return to its own origin. Lastly, all of the NF-Ants run AntEater.

Therefore, we can show that a group of N NF-Ants computationally dominate

Algorithm 3 NFantSpiral (Ant algorithm $\mathcal{A}$, list of robots R, map M)

```
1: if ID == 0 then
2:    Set current location as M(0, 0)
3:    r ← 1 { The number of robots traveling in the group }
4:    while r < |R| do
5:       Move within a clockwise spiral { recording movements }
6:       if ∃ robot r_i in current location (x, y) then
7:          Send (−x, −y) to robot r_i
8:          r ← r + 1
9:    Return to M(0, 0)
10:   Send all robots 'RUN'
11: else
12:   while Not received 'RUN' do
13:      if Received position (x, y) then
14:         Set M(x, y) as point of origin on M
15:   Run AntEater on (A, R, M)
```

a group of N ants:

Theorem 3. *Let* ANT$_N$ *and* NF-ANT$_N$ *be the models discussed above, where* N *is the number of robots, then* NF-ANT$_N \trianglerighteq$ ANT$_N$ *for* $N \geq 1$.

Proof. By applying NFantSpiral, a group of N NF-Ants first agree upon the origin of their map. From that moment on, they are equivalent to a group of N LF-Ants, which we have shown in theorem 2 to simulate any ant algorithm they are given. Thus, by running AntEater the group of N NF-Ants simulates the group of N ants, and therefore *NF-ANT*$_N \trianglerighteq$ *ANT*$_N$. □

Chapter 4

Ants Simulating Elephants

So, we know that a group of N NF-Ants that are communicating explicitly among themselves dominate a group of N ants. That raises the question, whether a group of ants dominates a group of NF-Ants. We start by showing that one ant dominates one NF-Ant on a ClearGrid (Section 4.1) and on an ObstacleGrid (Section 4.2). Then, we discuss two examples of problems that can be solved by both ants and elephants (Section 4.3) followed by two problems that can be solved by elephants, yet not by ants (Sections 4.3.1 and 4.3.2).

4.1 A Single Ant

We have established the fact that a group of N LF-Ants dominates a group of N ants for $N \geq 1$. This is strongly based on the communication between the LF-Ants. Therefore the question that arises is whether a single LF-Ant still dominates a single ant. In other words, after neutralizing the communication factor, is an LF-Ant computationally stronger than an ant.

We consider the subset of the general LF-Ant model - the NF-Ant model, in which the robots have no localization abilities. In the following, we prove that, surprisingly, for NF-Ants the answer is that one ant is equivalent to one NF-Ant.

The intuition is that while an ant has constant limited memory (making it equivalent to a finite state machine), it can use its own pheromones in space to give the ant the external storage needed to have the strength of a Turing machine,

given it has an infinite space to work in. Hence, in the proof we use a finite state machine and a Turing machine as the ant's and NF-Ant's computational mechanisms respectively.

However, the ant robot will need to move in space for two independent purposes: First, to simulate the NF-Ant's movements in space. And second, to utilize pheromones for storage. Thus, it will need to remember if it is simulating movements or conducting a calculation.

To solve that, we will keep track of the following two positions:

- **Memory position:** This position marks the location of the Turing machine head that is physically simulated by the ant. The memory position moves only during a calculation performed by the ant.

- **Physical position:** This position marks the actual position of the ant in space. The physical position moves only when the elephant simulated by the ant moves.

Also, we will add information to the pheromones, which will point to the directions of each position: east, west, south, north, and here. So, the pheromones will be divided into three fields: the original alphabet, a pointer with the direction to the $memory\ position$, and a pointer with the direction to the $physical\ position$. Each of the last two fields can take the form of all four basic directions as well as a symbol for pointing out that the ant is located exactly where the position is. Note that we restrict ourselves here to movements on a grid, and thus all directions include the four basic movements on a grid: east, west, south, and north[1].

Thus, when the ant simulates a calculation done by the NF-Ant, it will move in space, acting as a physical Turing machine. But, if interrupted by a movement of the NF-Ant it will first follow its own trail to find the $physical\ position$ and once reaching that position, it will move the $physical\ position$ to the target location, which the NF-Ant was supposed to move to. Similarly, when needed to continue a calculation, the ant will follow the trail to the $memory\ position$ and once reaching that position, it will continue the calculation, changing the trail to point to the new $memory\ position$.

[1] all four directions are relative to the robot's initial orientation, which is arbitrarily chosen to be "north"

However, in order to accomplish the above routine, the ant will need to be careful not to create loops of pointers or rather not to follow old trails that lead nowhere. Therefore, when the ant moves the $memory\ position$, it will both create a pointer to the $memory\ position$ in every step, even if there is already a pointer there, and create a pointer to the $physical\ position$ opposite of its own movement, except when there is already a pointer there. On the other hand, when the ant moves towards the $memory\ position$ it will not change any pointers, but follow the pointers that already exist.

More formally, an NF-Ant algorithm is a Turing machine ElephantAlgorithm such that:

$$\mathsf{ElephantAlgorithm} = (Q, \Sigma, b, \Gamma, \delta, s, F)$$

where Q is the set of states, Σ is the input's alphabet, b is the blank symbol, Γ is the tape's alphabet, δ is the transition function, s is the starting state, and F is the set of accepting states. We will define the finite state machine ElephantGun as a Turing machine without a tape (since both models are equivalent [37]), such that:

$$\mathsf{ElephantGun} = (Q', \Sigma', b, \Gamma', \delta', s', F')$$

ElephantGun will have the states $Q' = Q \cup Q$",$s' = s$,$F' = F$ where Q" is a set of additional states that are specified below, and transitions δ' that are also specified below. In addition it will be equipped with an infinite amount of each of the possible pheromones to be used. These pheromones will be constructed from a finite number of types, such that each of the symbols in Γ' is divided into three fields: The first field represents a symbol of the original alphabet Γ, the second points to the $memory\ position$, i.e. *east*, *west*, *south*, *north*, or *here*, and the third points to the $physical\ position$ with the same five options. Let us also define the operator $\overline{x}$ such that $\forall x \in \{E, W, S, N, H\}, \overline{E} = W, \overline{W} = E, \overline{S} = N, \overline{N} = S, \overline{H} = H$ where $E = east$, $W = west$, $S = south$, $N = north$, and $H = here$. Lastly, the input alphabet stays the same and hence, $\Sigma' = \Sigma$. The new states Q", will be composed as follows for each $q \in Q$ and $Z \in \{E, W, S, N, H\}$:

- $q_{setmem(E)}$- an intermediate state to update the current slot as the $memory\ position$

- $q_{setmem(W)}$- an intermediate state to update the current slot as the $memory\ position$

- $q_{setloc(Z)}$- an intermediate state to update the current slot as the robot's location

- $q_{findloc}$- an intermediate state to find the robot's location

- $q_{findloc(Z)}$- an intermediate state to find the robot's location and move one slot to $Z \in \{E, W, S, N, H\}$

Also, for each $q \in Q, p \in Q, a \in \Gamma, b \in \Gamma$:

- $q_{a,b,p,E}$- an intermediate state to find the $memory\ position$ and perform the $(q, a) \to (p, b, E)$ transition

- $q_{a,b,p,W}$- an intermediate state to find the $memory\ position$ and perform the $(q, a) \to (p, b, W)$ transition

In addition, we will replace the transitions δ by the new set of transitions δ' such that every transition in the form of $(q, a) \to (p, b, E)$ will be replaced by the following transitions, where $y \in \{E, W, S, N\}$, $z \in \{E, W, S, N, H\}$, and $\sqcup$ is the empty pheromone:

- $(q, (a, y, z)) \to (q_{a,b,p,E}, (a, y, z), y)$ - for the case that the ant is not on the $memory\ position$

- $(q, (a, H, z)) \to (p_{setmem(E)}, (b, E, z), E)$ - for the case that the ant is on the $memory\ position$

Also, we will add the following transitions, where S stands for no movement:

- $(q_{a,b,p,E}, (a, y, z)) \to (q_{a,b,p,E}, (a, y, z), y)$ - continue searching the $memory\ position$ in the pointed direction

- $(q_{a,b,p,E}, (a, H, z)) \to (p_{setmem(E)}, (b, E, z), E)$ - found $memory\ position$, process transition, and move to the east

- $(q_{setmem(E)}, (a, \sqcup, \sqcup)) \rightarrow (q, (a, H, W), S)$ - update $memory\ position$ pointer to "here" and pointer to $physical\ position$

- $(q_{setmem(E)}, (a, y, z)) \rightarrow (q, (a, H, z), S)$ - update $memory\ position$ pointer to "here"

Likewise, every transition in the form $(q, a) \rightarrow (p, b, W)$ will be replaced by the following transitions:

- $(q, (a, y, z)) \rightarrow (q_{a,b,p,W}, (a, y, z), y)$ - for the case that the ant is not on the $memory\ position$

- $(q, (a, H, z)) \rightarrow (p_{setmem(W)}, (b, W, z), W)$ - for the case that the ant is on the $memory\ position$

Also, we will add the following transitions:

- $(q_{a,b,p,W}, (a, y, z)) \rightarrow (q_{a,b,p,W}, (a, y, z), y)$ - continue searching the $memory\ position$ in the pointed direction

- $(q_{a,b,p,W}, (a, H, z)) \rightarrow (p_{setmem(W)}, (b, W, E), E)$ - found $memory\ position$, process transition, and move to the east

- $(q_{setmem(W)}, (a, \sqcup, \sqcup)) \rightarrow (q, (a, H, E), S)$ - update $memory\ position$ pointer to "here" and pointer to $physical\ position$

- $(q_{setmem(W)}, (a, y, z)) \rightarrow (q, (a, H, z), S)$ - update $memory\ position$ pointer to "here"

However, for every movement $z' \in \{E, W, S, N\}$ and for every $z \in \{E, W, S, N\}$,$y \in \{E, W, S, N, H\}$ of the elephant, the ant will have the following new transitions:

- $(q, (a, y, z)) \rightarrow (q_{findloc(z')}, (a, y, z), z)$ - for the case that the ant is not on the $physical\ position$

- $(q, (a, y, H)) \rightarrow (q_{setloc(z')}, (a, y, z'), z')$ - for the case that the ant is on the $physical\ position$

Together with the following new transitions:

- $(q_{findloc(z')}, (a, y, z)) \rightarrow (q_{findloc(z')}, (a, y, z), z)$ - continue searching the $physical\ position$ in the pointed direction
- $(q_{findloc(z')}, (a, y, H)) \rightarrow (q_{setloc(z')}, (a, y, z'), z')$ - found $physical\ position$, update pointer, and move to the desired direction $z' \in \{E, W, S, N\}$
- $(q_{setloc(z')}, (a, \sqcup, \sqcup)) \rightarrow (q, (a, \overline{z'}, H), S)$ - update $physical\ position$ pointer to "here" and $memory\ position$ pointer to where you came from
- $(q_{setloc(z')}, (a, y, z)) \rightarrow (q, (a, y, H), S)$ - update $physical\ position$ pointer to "here"

And lastly, for any action or sensing need to be done:

- $(q, (a, y, z)) \rightarrow (q_{findloc}, (a, y, z), z)$ - for the case that the ant is not on the $physical\ position$
- $(q, (a, y, H)) \rightarrow (q, (a, y, H), S)$ - for the case that the ant is on the $physical\ position$

Together with the following new transitions:

- $(q_{findloc}, (a, y, z)) \rightarrow (q_{findloc}, (a, y, z), z)$ - continue searching the $physical\ position$ in the pointed direction
- $(q_{findloc}, (a, y, H)) \rightarrow (q, (a, y, H), S)$ - found $physical\ position$, the ant can sense or act

Let us observe an example of an ElephantGun execution. Assume we are given the following simple NF-Ant algorithm $ElephantGunExample$ (Algorithm 4).

An ant running ElephantGun is given the Turing machine representation of $ElephantGunExample$ as an input. Figure 4.1 shows the a snapshot of the world four steps after the last line of the algorithm was executed (before the ant has

Algorithm 4 $ElephantGunExample$ ()

1: $x \leftarrow 150$
2: move($north, 2$)
3: move($east, 2$)
4: move($north, 4$)
5: move($east, 2$)
6: move($south, 2$)
7: move($west, 4$)
8: $x \leftarrow x + 1$

finished simulating it). The ant starts at the bold frame at the southwest part of the figure. It first saves the number 150 on the physical Turing machine it creates from west to east (shaded in the figure). It will do so using pheromones such that their symbol field will mark the original actions by the original Turing machine (here, the numbers 1,0,0,1,0,1,1,0) and their physical field will mark the direction in which the $physical\ position$ is (here, the directions are all "west"). Once finished simulating the first line, it will need to return to the $physical\ position$ in order to simulate movements along the grid. But, it will need to remember where the $memory\ position$ is, i.e., the physical Turing machine's head. So, it will mark the memory field in its pheromone as "here" and will place "east" pointers all along the way to the $physical\ position$. Then, it will start simulating lines (2–7) and move along the grid, while placing pointers to the $memory\ position$. Notice that once it crosses its own line of pheromones it does not change the memory pointer. Thus, when returning back to the $memory\ position$ for simulating the last line it will not make the same steps it has done before and go along the loop, but will shortcut using the old marks. Of course, it will do so while leaving pointers to the $physical\ position$.

In order to be sure that the procedure of moving between the $memory\ position$ and the $physical\ position$ does not include any loops or dead ends, we prove the following two lemmas.

Lemma 4. *At every instance in* ElephantGun *there is no infinite loop of pointers.*

Proof. Assume, towards contradiction, that there is a set of $physical\ position$ pointers $P = (p_1, p_2, ..., p_n)$ (w.l.o.g, since both the memory and the physical

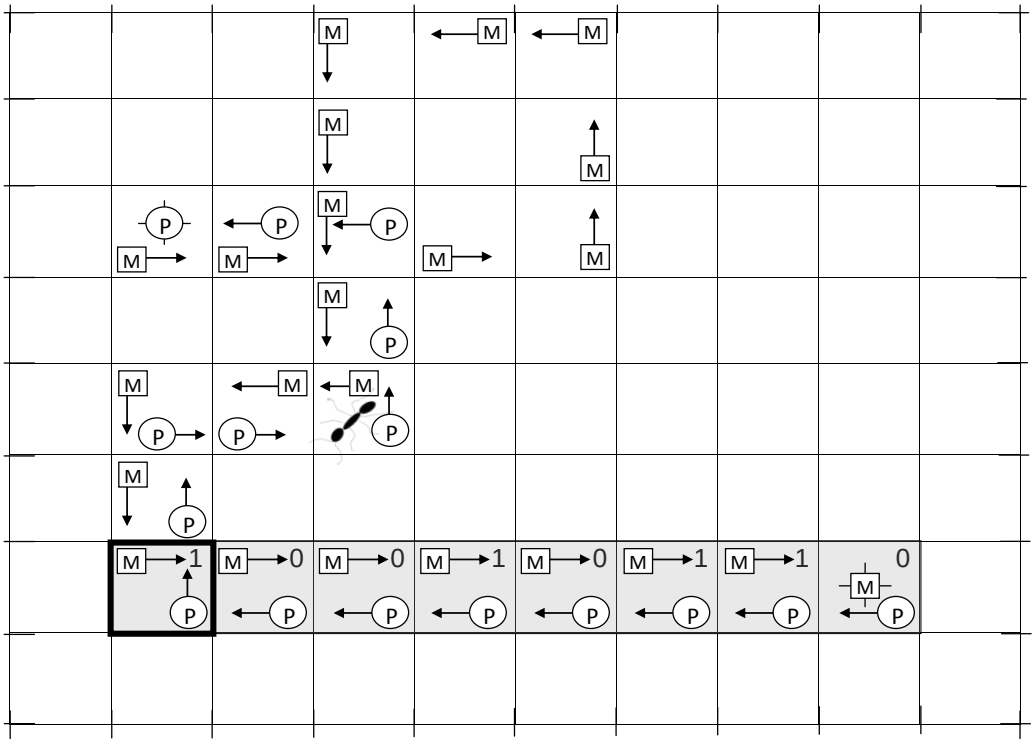


Figure 4.1: An example of ElephantGun. The shaded region is the simulated Turing machine. P symbols are the $physical\ position$ pointers. M symbols are the $memory\ position$ pointers. The bold cell is the ant's starting location.

pointers behave the same) such that $\forall i, i = 1..n, p_i \rightarrow p_{i+1} \; mod \; n$ (w.l.o.g), forming a loop. Thus, there exists no pointer p_i which points to the inside nor the outside of the loop. Now, if the loop was created by moving the $physical\ position$, then the last pointer in the loop will replace the first one and will point towards the $physical\ position$ (outside or inside the loop) and thus, will break the loop. Otherwise, if the loop was created by moving away from the $physical\ position$, the first pointer in the loop will not be replaced and will still point towards the $physical\ position$. Thus, the $physical\ position$ must be a part of the loop and since it does not point at anywhere, this is not a loop, leading to a contradiction. □

Lemma 5. *At every instance in* ElephantGun *there is a path of pointers between the* $physical\ position$ *and the* $memory\ position$ *in each direction (not necessarily the same path).*

Proof. Assume, towards contradiction, that there is no path of pointers from $physical\ position$ to $memory\ position$ (w.l.o.g). Thus, there exist at least one pointer p in the path of pointers from $physical\ position$ to $memory\ position$ that does not lead to $memory\ position$. Since the two positions start at the same place and since there is no action of erasing, we can deduce that there was a path until p was replaced, and not by moving the $memory\ position$. Therefore, p could only been replaced moving away from $memory\ position$, but such action does not replace existing pointers, leading to a contradiction. □

The former two lemmas therefore assure us that such consistent movement between the memory head and physical head can be achieved and thus, we can proceed towards constructing such simulation of the NF-Ant by the ant. Thus, we reach the following important conclusions: The first is that the ant's finite state machine with the assistance of the workspace is equivalent to the NF-Ant's Turing machine. This implies that the ant model is as least as strong as the NF-Ant when involving only one robot (lemma 6).

Lemma 6. *The finite state machine* ElephantGun*, when equipped with infinite amount of pheromones and being run on a* ClearGrid*, is equivalent to the Turing machine* ElephantAlgorithm*, as it preserves the original task completion in polynomial time using a polynomial number of constant-sized pheromones.*

Proof. **Task completion:** It is easy to see that one can construct such a finite state machine. The new transitions, states, and alphabet, though each is larger then the original, are still finite and thus, can be constructed to simulate the Turing machine. Furthermore, once created, ElephantGun uses its infinite amount of pheromones as the Turing machine's alphabet and the grid it is located in as the Turing machine's tape to write in and read from. Lastly, the extra transitions added allow ElephantGun to simulate both the ElephantAlgorithm's movements and calculations independently.

Time complexity: There are four couples of actions that can be taken by the ant: $move \rightarrow move$, $write \rightarrow write$, $move \rightarrow write$, and $write \rightarrow move$. The cost of the first two actions is 1, since the ant does not need to switch between $memory$ and $physical$ positions. Thus, in the worst case scenario there is a set $|A| = n$ of actions $A = move, write, move, ..., write$ such that the movements and writing actions are towards opposite directions. Suppose that the ant's simulated tape is written from west to east and all moves in A are moves in the west direction, then for each action the distance between the $memory$ and $physical$ positions grows by one. Therefore, the number of steps for action i is exactly i and altogether, for n actions the ant makes $\sum_{i=1}^{n} i = n(n+1)/2$ steps which is $\mathcal{O}(n^2)$ as opposed to the n steps performed by the elephant running the same algorithm.

Memory complexity: Recall that ElephantGun is a Turing machine without a tape and has also a finite number of states. Therefore, its internal memory is constant.

Size of pheromone: Let $g = |\Gamma|$ be the size of the original NF-Ant alphabet. Then, we can construct a pheromone, which will have the following three fields: (1) A $\log g$ size field to represent the original symbol. (2) A 3-bit field to represent the five directions to the memory head (east, west, south, north, and here). (3) Another 3-bit field to represent the same five directions to the physical robot head. Altogether, the size of the pheromone is $\log g + 6$ bits[2].

Total number of pheromones used: In the worst case, the ant places at most one pheromone in every step and thus, it will use no more than $\mathcal{O}(n^2)$ pheromones

[2]We are aware that the size of the pheromone can be reduced to $\log g + 5$. There are $5 \times 5 = 25 < 32$ combinations of the two pointers and therefore, they can be represented in 5 bits altogether.

for a task that requires n steps for the original NF-Ant (that is, n^2 steps for the simulating ant).

□

Theorem 7. *Let* ANT_1 *and* NF-ANT_1 *be the models portrayed above correspondingly. Then,* $\text{ANT}_1 \trianglerighteq \text{NF-ANT}_1$ *on a* ClearGrid.

Proof. Following Lemma 6, we can construct an ant that simulates the NF-Ant model which has no localization. Therefore, each problem that can be solved by the model $\textit{NF-ANT}_1$ can be solved by $\textit{ANT}_1$. Thus, $\textit{ANT}_1 \trianglerighteq \textit{NF-ANT}_1$ on a ClearGrid. □

When combining Theorem 2 and Theorem 7, we get the following conclusion for a single ant and a single NF-Ant.

Corollary 1. $\text{ANT}_1 \equiv \text{NF-ANT}_1$ *on a* ClearGrid.

Proof. Since we have shown in Theorem 2 that $\textit{NF-ANT}_N \trianglerighteq \textit{ANT}_N$ for $N \geq 1$, then $\textit{NF-ANT}_1 \trianglerighteq \textit{ANT}_1$. Also, we have shown in Theorem 7 that $\textit{ANT}_1 \trianglerighteq \textit{NF-ANT}_1$. Therefore, $\textit{ANT}_1 \equiv \textit{NF-ANT}_1$ on a ClearGrid. □

Hence, we are left with two open questions: First, what happens if there are obstacles in the environment? Secondly, what happens to the dominance relationship when we need to simulate N elephants? In the following sections we will try to extend the ant-elephant dominance problem to these two scenarios.

4.2 Environments with Obstacles

We have indeed created a mechanism for the ant to simulate any algorithm it is given by using the space around it for memory purposes. However, in reality, most robotic environments have a certain amount of obstacles and therefore, the ant cannot assume a clear working area in a predefined shape. Thus, assuming an infinite grid with an unknown number of obstacles which still maintain connectivity within the area (ObstacleGrid), we would like to have an algorithm that would iterate the grid such that it would produce an empty cell for each step.

Fortunately, the General Obstacle Algorithm (GOA) described in Section 5.3 achieves that property, since it performs a memoryless Depth First Iterative Deepening (DFID), which enables the ant to cover the whole area uniformly while ignoring obstacles and of course, while using only constant memory per move. While we wait for Section 5.3 to present the full details of GOA, we describe here certain properties and portions of DFID, which can be used to overcome obstacles in ElephantGun.

We introduce the RightMovement algorithm, which produces a right movement on a Turing machine within an obstacle prone environment (ObstacleGrid). In order to move one cell to the right in our Turing machine the ant will run the RightMovement algorithm until it has reached the next cell to work on. Just the same, in order to move one cell to the left within the Turing machine the ant can invoke the LeftMovement algorithm, which works similarly to RightMovement, in addition to the ability to identify when the ant reaches the Turing machine's leftmost cell.

In DFID, a variable d must be maintained in order to decide when to backtrack. The number of bits for this variable depends on the maximal value of d and is not constant. The ant can either store it in its internal memory, or alternatively, place it as a pheromone in a cell. However, with a constant amount of memory or with limited size of pheromone the depth of the search will be bounded. To overcome this, on each cell the ant visits, it stores the last direction it has taken from that cell (using a constant memory per cell). After visiting all directions, the ant returns to the previous depth without actually knowing what depth it is in.

So, in order to use these two new algorithms, we need to add two new fields to the pheromone used by the ant:

- $parent$**:** This field of the pheromone will point to the direction of the cell from which the ant (who placed the pheromone) arrived. There are four possible directions, a fifth symbol for the end of the Turing machine, and a null value for an empty cell. Thus, 3 bits are sufficient for this field. This field changes only when visiting a cell for the first time or when moving "left" in the Turing machine.

- $direction$**:** While the $parent$ field at cell c points to the cell that the ant

first came from to c, the $direction$ field points to the cell that the ant moved to after leaving c. Based on this field the ant determines where to go next and whether to continue the deepening or to backtrack. The $direction$ field stores only four different values (2 bits only), one for each possible direction. This field changes at every visit to a cell.

Algorithm 5 RightMovement (symbol b, state p)

```
 1: write(b)
 2: state ← p
 3: while true do
 4:    if current.direction = NULL then
 5:       current.direction = EAST
 6:    last_direction ← current.direction
 7:    nbr = sense(current.direction)
 8:    if nbr.parent = NULL then
 9:       move(current.direction)
10:       current.parent, current.direction ← last_direction + 180°
11:       break
12:    else if nbr.parent = current.direction + 180°
       ∨ current.direction = current.parent then
13:       move(current.direction)
14:       current.direction ← current.direction + 90°
15:    else
16:       current.direction ← current.direction + 90°
```

The idea behind this algorithm (Algorithm 5) is to produce right movements in a sequence determined by the DFID. Of course, due to the algorithm's nonlinear behavior (the number of steps per right movement changes dramatically when moving from one depth to another), while advancing to next cell the ant moves through old cells. This can be hazardous when implementing a Turing machine because the ant can write and read from the wrong cells. However, the ant can overcome that problem by marking every cell it passes by and by stopping its movement only when reaching an unmarked cell (the $parent$ field is NULL). The marking is done of course independently from the other ElephantGun marking discussed above since it is done only in the two new fields in the pheromone.

Let $(q, a) \rightarrow (p, b, R)$ be the next transition the ant should take such that q and p are the old and new states respectively, a and b are the old and new symbols

respectively and the ant should move right at the end of the transition (a right movement on the physical Turing machine). The ant first writes symbol b in its current cell and changes its internal state to p (lines 1–2). Then, the ant enters a while loop for searching the next cell in the Turing machine (lines 3–16). In each iteration within the loop, the ant senses the content of the neighbor pointed by the direction field of the current cell and decides its next action accordingly. If the parent field in that neighbor is empty then the ant has found the next cell in the Turing machine. The ant then moves to this new cell, sets the $parent$ and $direction$ fields to point at the last cell, and then the algorithm terminates (lines 8–11). Otherwise, if the neighbor is in the same branch in the tree as the ant (remember that DFID treats the grid as a tree), the ant moves to this cell and toggles the $direction$ field clockwise (lines 12–14). Lastly, if the neighbor is blocked by an obstacle or belongs to another branch in the tree, the ant ignores it and just toggles the $direction$ field of the current cell clockwise (line 15).

Algorithm 6 LeftMovement (symbol b, state p)

```
 1: write(b)
 2: state ← p
 3: if current.parent ≠ TM_END then
 4:    last_direction ← current.direction
 5:    current.parent ← NULL
 6:    move(last_direction)
 7:    first_direction ← current.direction
 8:    while true do
 9:       current.direction ← current.direction − 90°
10:       if current.direction = first_direction then
11:          break
12:          nbr = sense(current.direction)
13:       else if nbr.direction = current.direction + 180° then
14:          move(current.direction)
15:          first_direction ← current.direction
```

Algorithm LeftMovement works similarly to RightMovement with a few changes. Assuming transition $(q, a) \rightarrow (p, b, L)$, the ant first writes symbol b to the appropriate field in the pheromone and changes its internal state to p (lines 1–2). Then, if the $parent$ field in the current pheromone marks the end of the

Turing machine the algorithm will terminate (line 3). Else, the ant leaves the current cell towards its neighbor pointed by the $direction$ field while nullifying both $direction$ and $parent$ fields in the current cell (lines 4–6). Then, the ant saves the current $direction$ as the first direction taken (line 7) and enters a while loop in order to find the next cell (lines 8–15). In each iteration within the loop, the ant first toggles the $direction$ field counterclockwise and terminates the algorithm if the ant already explored this direction (lines 10). Otherwise, it senses the neighbor pointed by the $direction$ field and if the neighbor's $direction$ field points at the current cell the ant moves towards that neighbor (lines 13–6).

Therefore, we introduce the ObstacleGun algorithm (equivalent to an FSM), which is identical to the ElephantGun FSM except when moving the Turing machine head the ant executes the RightMovement and LeftMovement algorithms (also equivalent to FSMs) instead of just moving right and left on the Turing machine respectively.

Theorem 8. *The finite state machine* ObstacleGun*, when equipped with infinite amount of pheromones and being run on an* ObstacleGrid*, is equivalent to the Turing machine* ElephantAlgorithm*, as it preserves the original task completion in polynomial time using a polynomial number of constant-sized pheromones.*

Proof. **Task completion:** Since ObstacleGun and ElephantGun are identical except movements with the Turing machine head, we need to show that RightMovement and LeftMovement are traversing the Turing machine correctly on an ObstacleGrid. Specifically, we need to prove the following:

- Claim 1: Procedure RightMovement finds the next cell of the Turing machine
- Claim 2: Procedure LeftMovement finds the previous cell of the Turing machine
- Claim 3: If the ant is at the end of the Turing machine procedure LeftMovement does not instruct a movement

Procedure RightMovement simulates a depth first iterative deepening search until it finds a next cell in the Turing machine. At the beginning of execution the

Turing machine is empty and the ant places a pheromone with a TM_END $parent$ field and arbitrarily a "north" $direction$ field. From that point on, each cell that is being sensed fits in one of these four cases:

(1) The $parent$ field is NULL, which means that the cell is empty and that the ant has found the next cell in the Turing machine[3]. In this case, the ant moves to that cell, initializes its $parent$ and $direction$ fields for a future use, and the algorithm terminates.

(2) The $parent$ field is pointing to the current cell, which means that the sensed cell is on the same branch in the tree as the current cell. In this case, the ant moves to that cell and shifts the $direction$ field clockwise for future usage.

(3) The $parent$ and $direction$ fields are equivalent, which means that the ant has pruned the current subtree and should backtrack to explore a different subtree. In this case the ant acts similarly as in (2).

(4) The cell is blocked by an obstacle or its $parent$ field does not point at the current cell, which means that the sensed cell belongs to another branch in the tree. In this case, the ant just shifts the $direction$ field clockwise so it can sense a different cell in the next iteration. Therefore, as long as there is a connected space around the ant, RightMovement will find an empty cell in a depth first iterative deepening fashion. And we are done with Claim 1.

Procedure LeftMovement finds the previous cell by retracing the steps in RightMovement. When invoked, the ant is at the end of the tree. Thus, it first nullify the $parent$ field for a future use and leaves the current cell to its parent in the tree. Then, it systematically scans each cell's neighbors in a counterclockwise manner while following the $direction$ fields in each cell. Each such $direction$ field is initially pointed towards the current cell, since it was the last direction the ant have taken. Therefore, the ant consistently follows the neighbor, which points at the ant and then start shifting its $direction$ field until it points at the next cell. Once the $direction$ field returns to point at the first direction the algorithm terminates, since the previous cell had been found. This is because the previous cell is always a leaf in the tree. This proves Claim 2.

[3]Note that the cell could have been already visited and can contain a pheromone with information, but its $parent$ field will be empty. We are not looking for the first unexplored cell in the Turing machine, but the next cell on the right.

Lastly, if the ant is at the end of the Turing machine procedure LeftMovement does not instruct a movement as shown in line 3 in algorithm 6. This proves Claim 3.

Time complexity: The costliest action in a DFID search is when finishing depth d and moving towards depth $d+1$. At that point, one needs to backtrack d steps towards the root of the tree and then $d+1$ steps towards the first node in depth $d+1$, altogether $2d+1$ steps. Therefore, if the ant is in a maze and can move in one direction only, then it performs a depth change in every iteration and altogether $\sum_{i=0}^{n-1} 2i+1 = n^2 = \mathcal{O}(n^2)$ steps for n movements on the original Turing machine. Moreover, the cost of the four available couples of actions is at the worst case as follows: $cost(move \rightarrow move) = 1$, $cost(write \rightarrow write) = 2w+1$, $cost(move \rightarrow write) = m+3w+1$, and $cost(write \rightarrow move) = w+m+1$, where w and m are the number of write and move actions respectively such that $w+m=n$. Therefore, $move \rightarrow write$ is the costliest action. Assume an ant located at a maze and starts at a dead end, where the only direction it can take is always east. Also, assume that a sequence of n $move \rightarrow write$ actions is possible and that the ant start checking if it is on the $memory$ or $physical$ positions only after it reached the root. In this impossible scenario, the ant produces $3w+1$ steps per $write$ action and altogether $\sum_{i=0}^{n-1} 3i+1 = n^2 + n(n-1)/2 = \mathcal{O}(n^2)$ steps for n actions and therefore, the time complexity of ObstacleGun is still $\mathcal{O}(n^2)$ (the same as ElephantGun) as opposed to the n steps performed by the elephant running the same algorithm.

Size of pheromone: ObstacleGun uses the the same pheromone as ElephantGun with additional 3 bits for the $parent$ field and 2 bits for the $direction$ field. Altogether, assuming the original NF-Ant algorithm had g symbols, then the ObstacleGun's pheromone needs $\log g + 6 + 5 = \log g + 11$ bits.

Number of pheromones used: In the worst case the ant places at most one pheromone in every step and thus, it will use no more than $\mathcal{O}(n^2)$ pheromones for a task that requires n steps for the original NF-Ant. □

4.3 Multiple ants

When investigating the problems involving N robots it seems like we could easily find ones, which are solvable by a group of N LF-Ants, yet are unsolvable by a group of N ants. However, looking more closely, we find that many of these problems are indeed solvable by a group of N ants, usually at the price of additional time complexity. For instance, let the rendezvous problem Rendezvous be defined as follows.

Definition 6. Rendezvous
Given two mobile robots r_1 and r_2, which are positioned on a 2-dimensional grid in positions (x_1, y_1) and (x_2, y_2) respectively, we say that an algorithm $\mathcal{A}$ running on both robots succeeds $\iff$ for every pair of points (x_1, y_1) and (x_2, y_2), r_1 and r_2 meet within a finite time.

It is easy to construct an algorithm for two LF-Ants that can solve Rendezvous on a ClearGrid like the following Manhattan algorithm.

Algorithm 7 Manhattan (robot r)

1: broadcast initial position (x_i, y_i)
2: receive other robot's position (x_{1-i}, y_{1-i})
3: calculate mid point p of the manhattan distance between (x_i, y_i) and (x_{1-i}, y_{1-i})
4: move towards p

Since LF-Ants can communicate directly, the first two steps are possible and so is the rest of the algorithm. The time complexity for Manhattan is exactly the midpoint of the manhattan distance $m = \left\lceil \frac{|y_{1-i}-y_i|+|x_{1-i}-x_i|}{2} \right\rceil$, which is optimal in a grid. We will denote this time complexity as $\mathcal{O}(m)$. Moreover, we have shown (Algorithm 3) that an NF-Ant can also solve Rendezvous.

Nevertheless, there is also an ant algorithm which solves Rendezvous on a ClearGrid, called the No-Obstacle Algorithm, using a state machine to create a spiral and to meet the other ant (see Figure 5.2 in Section 5.1).

In this finite state machine, the ant decides upon the next step according to the pheromone's locations within the eight squares surrounding it. In each step, the ant places a pheromone in its own location and then moves according to the

FSM. Whenever it senses another ant within its sensory radius, it moves towards the other ant.

As we can see in section 5.1, in the worst case both ants will meet after creating a spiral with a $\left\lceil \frac{|y_{1-i}-y_i|+|x_{1-i}-x_i|}{2} \right\rceil$ radius. This spiral has a total distance of $\left\lceil \frac{(|y_{1-i}-y_i|+|x_{1-i}-x_i|)^2}{4} \right\rceil$ and thus, $\mathcal{O}(m^2)$ is its time complexity. Note that this time complexity is only quadratic relative to the time complexity of Manhattan.

Let us look at another problem we call Stretcher, which is a variant of the foraging problem [13].

Definition 7. Stretcher

Given a group of N robots, all of which start from some initial point p_{start} on a grid, the goal of the robots is to find an injured person that is positioned in an unknown point p_{goal} on the grid, and then bring him back to p_{start}. However, in order to carry the injured, the effort of all N robots is needed.

First, we present the following NF-Ant algorithm ElephantStretcher for solving Stretcher.

Algorithm 8 ElephantStretcher (robot r, subspace S)

```
1: while (not found injured person) ∧ (not received 'FOUND') do
2:     Search along S
3: if received 'FOUND' message then
4:     Go to position p_goal given in 'FOUND' message
5: else
6:     Broadcast 'FOUND' with own position p_goal to all robots
7:     Wait for all robots to arrive
8: Carry injured person to p_start
```

Now, we notice here that when we try to construct an ant algorithm for this problem, an ant could not inform all other ants once it finds the injured person. Nevertheless, we find an ant algorithm AntStretcher that indeed solves Stretcher, yet with a greater time complexity.

Here, we see that all N ants ignore the searching subspace given, and instead spiral together until they find the injured person, at the cost of searching the whole space and not splitting the searching task among the ants.

Algorithm 9 AntStretcher (subspace S)

1: **while** not found injured person **do**
2: Run Spiral
3: Carry injured person to p_{start}

So, are there any problems which cannot be solved by a group of N ants? Unfortunately, the answer is yes. Two examples are shown in Sections 4.3.1–4.3.2.

4.3.1 Tragedy of the Common Ant

In this section we will prove by a counter example that a group of N ants cannot fully simulate a group of N NF-Ants. As a result, since N LF-Ants dominate N NF-Ants, then N LF-Ants dominate N ants. In order to do this, we will define the following problem we call LimitedKServer.

Definition 8. LimitedKServer
Let $R = r_1, ..., r_N$ be a set of mobile robots with sensing radiuses of $\frac{M}{2N}$ each, all are positioned on a finite $M \times M$ grid such that all sensing radiuses are disjoint and their union covers the whole grid. Let $C = c_1, ..., c_x$ be a set of calls and y be a positive integer such that $y \leq x$ where y is known, but neither C nor x is known. Assume that within a finite period of time t, x calls are made such that no two calls are made in parallel. Assume that every robot $r_i \in R$ can answer a call immediately only within its sensing radius and that each call lasts for one time cycle only. We say that an algorithm $\mathcal{A}$ succeeds $\iff$ for every sequence of x calls, $\mathcal{A}$ answers exactly y. Otherwise, $\mathcal{A}$ fails.

Note that the above problem is a variant of a physical k-server problem [25] where there are a finite number of calls and the servers need to collectively answer only a certain number of these calls (see Fig. 4.2). This problem represents an abstraction of problems related to the tragedy of the commons such as overfishing [43]. For our purpose of showing that NF-Ants dominate ants we will first show that there exist an algorithm called Fisherman for NF-Ants which solves LimitedKServer. Following that, we will prove that there exist no algorithm for ants which solves that same problem.

Algorithm 10 Fisherman (list of robots R, call limit y)

```
1: calls ← 0
2: while calls < y do
3:   if a message 'CALL' has been received then
4:     calls ← calls + 1
5:     if calls = y then
6:       break
7:   else if ∃ call c within sensing radius then
8:     broadcast 'CALL' to all r ∈ R
9:     calls ← calls + 1
10:    answer call
```

Theorem 9. *Algorithm* Fisherman *solves* LimitedKServer *when running N NF-Ants on an* ObstacleGrid.

Proof. Let X be the finite set of calls. Since the N NF-Ants cover the $M \times M$ entirely, there exist no call that is overlooked by all of the NF-Ants. Also, since each NF-Ant reigns over a disjoint territory, no call is being answered by two NF-Ants. In addition, since an NF-Ant is only occupied for one cycle per call and no two calls are made in parallel, there is no situation in which an NF-Ant is occupied and cannot answer a new call. Thus, after y calls the herd of NF-Ants have answered exactly y calls and therefore, all NF-Ants break from the while loop upon receiving the y-th 'CALL' message. Lastly, there is no usage of localization along the messages transferred by the NF-Ants and thus, the NF-Ants will have no problem processing the algorithm. □

Now, if we inspect the ant behavior within LimitedKServer we can conclude the following lemmas:

Lemma 10. *There is no ant algorithm which solves* LimitedKServer *when running N ants and at least one ant moves from its initial position.*

Proof. Assume $y = x$. Therefore, if an ant algorithm involves an ant moving from its initial position p at time t, there can always exist a sequence with a call at time t at position p which will be missed and thus, the algorithm will fail. □

Lemma 11. *Every ant algorithm $\mathcal{A}$ solving* LimitedKServer *requires at least one ant to move from its position during its execution.*

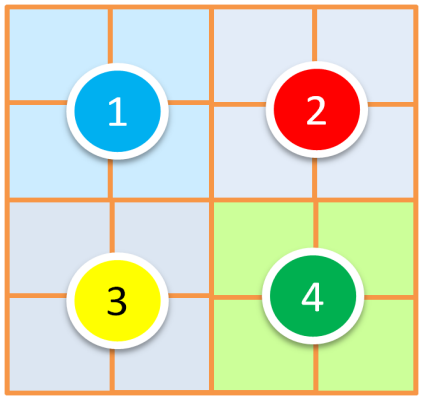


Figure 4.2: An illustration of LimitedKServer with four robots.

Proof. Since ants do not have any direct communication, the only way they can propagate information among themselves is by leaving pheromones over the grid or moving towards each other, both involve at least one ant moving. Now, suppose that there is an ant algorithm $\mathcal{A}$, which attempts to solve LimitedKServer without any movement by any of the ants. Then, there is no way an ant could know whether or not to answer the $(y+1)$-th call for it cannot know that there were y calls before. □

Based on Lemmas 10 and 11 we can now conclude:

Theorem 12. *There is no ant algorithm which solves* LimitedKServer *when running* N *ants on* ObstacleGrid.

Proof. Recall that we assume that the ant's sensing radius is identical to the LF-Ant's sensing radius. Since ants do not have explicit communication beyond their sensing radius, we can extract from the above lemma that no information can be transferred from one ant to another when trying to solve LimitedKServer. So, in order to solve LimitedKServer, any ant algorithm would need to know which calls to answer in advance, and since this information is not available we conclude that there is no ant algorithm which solves LimitedKServer when running N ants. □

Notice that the infinite space in ObstacleGrid does not assist in order to create an ant algorithm for solving LimitedKServer. This is because every ant algorithm

$\mathcal{A}$ solving LimitedKServer requires at least one ant to move from its position during its execution, as we have seen in Lemma 11. Therefore, *NF-ANT*$_N \triangleright$*ANT*$_N$, regardless of the size of the working space.

In other words, we reach the conclusion that it is not the memory deficiency, but the lack of instant communication that is what ultimately distinguishes computationally the ants from the NF-Ants. Thus, even an infinite workspace is not sufficient for ants to simulate NF-Ants in certain problems.

4.3.2 Duplicate Agent Patrol Problem

The LimitedKServer problem might strike the reader as just a latency problem, i.e, if the calls were to hold a bit longer then the ants might have been able to both answer them all and move around the area in order to propagate the total number of calls. Therefore, we would like to introduce another counterexample, which will emphasize the computational inferiority of robot ants relative to robot elephants, due to the lack of instant communication.

One of the canonical problems in multi-robot research which was mentioned earlier is the patrol problem [11, 12, 1], where a group of robots perform continuous coverage around an open or closed polygon (also known as *fence patrol*) or within an area (also known as *area patrol*). Many variants exist for this patrol problem, each changes the assumptions regarding the domain of the patrolled area or fence (friendly versus adversarial), the given task (cleaning, surveillance, etc.) and the topology of the map (graphs, continuous domains, etc.).

We would like to focus on the following variant. Assume that a top secret area is to be guarded by a group of robots patrolling the perimeter around it. This area is to be continuously visited by agents, each having a unique id. Therefore, we assume that if an agent enters the area and there is another agent already in the area with the same id, then the entering agent is a duplicate agent and therefore, should be held by the patrolling robots. Thus, we seek to find an algorithm that will be run by all robots and will assure that no duplicate agent crosses the area's perimeter.

We define the Duplicate problem formally as follows:

Definition 9. Duplicate

Let $R = r_1, ..., r_n$ be a set of mobile robots which are positioned on a perimeter fence p and let $G = g_1, ..., g_m$ be a set of agents, each having a not necessarily unique id marked $g_i.id$. Let g_i and g_j be two agents such that $g_i.id = g_j.id$ and assume g_i crossed p at time t_i. Then, agent g_j is called a duplicate if it crosses p at time t_j such that $t_i < t_j$. We say that an algorithm $\mathcal{A}$ succeeds $\iff$ each time a duplicate agent g_{dup} advances towards p there exists a robot $r \in R$ such that r stops g_{dup}. Otherwise, $\mathcal{A}$ fails.

It is easy to see that a group of N NF-Ants can solve Duplicate using any of the perimeter patrol algorithms in [1, 12] as a baseline. We propose the following DuplicateFinder algorithm (Algorithm 11).

Algorithm 11 DuplicateFinder (list of robots R, patrol algorithm PA)

```
1:  agent_list ← NULL
2:  run PA
3:  while no duplicate found do
4:      if received 'DUPLICATE' message then
5:          break
6:      if received agent.ID message then
7:          add agent.ID to agent_list
8:      if encountered agent then
9:          if agent.ID ∈ agent_list then
10:             broadcast 'DUPLICATE' to all r ∈ R
11:             break
12:         else
13:             add agent.ID to agent_list
14:             broadcast agent.ID to all r ∈ R
```

Theorem 13. *Algorithm* DuplicateFinder *solves* Duplicate *when running N NF-Ants on an* ObstacleGrid.

Proof. Each NF-Ant initializes a list of agents and runs a given patrol algorithm. Each time an NF-Ant encounters an agent (lines 8–14) it adds its id to the list and update all other NF-Ants, creating a global list. Since NF-Ants have instantaneous communication, this list is updated immediately upon sensing an agent (line 6). Therefore, suppose there exist an agent g_1 that approaches NF-Ant r_1 at time t and a corresponding duplicate agent that approaches NF-Ant r_2 at time $t + 1$, r_2 will

already have $g_1.id$ at time t (since communication is instantaneous) and therefore, will spot g_2 as a duplicate (lines 9–11). □

However, this problem cannot be solved by ants.

Theorem 14. *There is no ant algorithm which solves* Duplicate *when running* N *ants on* ObstacleGrid.

Proof. Assume $|p| = P$ such that r_1 and r_2 are two ants that are positioned on perimeter p at points p_1 and p_2 respectively and that $d(p_1, p_2) = P/2$. Now, assuming g_1 approaches r_1 at time t and r_1 is trying to propagate $g_1.id$ to all other ants. Then, in the best case $g_1.id$ is carried a distance of $P/2$ and assuming the ants can move one cell per time step the time it takes for a message to travel $P/2$ cells is $P/2$. Thus, if a duplicate agent g_{dup} approaches r_2 at time $t' < t + P/2$ then it cannot be spotted as a duplicate and the algorithm will fail. □

Chapter 5

Ants Meeting Algorithms

Until now, we have focused on the problems which ants cannot solve. We now turn to deepen into a problem that can be solved by ants – the rendezvous problem – as mentioned in section 4.3 (Definition 6). We start by presenting the No-Obstacles Algorithm (NOA), an algorithm for an environment with no obstacles, i.e., ClearGrid (section 5.1). Them, in section 5.2 we present an attempt to extend NOA to an environment with finite rectangular obstacles (RectangleGrid) called the Rectangular Obstacles Algorithm (ROA). And lastly, we present the General Obstacles Algorithm (GOA), an algorithm that can solve the rendezvous problem in an obstacle prone environment, i.e., ObstacleGrid (section 5.3).

Suzuki and Yamashita [39] have shown that there is no algorithm that can assure two simple robots to converge to a single point. The problem is that in order to achieve a meeting each robot needs to make a different decision for the same situation. In order to break this symmetry, we will need to introduce different variants of the original ant defined in section 3.1 (Definition 2).

For the first algorithm NOA, we introduce a variant of the ant model (D-Ant) which is identical to the ant in Definition 2, except the ant has the ability to know "north". For ROA and GOA, we introduce another variant of the ant model (NA-Ant) which differ from the original ant by having a unique id.

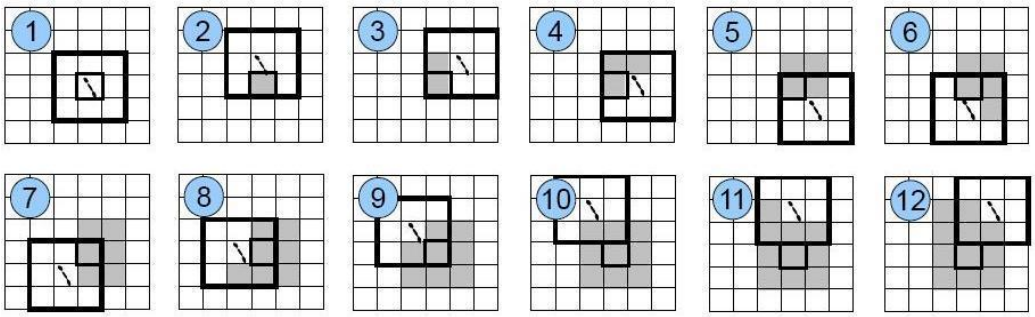


Figure 5.1: NOA: step by step ant's traveling. Gray cells are pheromones. The framed cell is the ant's starting location.

5.1 No Obstacles Algorithm

A first attempt to solve the rendezvous problem in an infinite grid with no obstacles (ClearGrid) is called the No-Obstacles Algorithm (NOA) and is indented to be used by ants which have the ability to know "north". We call this variant *D-Ants* (the *D* stands for directional). NOA also makes the additional assumption that both D-Ants execute the algorithm at the same time.

In NOA, each D-Ant spirals around its starting location by leaving a pheromone in each new cell it visits. Figure 5.1 demonstrates a step by step traveling. The bold $\{3 \times 3\}$ square box indicates the current position (center of box) and its 8 neighbors which are the sensory radius of the D-Ant. The grey cells represent pheromones left by the D-Ants. The bold cell represents the start location of the D-Ant. For example, when moving from frame 1 to frame 2, the D-Ant placed a pheromone in its own cell and moved one cell north. Then it placed a pheromone in that cell and moved east (frame 3) etc.

The D-Ant follows the FSM presented in Figure 5.2 at all times. Each state of the FSM represents the previous action of the D-Ant. Each edge corresponds to different possible sensing scenarios. The numbers above some of the scenarios that match to the frames of Figure 5.1 are labeled with the corresponding frame numbers above them.

In each state of the FSM the D-Ant activates its sensory radius (a $\{3 \times 3\}$ box) and moves according to the edge that corresponds to the content of that box. The

D-Ant starts in *Start* and moves north to state *North*. The D-Ant will stop in any case that another D-Ant is in its sensory radius (this is not shown in the FSM of figure 5.2). Each $\{3 \times 3\}$ box in the figure is interpreted as follows. The current position of the D-Ant is in the middle square. Grey cells represent pheromones, and white cells are free. X cells denote a *don't care* cell. For example, scenario 5 corresponds to the case that the D-Ant just moved *South* and that the two cells, north and north west contain pheromones.

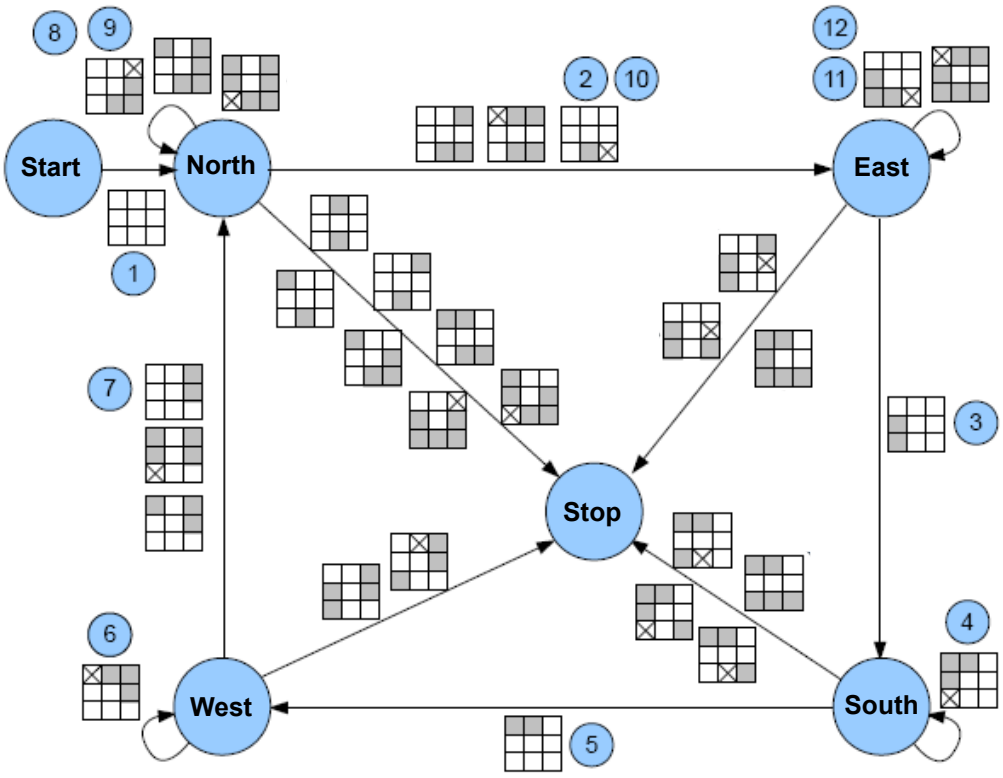


Figure 5.2: NOA finite state machine representation.

The FSM can implicitly determine from the sensory radius, based on a case by case analysis, whether a pheromone was placed by itself (its own spiral) or by the other D-Ant (it has just encountered the spiral of the other D-Ant). In this case it will either stop (move to the *Stop* state) and wait for the other D-Ant to reach it or it will continue spiraling until it finds the other D-Ant while assuming that the other D-Ant stopped. The decision is made again according to the exact content

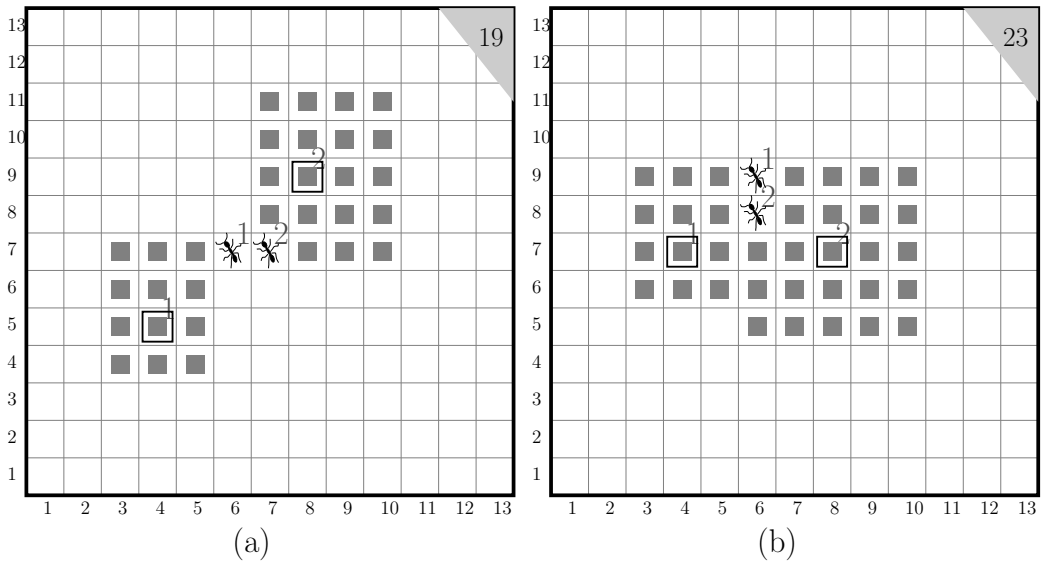


Figure 5.3: Two meeting examples with NOA. At (a), starting locations are (4,5) and (8,9).

of sensory radius. The FSM covers all possible cases (of a ClearGrid) and can be proved to be complete and correct. The main idea is that since NOA works in a ClearGrid we know exactly where each D-Ant is located with respect to its start position and a unary pheromone is sufficient.

For example, in Fig. 5.3(a), a_1 and a_2 start at the framed cells (4,5) and (8,9) respectively. Then, they start spiraling until a_1 reaches position (6,7) which has the sensory radius (a) of Fig. 5.2. It is easy to see that the pheromone in the north east could have only been placed by the other D-Ant and that this is the fringe of the spiral of the other D-Ant. Therefore, the the FSM tells that D-Ant to stop. At the same time a_2 continues its spiral and after seven steps, while located at (7,7) it senses a_1 in (6,7) and therefore the D-Ants meet and the algorithm halts. Another example is presented in Fig. 5.3(b). a_1 and a_2 start at (4,7) and (8,7) respectively. While a_1 stops due to sensory radius (b) in Fig. 5.2, a_2 continues through sensory radius (c) and (d) that directs it to move north until it finds a_1 after eleven steps.

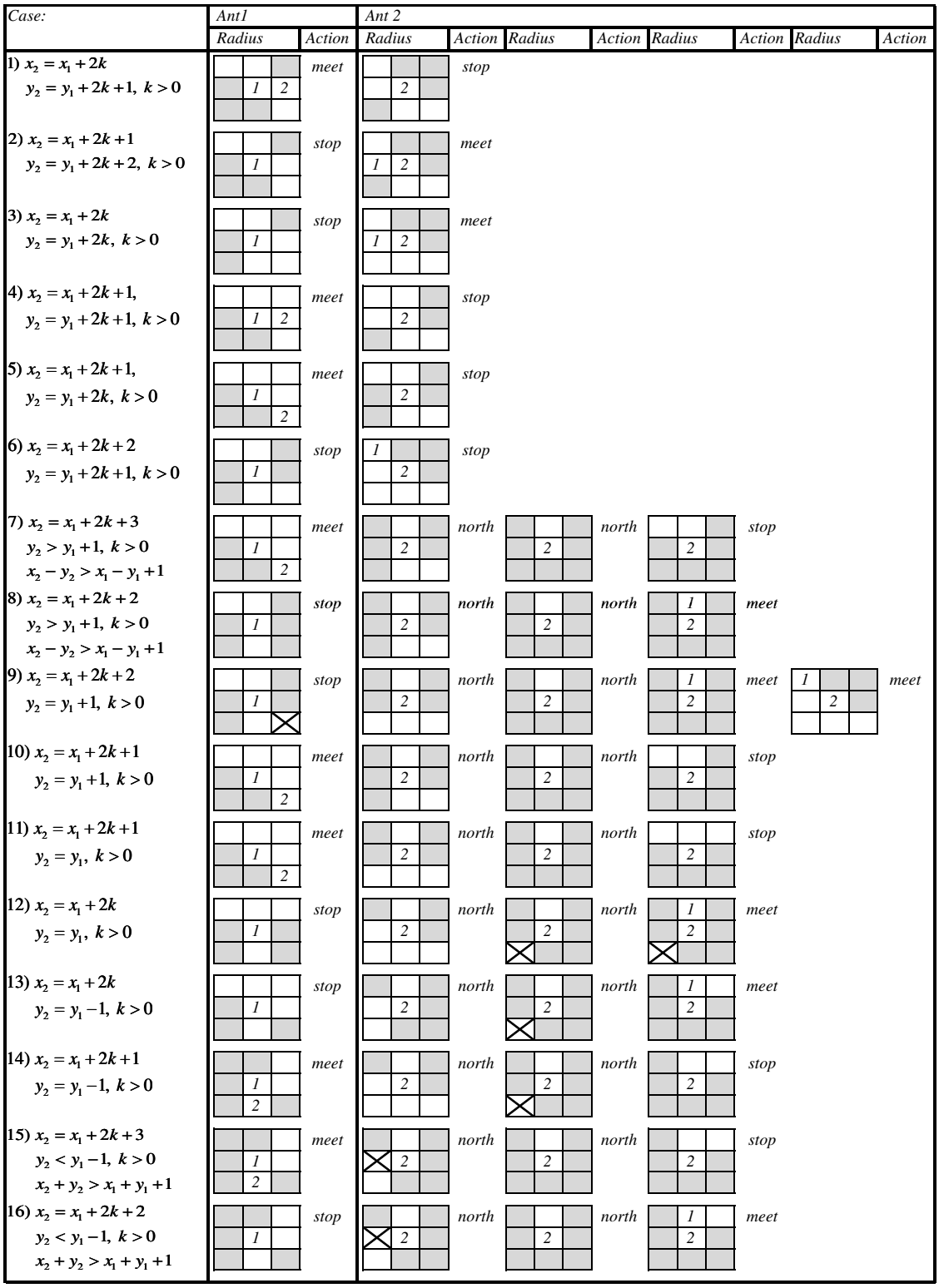


Figure 5.4: All cases of NOA.

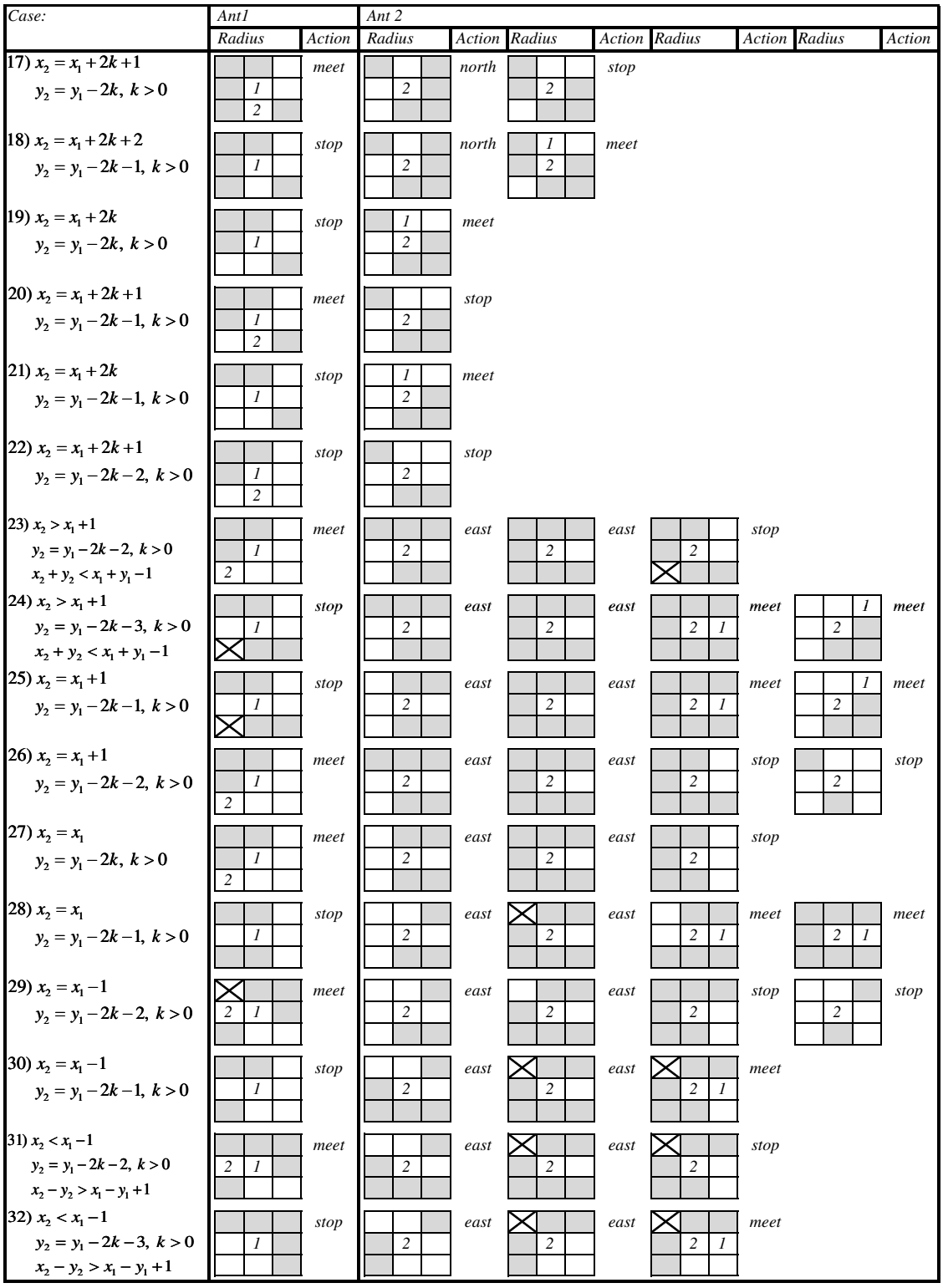


Figure 5.5: All cases of NOA (cont.).

5.1.1 Limitations of NOA

As explained above, NOA is a first step for solving the ant rendezvous problem. Its main advantage is that it can use a unary pheromone, but its limitations are that it makes a number of strict assumptions which may prevent its applicability in many scenarios. It requires both ants to start the algorithm at the same time and it can work only with D-Ants that can agree where "north" is and most importantly, it assumes the grid has no obstacles. We now turn to present ant meeting algorithms for environments with obstacles, which relief the above assumptions and instead use a different variant of ants.

5.1.2 Theoretical Analysis of NOA

NOA guarantees a meeting:

Theorem 15. *Let a_1 and a_2 be two D-Ants running* NOA *in a* ClearGrid. *Then, a_1 and a_2 meet within finite time.*

Proof. **Task completion:** Since a_1 and a_2 start the algorithm at the same time and since they start facing the same direction, then given the two initial positions of the two D-Ants, the point in which one D-Ant senses the other D-Ant's pheromones is known in advance. Also, all possible pairs of initial positions can be divided into a finite set of cases (as shown in Fig. 5.4 and 5.5). Going over all pairs of cases, in each pair only one D-Ant stops and the other continues. Therefore, a_1 and a_2 meet within finite time.

Time complexity: Let a_1 and a_2 be two D-Ants and d the initial distance between them. Then, in the worst case scenario the D-Ants start at the same longitude (or latitude w.l.o.g) an therefore, each create a square of pheromones with radius of $d/2$, until one of the D-Ants encounters the pheromones of the other. Thus, a_1 and a_2 meet after at most $\mathcal{O}(d^2)$ steps.

Memory complexity: NOA is suitable for D-Ants with very limited memory as only one variables is needed (state), with a small constant number of possible values. Thus, constant amount of memory is needed.

Size of Pheromone: The algorithm uses unary pheromones.

Total number of pheromones used: In the worst case scenario, the total number of pheromones used is equal to the time complexity, which is $\mathcal{O}(d^2)$. □

5.2 Rectangular Obstacles Algorithm

Extending NOA to work in an obstacle prone environment is not trivial. Recall that upon encountering the other ant's pheromones, the FSM in NOA directs the ant whether to stop or continue spiraling. It can do so only because it knows the exact position of the other ant, since they start at the same time, facing the same direction. However, any obstacle can break this symmetry. In NOA, the agreement on a common north have enabled the two robots to react differently to the same situation and thus, to ensure meeting. However, in an obstacle prone environment we will need to use a stronger mechanism. Therefore, let us remove the usage of the D-Ant and the assumptions of a coordinated start and instead introduce a new variant of ant. This variant is identical to the original ant defined in Definition 2 except each ant has a unique id. We call this variant *NA-Ants* (the *NA* stands for not anonymous). NA-Ants may leave their id as a field in pheromone. We assume that the different id is an ordered set (e.g. integers). Therefore, an NA-Ant can compare its own id to the id in a cell. throughout this thesis we denote the NA-Ant with the lower id by a_l and the NA-Ant with the higher id as a_h. Thus, if two NA-Ants are drawn out of k NA-Ants in order to meet in the environment, then each pheromone produced by an NA-Ant will posess a $\log k$ field for its unique id.

However, even with the addition of the unique ids, constructing an algorithm for obstacle prone environments is problematic. To illustrate this, let us start with a less complex environment and introduce the Rectangular Obstacles Algorithm (ROA), which should solve the rendezvous problem in environments with rectangular obstacles of finite size (RectangleGrid).

As in NOA, in ROA each NA-Ant creates a spiral around its starting location by leaving a pheromone in each new cell it visits. In fact, until the first obstacle is sensed, ROA behaves identically to NOA and the 12 steps shown in Figure 5.1 demonstrate the behavior of ROA too in such cases (although the pheromones will be different as will be detailed below). The main idea behind ROA is that

when an obstacle is encountered the NA-Ants should encircle the obstacles and continue spiraling.

Each pheromone in ROA includes the following fields:

- id**:** This will enable an NA-Ant to determine who placed the sensed pheromone. The number of bits for this field is $\log k$ where k is the number of NA-Ants in the system from which **two** NA-Ants are drawn. This field does not change.

- $parent$**:** This field of the pheromone will point to the direction of the cell which the NA-Ant (who placed the pheromone) arrived from. There are four possible directions plus a fifth symbol for the starting location (Null direction). Thus, 3 bits are sufficient for this field. This field does not change.

In general, the NA-Ants start spiraling and at each cell they place a pheromone with their id and with their $parent$ field. In ROA, once an NA-Ant senses an obstacle, the NA-Ant will encircle the obstacle tightly, clockwise, such that the obstacle is always on the right side of the NA-Ant. Again, similar pheromones with both fields are placed in each cell.

This process continues until an NA-Ant senses a pheromone of the other NA-Ant (by reading the id field). In this case, the protocol should make sure that the NA-Ants meet. ROA ensures this by using the following rule based on the id field.

When an NA-Ant senses the pheromone of the other NA-Ant it compares the two ids. a_l (the ant with the lower id) goes back to its own starting location while backtracking its own $parent$ field. a_h will follow the $parent$ tracks of a_l towards its starting location. Thus, they will finally meet at (or near) the staring location of a_l. The reader is encouraged to watch the video at http://vimeo.com/6930898 which demonstrates how ROA works. Although the NA-Ants move synchronously in the video, it is not a requirement for the algorithm to work.

Note that we assume a shared grid for this protocol. We show how to overcome alignment issues and therefore work without this assumption in Section 6.2.

Algorithm 12 ROA (Ant_ID id)

```
 1: if ∃ ant in radius_1 then
 2:    state ← ANT_FOUND
 3: else if state = ANT_SENSED then
 4:    move(current.parent)
 5: else if ∃ pheromone in radius_1 ∧ pheromone.id ≠ id then
 6:    state ← ANT_SENSED
 7:    if id > pheromone.id then
 8:       move(pheromone)
 9: else if state = ANT_NOT_SENSED then
10:    nbr = sense(orientation)
11:    if nbr.parent = orientation + 180° then
12:       move(current.parent)
13:       orientation ← current.parent − 90°
14:    else if nbr = NULL then
15:       move(orientation)
16:       current.parent ← orientation + 180°
17:       orientation ← orientation + 90°
18:    else {change direction counterclockwise}
19:       orientation ← orientation − 90°
```

Algorithm 12 presents the moving strategy of ROA for each NA-Ant for a given step. There are three possible states in the algorithm.
(1) *ANT_NOT_SENSED:* the NA-Ant did not sense a pheromone of the other NA-Ant yet and should continue spiraling.
(2) *ANT_SENSED:* The NA-Ant sensed the pheromone of the other NA-Ant and should follow the $parent$ field of itself or of the other NA-Ant.
(3) *ANT_FOUND:* the NA-Ant sensed the other NA-Ant.

The NA-Ant keeps track of its current orientation (i.e, north, east, south, or west) and changes it accordingly (e.g., if the orientation is "north" and the NA-Ant turns left then the orientation is now "west"). In ROA, the NA-Ant's initial orientation is chosen to be arbitrarily "east". Note again, that the NA-Ants do not agree about the directions and each of them has its own "north". The NA-Ant's current cell in each iteration is labeled as $current$ while a pheromone field in that cell is labeled as $current.field$.

The NA-Ant starts at an "ANT_NOT_SENSED" state and keeps running the

algorithm until it reaches the "ANT_FOUND" state. First, the NA-Ant senses the 8-neighbor radius. If the other NA-Ant was sensed the algorithm halts (line 1). If a pheromone of the other NA-Ant was sensed (line 5) then the state changes into "ANT_SENSED". The NA-Ant compares the ids and if its own id is larger it moves to that cell (Line 8) and it is now in a cell that was visited by the other NA-Ant. From that point on the NA-Ant just follows the $parent$ field of its current cell (line 3) until it reaches the starting location.

Lines (9–19) show the actions that are taken if the "ANT_NOT_SENSED" state is still valid and the NA-Ant needs to continue its spiral. A neighbor *nbr* is determined based on the *orientation* variable. There are three possible scenarios now and for each of them a different action is taken to guarantee that the spiral will continue. (1) lines (14–17): If *nbr* is free the NA-Ant moves to neighbor and the orientation rotates clockwise in $90\,^{\circ}$. (2) line (19): If *nbr* is blocked by an obstacle or contains a pheromone with a $parent$ field that does not point to the current cell, the orientation rotates counterclockwise in order to encircle the obstacle or the pheromone in the next step. (3) lines (11–13): the $parent$ field of nbr is pointing to the current cell. In this case, the NA-Ant reached a dead end and should backtrack.

Figure 5.6 shows different scenarios of ROA. The arrows in the figure are the $parent$ field of the pheromones and point to the previous location of the NA-Ant. The ant symbol shows the positions of the NA-Ant. Black boxes are obstacles. The framed cells are the starting locations of the NA-Ants. In Figure 5.6(a) The starting location of NA-Ant a_1 is (7,9), and that of NA-Ant a_2 is (9,4).

a_1 starts spiraling and when it reaches (9,8) it senses the obstacle in (9,7). It encircles the obstacle from left, and then after three steps it senses a pheromone of a_2 while located at (10,6). Since a_1 has a lower id, it follows its own $parent$ field towards its starting location. a_2 performs its spiral. When it reaches (9,6) (after 22 steps) it senses a_1's pheromone at (10,6). Since a_2 has a higher id, it follows a_1's pheromones. Finally, they meet in (6,8) and (6,9).

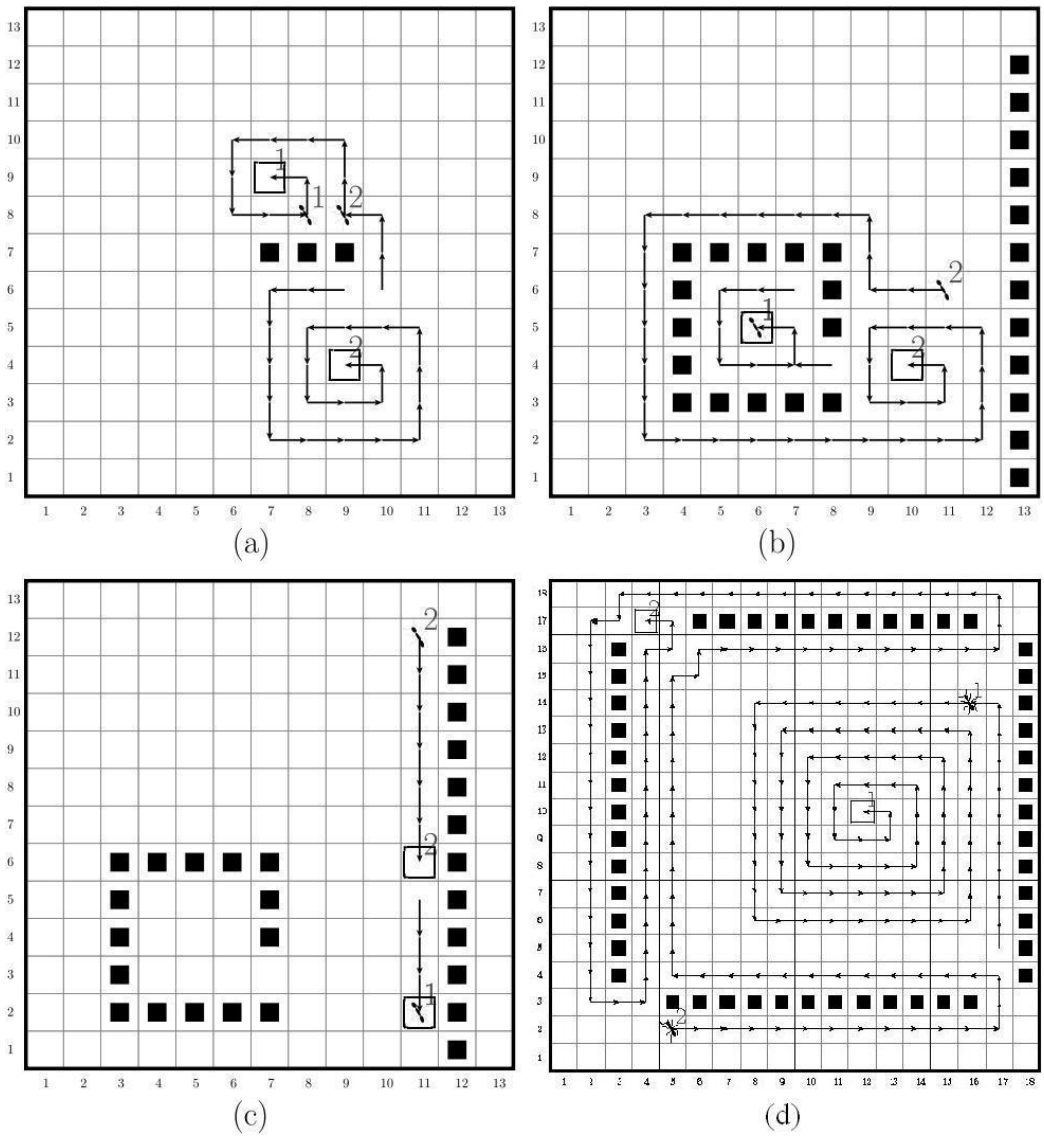


Figure 5.6: ROA examples: Successful (a) and unsuccessful (b,c,d).

5.2.1 Limitations of ROA

Unfortunately, ROA does **not** solve the rendezvous problem with obstacles. In order to assure a meeting, an algorithm must ensure a full coverage of the environment given enough time. Otherwise, if some area is not covered and the ant with the lower id is in this area, the following scenario can occur: the ant with the lower id will encounter the pheromones of the other ant and will backtrack to its own starting point. The ant with the higher id will continue spiraling while never returning to this area. Figure 5.6(d) shows such a pathological scenario. a_2 starts at (3,17) and starts spiraling around the three obstacles (west, north, south). a_1 starts at (12,10), spirals around itself, and once encountering a_2's pheromones it heads back to its own starting location. By that time, a_2 is outside the obstacle area and will keep spiraling eternally.

Also, even if ROA worked in a finite rectangular obstacle environment, it could not handle either concave or infinite obstacles. For example, Figure 5.6(b) presents the behavior of ROA in an environment with a concave obstacle. a_1's starting point is inside the concave obstacle. Once it finds the other ant's pheromone at cell (8,4) it backtracks to its starting point, since its id is lower than a_2's id. At the same time a_2 circles the obstacle, while not entering inside the "cave", since it has already blocked the entrance of the cave with its own pheromones. Thus, it continues to spiral endlessly. Another example of ROA's failure is in Figure 5.6(c) where a_2 tries to encircle the infinite wall and will never return, avoiding any possible meeting, while a_1 senses a_2's pheromone at (11,6) and returns to its own starting location, waiting forever.

Therefore, we conclude that ROA is ultimately an alternative meeting algorithm for an area with no obstacles (ClearGrid). However, as opposed to NOA, it uses NA-Ants instead of D-Ants and does not require the robots to start at the same time.

5.2.2 Theoretical Analysis of ROA

ROA guarantees a meeting in ClearGrid:

Theorem 16. *Let a_1 and a_2 be two NA-Ants running* ROA *in a* ClearGrid. *Then,*

a_1 and a_2 meet within finite time.

Proof. **Task completion:** Since there are no obstacles, each spiral covers the environment systematically until both NA-Ants eventually encounter each other's pheromones. At that point, both NA-Ants know each other's ids and travel to the starting location of the NA-Ant with the lower id using the $parent$ fields of the pheromones. Therefore, both NA-Ants meet at that location (or on the way) within finite time, since the initial distance between them is finite.

Time complexity: Let d be the initial manhattan distance between a_1 and a_2. Assume a_1 completed the search before a_2 started to act. ROA performs the same spiral path as NOA ($\mathcal{O}(4d^2) = \mathcal{O}(d^2)$) with the addition of a trip to the center of one of the spirals. This last movement is in the worst case from the corner of the spiral to its center and therefore costs d steps. Altogether, ROA's time complexity is $\mathcal{O}(4d^2 + d) = \mathcal{O}(d^2)$.

Memory complexity: ROA is suitable for NA-Ants with very limited memory as only two variables are needed (orientation and state), each with a small constant number of possible values. Thus, constant amount of memory is needed.

Size of Pheromone: The algorithm uses $\log k + 3$ bits of pheromones: three bits for marking the $parent$ field of the pheromones (four directions and one starting location), and $\log k$ bits for the id, assuming that the two NA-Ants are drawn from a population of up to k NA-Ants.

Total number of pheromones used: The total number of pheromones used is asymptotically equal to the time complexity of the spiral itself, which is $\mathcal{O}(d^2)$. □

5.3 General Obstacles Algorithm

In this section we present the General Obstacles Algorithm (GOA), which solves the rendezvous problem for NA-Ants in an environment which contains unbounded-sized any-shape obstacles (ObstacleGrid).

The main idea behind GOA is that it guarantees the coverage of the entire environment in a breadth-first manner by visiting the cells in radius r only after visiting all cells in radius $r - 1$. Therefore, it can handle concave or infinite obsta-

cles, since it does not try to encircle them. Because the NA-Ant's memory is very limited, an NA-Ant cannot implement ordinary breadth-first search (BFS) because (1) the memory grows quadratically with the depth of the search (in a grid) (2) the NA-Ant physically visits the cells and cannot instantly jump from a node to its successor in the open-list. To overcome this we use a Depth-First Iterative Deepening (DFID) search which simulates BFS without the memory limitation [21]. DFID searches a graph by performing a series of depth-first searches up to a given depth d. In each DFS call we increment d by one. However, as mentioned in Section 4.2, the number of bits for this variable depends on the maximal value of d and is not constant. The NA-Ant can either store it in its internal memory, or alternatively, place it as a pheromone in a cell. Thus, with a constant amount of memory or with limited size of pheromone the depth of the search will be bounded. To overcome this, on each cell the NA-Ant visits, it stores the last direction it has taken from that cell (using a constant memory per cell). After visiting all directions, the NA-Ant returns to the previous depth without actually knowing what depth it is in.

In GOA we use a pheromone which includes the following fields (the $parent$ and $direction$ field were already presented in section 4.2):

- id. Identical to the id field in ROA. This field does not change.
- $parent$. Identical to the $parent$ field in ROA. The $parent$ field actually spans the DFS tree and this field does not change.
- $direction$. While the $parent$ field at cell c points to the cell that the NA-Ant came from to c, the $direction$ field points to the cell that the NA-Ant moved to after leaving c. Based on this field the NA-Ant determines where to go next and whether to continue the deepening or to backtrack. The $direction$ field stores only 4 different values (2 bits only), one for each possible direction. This field changes at every visit to a cell.

GOA (Algorithm 13) presents the moving strategy of each NA-Ant. The initial state of the NA-Ant is "ANT_NOT_SENSED". GOA is identical to ROA in the steps that are taken if the other NA-Ant or the pheromone of the other NA-Ant is sensed. The main change in GOA is that the NA-Ant moves in a DFID manner

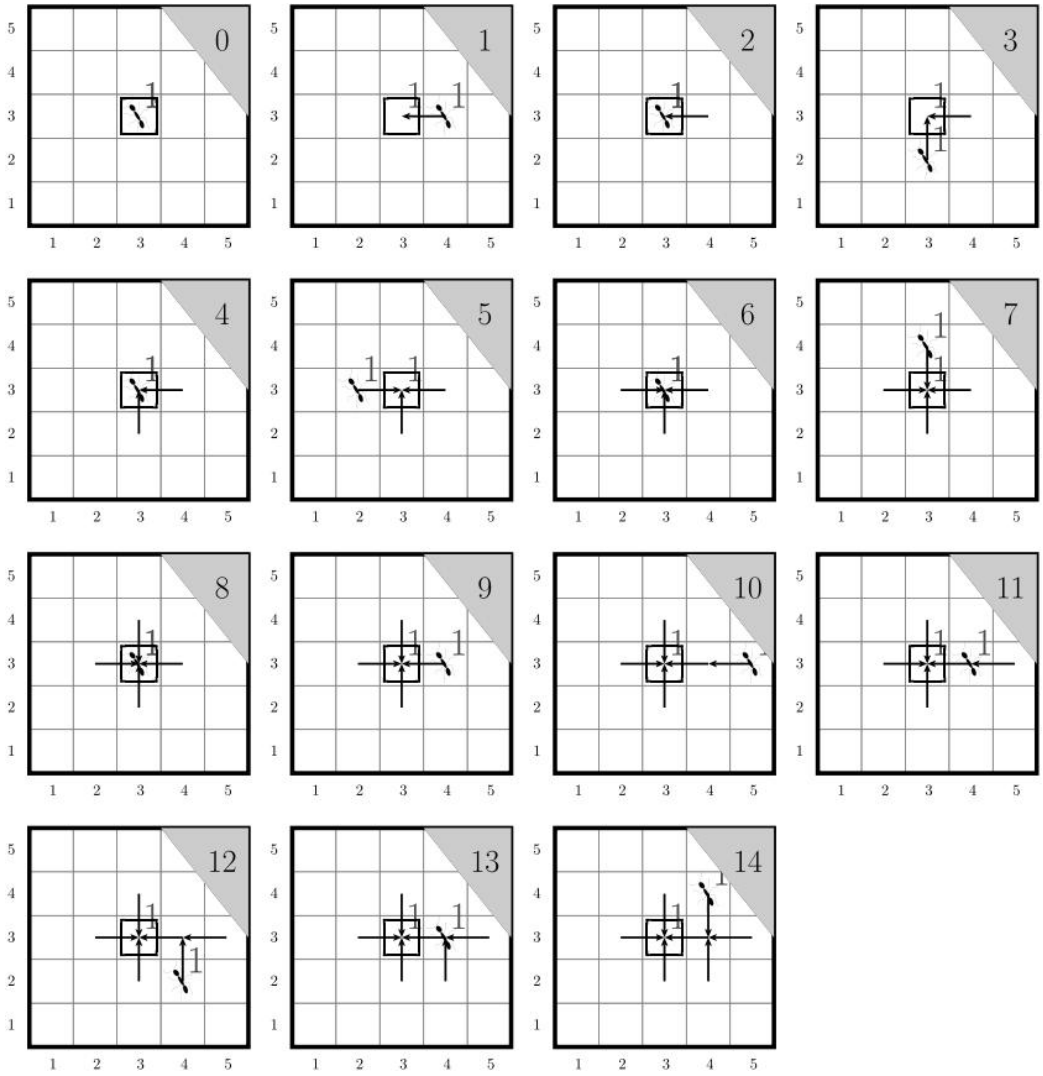


Figure 5.7: GOA demonstration on a ClearGrid. the framed cell is the NA-Ant's starting location.

Algorithm 13 GOA (Ant_ID id)

```
 1: if ∃ ant in radius_1 then
 2:     state ← ANT_FOUND
 3: else if state = ANT_SENSED then
 4:     move(current.parent)
 5: else if ∃ pheromone in radius_1 ∧ pheromone.id ≠ id then
 6:     state ← ANT_SENSED
 7:     if id > pheromone.id then
 8:         move(pheromone)
 9: else if state = ANT_NOT_SENSED then
10:     par ← current.parent, direct ← current.direction
11:     direct ← direct + 90°
12:     nbr = sense(current.direction)
13:     if cell_direct = NULL then
14:         move(direct)
15:         par, direct ← direct + 180°
16:         move(par)
17:     else if nbr.parent = direct + 180° then
18:         move(direct)
```

instead of a spiral (as in ROA). In particular, assume the NA-Ant is located in cell c. The next cell to visit is determined according to the $direction$ field in c as follows. The NA-Ant reads that field, which points to the last direction the NA-Ant have taken from c in its last visit, rotates it by $90\,^\circ$ clockwise and moves accordingly (Line 11). The meaning is that if the $direction$ field points north, then the subtree of the DFS tree rooted at the north neighbor was searched already and we should now visit the subtree in the east.

If all three subtrees were searched the $direction$ field will be now changed to be equal to the $parent$ field and the search will backtrack up in the tree. The initialization of the $direction$ field at the starting location is chosen to be arbitrarily "north".

Similarly, if an empty cell is reached (Line 13) the NA-Ant will initialize the $direction$ field to point to the parent and the search will backtrack. This ensures that only one new depth is reached in every iteration. There are two cases that we do not open a new branch from a cell c. If the $direction$ field points to a cell with (1) an obstacle (2) a $parent$ field which does not point to c. In this

case we are seeing the same node in another branch of the DFS tree and there is no point to enter this node again in the same iteration. This is referred to as *duplicate pruning* in the literature. The reader is encouraged to watch the video at http://vimeo.com/6962290 which demonstrates how ROA works.

In Figure 5.7 we show how GOA proceeds from the starting location (0) to systematically cover the area around the NA-Ant. In Figure 5.7(1) the NA-Ant moves east, returns back west in (2), then moves south and returns north (3–4), west and returns (5–6), north and returns (7–8). The first iteration to depth 1 is completed. Now an iteration to depth 2 starts. The NA-Ant moves east again in (9). At this time it moves east another time (10) and again back at (11) etc. It will then continue to all possible locations in depth 2. A result of running more steps can be seen in Figure 5.8 (a): there are no obstacles, the arrows show the $parent$ fields in the pheromones for each cell.

To illustrate more how GOA proceeds, Figure 5.8(b) shows a successful meeting when one NA-Ant starts in a cave. A more complex example is presented in Figure 5.8(c). a_1 and a_2 start at (5,5) and (10,8) respectively. Then, they start exploring all nodes of distance 1 from their starting locations, in a clockwise order while skipping any direction leading to an obstacle (e.g., (9,8) for a_2 and (5,6) for a_1). They continue this process for larger radiuses until they sense each other's pheromones at (9,3) and (10,3). Then, a_1 returns to its staring location, and a_2 follows the same path to a_1's starting location as well.

5.3.1 Theoretical analysis of GOA

GOA can be seen as a memoryless simulation of Breadth-First Search where the open list is physically distributed in the environment in the form of pheromones. We now prove that GOA guarantees a meeting:

Theorem 17. *Let a_1 and a_2 be two NA-Ants running* GOA *in an* ObstacleGrid. *Then, a_1 and a_2 meet within finite time.*

Proof. **Task completion**: DFID simulates BFS and thus, each cell at distance d will be finally reached at iteration d. The same reasoning that was presented for ROA is valid here too. The NA-Ants will either meet or will sense each other

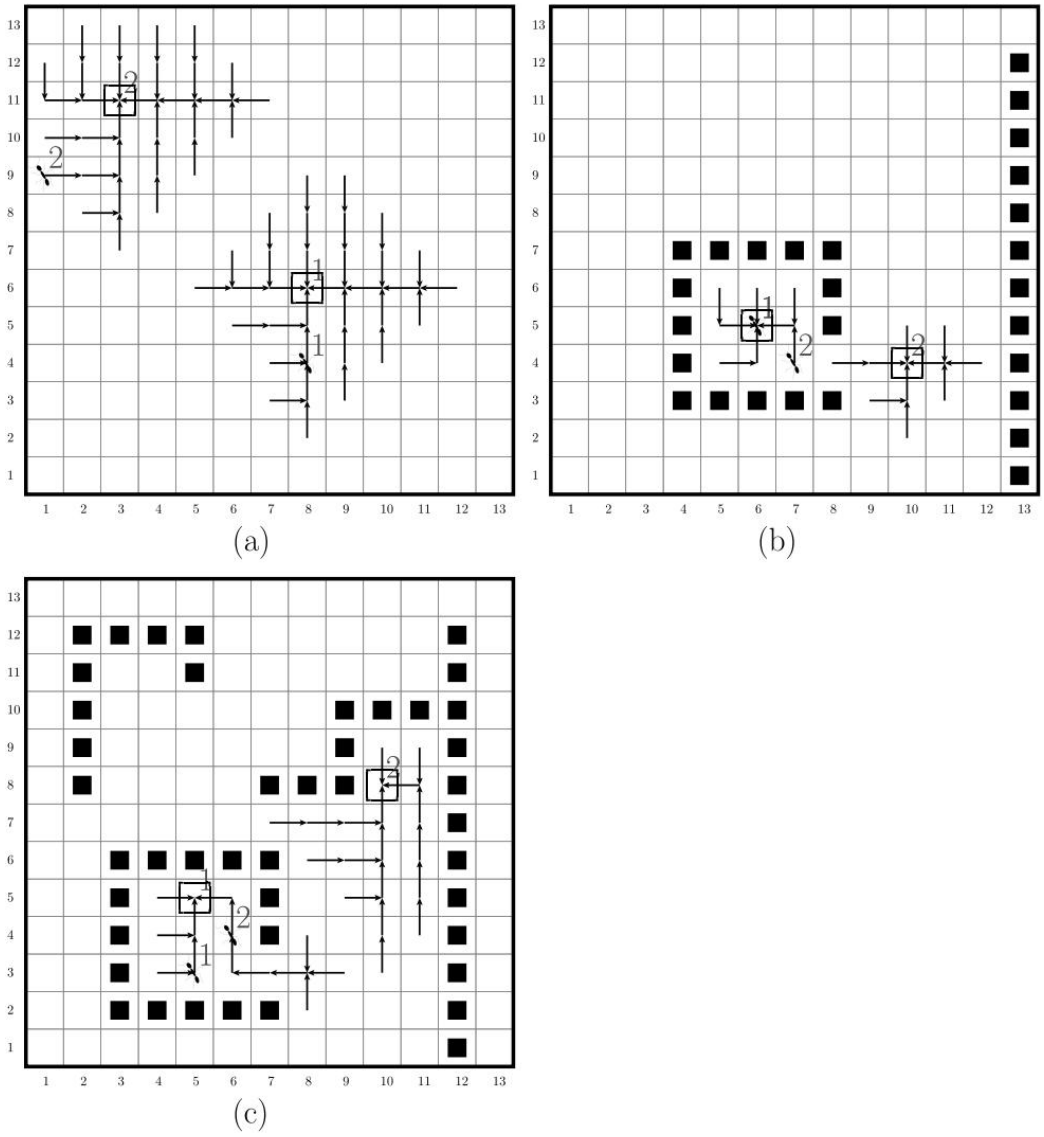


Figure 5.8: GOA examples.

stationary pheromones at some point of time and will therefore backtrack using the $parent$ field and meet in the same way as in ROA.

Time complexity: Let d be the length of shortest path between a_1 and a_2. Assume a_1 completed the search before a_2 started to act. If obstacles exists, this only reduces the number of visited cells since it is pruning the branches of the tree. Thus, assume that there are no obstacles in the grid. In this case, a_1 runs the algorithm for every depth up to depth d. The constructed tree can be seen as 4 subtrees rooted at the origin, one for each direction. Each depth in each subtree has one more node and thus, the NA-Ant iterates 1 node, then $1 + 2 = 3$ nodes, then $1+2+3 = 6$ nodes, and according to the DFID time complexity [21], GOA's time complexity in the worst case is $4[2d(d + 1)(d + 2)/6] = \mathcal{O}(d^3)$.

Memory complexity: Similar to NOA and ROA, GOA is suitable for NA-Ants with very limited memory as only one state variable with a small constant number of possible values is needed. Thus, its memory needs is constant at all times.

Size of Pheromone: The algorithm uses $\log k + 5$ bit pheromones. Three bits for marking the $parent$ field of the pheromones (four directions and one starting location), two bits for the $direction$ field and $\log k$ bits for the id, assuming that the two NA-Ants are drawn from a population of up to k NA-Ants.

Total number of pheromones used: Similarly to the time complexity, assume there are no obstacles in the world and only a_1 searches. Thus, by the time the NA-Ants meet at depth d, a_1 had produced a square of pheromones with a radius of d. thus, the total amount of pheromones placed by a_1 is $1+4\sum i = 1+4d(d+1)/2 = \mathcal{O}(d^2)$

□

Chapter 6

Extending Meeting Algorithms

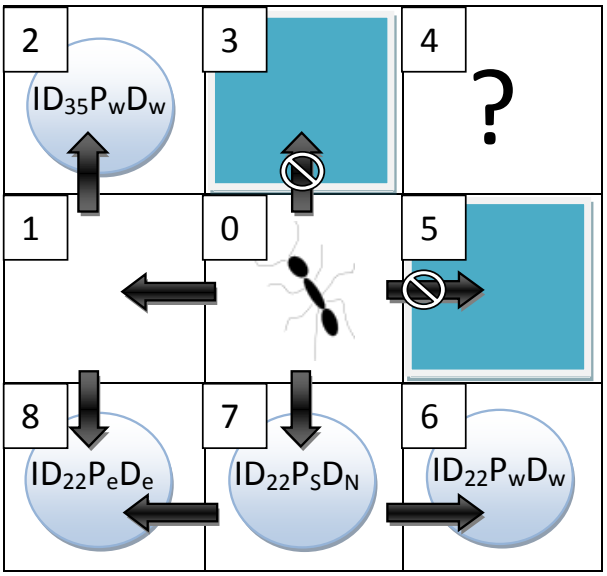
All three algorithms (NOA, GOA, and ROA) assume that the sensing radius is one and that the ants share the grid alignments. To extend the applicability and to further restrict the capabilities of the ants, we now generalize the algorithm to handle a sensing radius of zero (Section 6.1) and to the case where the grids are not aligned (Section 6.2).

6.1 Sensory Radius of Zero

So far we have assumed a sensory radius of one cell. However, in reality the simplest robots might not by able to sense in each direction, but can only sense the content of the current cell. Therefore, we show a simple routine that simulates a sensory radius of one by using a sensory radius of zero. To do this we add eight internal memory registers, each is capable of storing one pheromone.

In the algorithm with a sensing radius of one, we performed a sensing action to all eight neighbors. To simulate this with a sensing radius of zero we do the following. Each time a picture of the eight neighboring cells is needed by the algorithm then the ant will physically move to these eight cells and store their content in the corresponding eight registers.

The idea is to perform the following variant of DFS to all four directions. Assume that the ant is in cell c. The ant will move one step north (to the adjacent cell). It will then move to the east (north east corner) and backtrack and then to the



(a)

Cell	AntID	parent	direction
1	000000	110	00
2	100011	011	11
3	000000	111	00
4	000000	111	00
5	000000	111	00
6	010110	011	11
7	010110	010	00
8	010110	001	01

(b)

Figure 6.1: Simulation of a 1-radius sensing with no sensing.

left (north west corner) and backtrack. Finally, it will then move back to c. Each corner will be visited twice but this is needed in case there is an obstacle in one of the cells adjacent to c. For example, cell 2 in Figure 6.1 can only be visited via cell 1 but not via cell 3. Cell 4 may not be sensed this way. But, even assuming a sensing radius of one the exact status of that cell is of no importance and can be treated as a "don't care" as it cannot affect the decision of the ant. Eventually, after at most 24 steps the ant will be back at c with a complete vision of the eight cells around it.

Figure 6.1(a) shows the status of each cell while Figure 6.1(b) shows how this picture is encoded into the eight registers. Circles are pheromones and boxes are obstacles. Cell 4 is blocked and cannot be sensed. Thus, it is marked as an obstacle in the corresponding register.

Also, we note that the unit cell of the grid represented by the ant should be

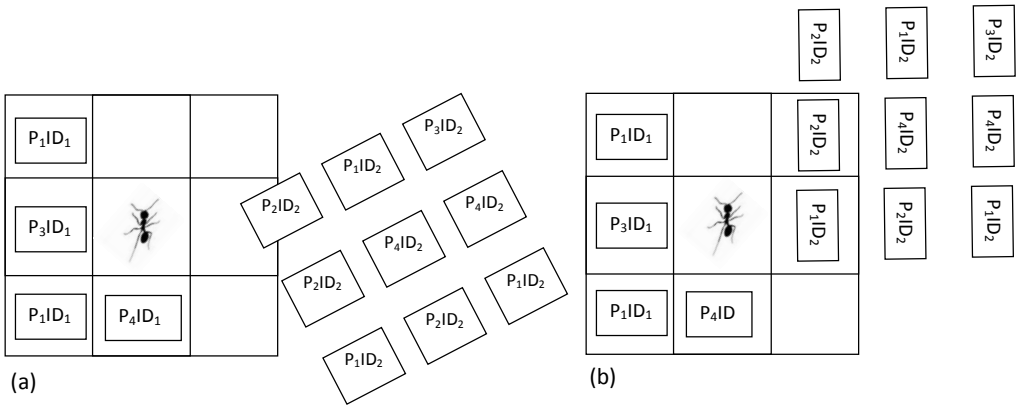


Figure 6.2: Two non aligned examples.

the smallest unit, in which the ant can fit and consequently, the sensory radius should be chosen to be the length of the maximum number of cells, such that the ant sensors are still reliable. Thus, the ant can minimize odometry mistakes by always aligning to the grid of pheromones it creates, i.e., according to the local environment.

6.2 Alignment

So far we assumed a shared grid for both ants. However, when two ants start running any of the algorithms above, the grids that each of them uses to represent the world may be non aligned in angle as shown in Figure 6.2(a). This is because the ants do not have a global sense of direction and therefore, each can call the angle it starts at as "north" (with the exception of NOA). Furthermore, even when the two grids look aligned (as in Figure 6.2(b)), their directionality could be orthogonal. In this case, assume ant a_1 encounters a_2's pheromone and recognizes the $parent$ field of "north". But, that north is relative to a_2's starting angle and therefore, a_1 cannot interpret the direction p is pointing at.

To solve this problem, we propose extensions to ROA and to GOA. Since solving this with GOA is simpler we present it first. Of course, since NOA requires aligned ants, it is not discussed in this section.

6.2.1 GOA Alignment

In fact, in order to meet (using GOA) with non aligned grids the ants do not need to align themselves to each other's grid. In addition, the resulting algorithm, GOA_align, produces a different behavior than in the original GOA. Here, instead of meeting in the starting location of a_l (the ant with the lower id), the ants meet at cell c where a_h (the ant with the higher id) first senses a_l's pheromone. When a_h first senses the pheromone of a_l it moves to the location with that pheromone and stays idle. Note that this location does not necessarily fit one cell within the ant's grid. a_l continues its DFID search until it reaches the location of a_h. Furthermore, if it finds a pheromone of a_h, it can treat it as an obstacle because we know that the other ant is in the frontier of its DFID.

Both algorithms, GOA and GOA_align, share the same asymptotic time complexity. However, GOA_align is slower than GOA because the entire DFS tree needs to be spanned up to the depth of the other ant. By contrast in GOA they both follow a simple track to the start location. Algorithm 14 presents the lines for GOA_align which should replace lines (3–8) in GOA.

Algorithm 14 GOA_align (Ant_ID id, $pheromone$)

```
1: if id > pheromone.id then
2:     move(pheromone)
3:     break
4: else
5:     pheromone.parent ← obstacle
```

6.2.2 ROA Alignment

Recall that ROA solves the rendezvous problem on ClearGrid. In ROA, once an ant visits a cell it might not return to this cell ever again. Therefore, unlike GOA, for ROA we do need to realign the ant's grid. Algorithm ROA_align is invoked upon discovering the other ant's pheromone p at some location and replaces lines (3–8) in the original ROA. a_l does not need to align and behaves exactly like in ROA. However, a_h first moves over to p, aligns itself to a neighboring pheromone p_1, and then finds a free neighboring cell c, which does not contain a pheromone

of a_l and faces towards it (lines 1–5). Of course, one such pheromone must exist because p points at one of the four directions (it has a parent). Also, the cell c must exist since the ant has just encountered the first pheromone and therefore, it must be on the fringe. Now, the ant can interpret its own orientation from the configuration of the surrounding pheromones, since they were all placed according to the same algorithm. One of the following two cases occur:

1. There is no pheromone to the ant's left (line 6): then the ant must have reached the current cell from the cell behind the ant and thus, the ant's orientation is opposite to the pheromone in the current cell.

2. There is a pheromone to the ant's left (line 8): then the ant must have reached the current cell from the cell to its left and so, the ant's orientation is off by $90\,^\circ$ counterclockwise.

Once the ant is aligned it can follow the pheromones to the other ant's starting location. Notice that this amendment does not change the time complexity of ROA nor the amount of pheromones needed.

Algorithm 15 ROA_align (Ant_ID id, p)

```
1: if id > p.id then
2:     move(p)
3:     face p_1 such that d(p, p_1) = min (d(p, p')), ∀p' ∈ radius_1
4:     face c ∈ radius_1 such that c = NULL ∨ ∀p ∈ c, p_id = id
5:     nbr_left ← sense(left)
6:     if nbr_left = NULL then
7:         orientation ← current.parent + 180°
8:     else
9:         orientation ← current.parent + 90°
```

Chapter 7

Discussion & Conclusions

It was proposed that ant robots can perform difficult computational tasks despite their weak computational abilities [46]. However, the computational limits of this model were not known. We defined elephant, as the most used model of robots with strong computational, sensing, and communication abilities and investigated the computational relationship between the two models (see Table 7.1 for a comparison of the models' capabilities).

We have shown that assuming reliable, instantaneous communication, elephant robots can simulate any task done by ant robots and therefore, are at least as computationally strong as ants. This result is not surprising, as elephants are by definition stronger. However, more surprisingly, we have also shown that given a large enough space and infinite amount of pheromones, a single ant can simulate any task done by a single elephant that has no localization abilities. Lastly, we have shown that this simulation still holds even when there are obstacles in the environment. Unfortunately, we have found a boundary for ants computational strength, as we have shown that there exist some problems that can be solved by N elephants, but not with N ants. The results can be summarized in Table 7.2.

We have also presented three algorithms that solve the rendezvous problem for two ants (see Table 7.3). NOA, a simple spiraling algorithm for a grid with no obstacles. This algorithm's running time is quadratic in the distance between the two ants and consequently, uses a quadratic amount of unary pheromones. ROA, a similar version of NOA, which solves the problem for the same environment,

but with different assumptions. And GOA, an algorithm that handles all types of obstacles, in which ants move in iterative deepening approach, while distributing the memory used for this search to the pheromones placed in the environment. Here, the running time is cubic in the distance between the two ants, but uses only a quadratic amount of pheromones, all are also of constant size. We now have a set of algorithms, each is designed to work for a different grid setting and for different variants of ants.

Now that the basic computability differences between these models are known, we hope to extend the analysis to more realistic robots, which for the most part are in-between the two computational extremes discussed above. Moreover, we wish to explore dynamic environments, in which stigmergy can be a strong factor. We also seek to combine the analysis with sensing models (e.g., as in [27]), determining complexity tradeoffs for the subset of problems that are solvable by both models, or finding out exactly how many ants are needed to simulate an elephant in minimal time and space overhead.

We also plan to extend our meeting algorithms to address the rendezvous problem for more than two ants. In addition, in this thesis we represented the world by a discrete grid, we would like to extend it to continuous environments or to other types of maps such as roadmaps.

Table 7.1: Models Summary

Robot	Localization	Directionality	Communication	Computational Power	Anonimity
LF-Ant	Yes	Yes	Instantaneous	Turing Machine	No
NF-Ant	No	Yes	Instantaneous	Turing Machine	No
D-Ant	No	Yes	Pheromones	Finite State Machine	Yes
NA-Ant	No	No	Pheromones	Finite State Machine	No

Table 7.2: Models Dominance Relationship

Model Dominance	Algorithm	Time Complexity
(1) *LF-ANT*$_N \trianglerighteq$ *ANT*$_N$, $N \geq 1$	AntEater	$\mathcal{O}(n)$
(2) *NF-ANT*$_N \trianglerighteq$ *ANT*$_N$, $N \geq 1$	NFantSpiral	$\mathcal{O}(n)$
(3) *ANT*$_1 \trianglerighteq$ *NF-ANT*$_1$	ObstacleGun	$\mathcal{O}(n^2)$
(4) *ANT*$_1 \equiv$ *NF-ANT*$_1$	(2) and (3)	
(5) *ANT*$_N \not\trianglerighteq$ *NF-ANT*$_N$, $N > 1$	LimitedKServer	Counter Example
(6) *NF-ANT*$_N \triangleright$ *ANT*$_N$, $N > 1$	(2) and (5)	

Table 7.3: Ant Meeting Algorithms

Meeting Algorithm	Environment	Ant Model	Time Complexity	Pheromone Size
NOA	Clear grid	D-Ant	$\mathcal{O}(d^2)$	1
ROA	Clear grid	NA-Ant	$\mathcal{O}(d^2)$	$\log k + 3$
GOA	Obstacle grid	NA-Ant	$\mathcal{O}(d^3)$	$\log k + 5$

Bibliography

[1] N. Agmon, S. Kraus, and G. A. Kaminka. Multi-robot perimeter patrol in adversarial settings. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-08)*, 2008.

[2] N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1070–1078, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[3] S. Alpern. The rendezvous search problem. *SIAM Journal on Control and Optimization*, 33(3):673–683, 1995.

[4] S. Alpern and S. Gal. Searching for an agent who may or may not want to be found. *Operations Research*, 50(2):311–323, 2002.

[5] H. Azzag, N. Monmarché, M. Slimane, C. Guinot, and G. Venturini. A clustering algorithm based on the ants self-assembly behavior. In W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, and J. Ziegler, editors, *European Conference on Artificial Life (ECAL)*, volume 2801 of *Lecture Notes in Computer Science*, pages 564–571. Springer, 2003.

[6] S. Burlington and G. Dudek. Spiral search as an efficient mobile robotic search technique. Technical report, Center for Intelligent Machines, McGill University, January 1999.

[7] M. Cieliebak and G. Prencipe. Gathering autonomous mobile robots. In *Proc. of 9th International Colloquium On Structural Information And Communication Complexity (SIROCCO 9)*, pages 57–72, 2002.

[8] Y. Crispin and M.-E. Ricour. Interception and cooperative rendezvous between autonomous vehicles. In *Proceedings of the Fourth International Conference on Informatics in Control, Automation and Robotics, Robotics and Automation (ICINCO-RA 2)*, pages 149–154, 2007.

[9] E. Şahin. Swarm robotics: From sources of inspiration to domains of application. In *Swarm Robotics*, volume 3342 of *Lecture Notes in Computer Science*, pages 10–20. Springer, 2005.

[10] G. De Marco, L. Gargano, E. Kranakis, D. Krizanc, A. Pelc, and U. Vaccaro. Asynchronous deterministic rendezvous in graphs. *Theoretical Computer Science*, 355(3):315–326, 2006.

[11] Y. Elmaliach, N. Agmon, and G. A. Kaminka. Multi-robot area patrol under frequency constraints. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-07)*, 2007.

[12] Y. Elmaliach, A. Shiloni, and G. A. Kaminka. A realistic model of frequency-based multi-robot fence patrolling. In *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-08)*, volume 1, pages 63–70, 2008.

[13] A. S. Fukunaga and A. B. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4:226–234, 1997.

[14] N. Gordon, Y. Elor, and A. M. Bruckstein. Gathering multiple robotic agents with crude distance sensing capabilities. In *ANTS '08: Proceedings of the 6th international conference on Ant Colony Optimization and Swarm Intelligence*, pages 72–83, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] N. Hazon, F. Mieli, and G. A. Kaminka. Towards robust on-line multi-robot coverage. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-06)*, 2006.

[16] T. Hogg. Coordinating microscopic robots in viscous fluids. *Autonomous Agents and Multi-Agent Systems*, 14(3):271–305, 2007.

[17] T. Hogg. Modeling microscopic chemical sensors in capillaries. *Computing Research Repository (CoRR)*, abs/0811.1520, 2008.

[18] O. Holland and C. Melhuish. Stigmergy, self-organization, and sorting in collective robotics. *Artificial Life*, 5(2):173–202, 1999.

[19] S. Koenig and Y. Liu. Terrain coverage with ant robots: a simulation study. In *Autonomous Agents*, pages 600–607. ACM, 2001.

[20] S. Koenig, B. Szymanski, and Y. Liu. Efficient and inefficient ant coverage methods. *Annals of Mathematics and Artificial Intelligence*, 31(1-4):41–76, 2001.

[21] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[22] E. Kranakis, D. Krizanc, and S. Rajsbaum. Mobile agent rendezvous: A survey. In *Proc. of 13th International Colloquium On Structural Information And Communication Complexity (SIROCCO 13)*, pages 1–9, 2006.

[23] T. H. Labella, M. Dorigo, and J.-L. Deneubourg. Division of labor in a group of robots inspired by ants' foraging behavior. *ACM Transactions on Autonomous Adaptive Systems*, 1(1):4–25, 2006.

[24] M. Mamei and F. Zambonelli. Physical deployment of digital pheromones through rfid technology. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1353–1354. ACM, 2005.

[25] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

[26] M. J. Mataric. Designing emergent behaviors: from local interactions to collective intelligence. In *Proceedings of the second international conference on From animals to animats 2 : simulation of adaptive behavior*, pages 432–441, Cambridge, MA, USA, 1993. MIT Press.

[27] J. M. O'Kane and S. M. LaValle. On comparing the power of robots. *International Journal of Robotics Research*, 27(1):5–23, January 2008.

[28] E. Osherovich, A. M. Bruckstein, and V. Yanovski. Covering a continuous domain by distributed, limited robots. In *ANTS Workshop*, pages 144–155, 2006.

[29] G. Prencipe. CORDA: Distributed coordination of a set of autonomous mobile robots. In *Proc. 4th European Research Seminar on Advances in Distributed Systems*, pages 185–190, May 2001.

[30] G. Prencipe. Instantaneous actions vs. full asynchronicity: Controlling and coordinating a set of autonomous mobile robots. In *Proceedings of the 7th Italian Conference on Theoretical Computer Science*, pages 185–190, October 2001.

[31] N. Roy and G. Dudek. Collaborative robot exploration and rendezvous: Algorithms, performance bounds and observations. *Autonomous Robots*, 11(2):117–136, 2001.

[32] R. Russell. Heat trails as short-lived navigational markers for mobile robots. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-97)*, volume 4, pages 3534–3539, 1997.

[33] R. Russell. Ant trails: An example for robots to follow? In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-99)*, volume 4, pages 2698–2703, 1999.

[34] T. C. Schelling. *The strategy of conflict*. Oxford University Press, 1960.

[35] A. Sempe, F.; Drogoul. Adaptive patrol for a group of robots. In *International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2865–2869, 2003.

[36] A. J. Sharkey. Robots, insects and swarm intelligence. *Artificial Intelligence Review*, 26(4):255–268, 2006.

[37] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[38] S. Souissi, X. Défago, and M. Yamashita. Using eventually consistent compasses to gather memory-less mobile robots with limited visibility. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1):1–27, 2009.

[39] I. Suzuki and M. Yamashita. Agreement on a common x-y coordinate system by a group of mobile robots. In *In proceedings of the 1996 Dagstuhl Workshop on Intelligent Robots: Sensing, Modeling and Planning*, pages 305–321. World Scientific Press, 1997.

[40] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28:1347–1363, 1999.

[41] G. Theraulaz and E. Bonbeau. A brief history of stigmergy. *Artificial Life*, 5(2):97–116, 1999.

[42] V. Trianni, E. Tuci, and M. Dorigo. Evolving functional self assembling in a swarm of autonomous robots. In S. Schaal, A. Ijspeert, A. Billard, S. Vijayakamur, J. Hallam, and J. Meyer, editors, *From Animals to Animats 8. Proceedings of the Eighth International Conference on Simulation of Adaptive Behavior (SAB 04)*, pages 405–414. MIT Press, Cambridge, MA, 2004.

[43] R. M. Turner. The tragedy of the commons and distributed AI systems. In *in Proceedings of the 12th International Workshop on Distributed Artificial Intelligence*, pages 379–390, 1993.

[44] I. Wagner and A. Bruckstein. Row straightening via local interactions. Technical report, Center for Intelligent Systems, Technion, Haifa, 1994.

[45] I. Wagner, M. Lindenbaum, and A. Bruckstein. Distributed covering by ant-robots using evaporating traces. *IEEE Transactions on Robotics and Automation,*, 15(5):918–933, 1999.

[46] I. A. Wagner, Y. Altshuler, V. Yanovski, and A. M. Bruckstein. Cooperative cleaners: A study in ant robotics. *International Journal of Robotics Research*, 27(1):127–151, 2008.

[47] I. A. Wagner and A. M. Bruckstein. From ants to a(ge)nts: A special issue on ant-robotics (editorial). *Annals of Mathematics and Artificial Intelligence*, 31(1–4):1–5, 2001.

[48] V. Yanovski, I. A. Wagner, and A. M. Bruckstein. Vertex-ant-walk: A robust method for efficient exploration of faulty graphs. *Annals of Mathematics and Artificial Intelligence*, 31(1–4):99–112, 2001.

[49] P. Zebrowski, Y. Litus, and R. T. Vaughan. Energy efficient robot rendezvous. In *CRV '07: Proceedings of the Fourth Canadian Conference on Computer and Robot Vision*, pages 139–148, Washington, DC, USA, 2007. IEEE Computer Society.