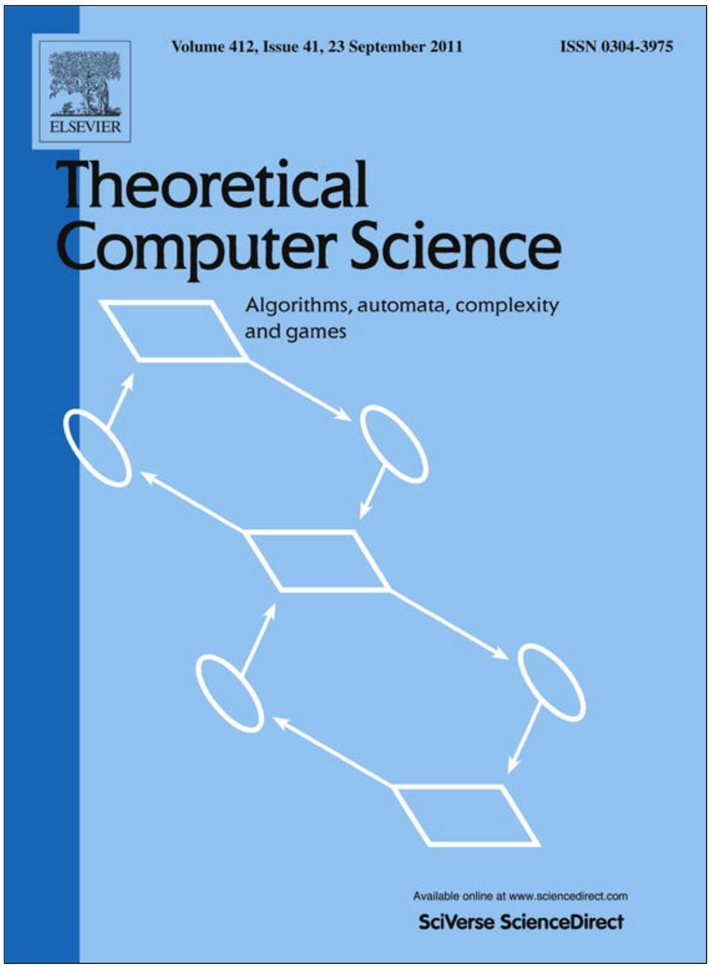
Volume 412, Issue 41, 23 September 2011
ISSN 0304-3975
ELSEVIER
Theoretical Computer Science
Algorithms, automata, complexity and games
Available online at www.sciencedirect.com
SciVerse ScienceDirect

Theoretical Computer Science 412 (2011) 5771–5788

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Review

Of robot ants and elephants: A computational comparison

Asaf Shiloni [*], Noa Agmon, Gal A. Kaminka

The MAVERICK Group, Computer Science Department, Bar Ilan University, Israel

A R T I C L E   I N F O

Article history:
Received 19 July 2010
Received in revised form 27 April 2011
Accepted 22 June 2011
Communicated by G. Ausiello

A B S T R A C T

In the robotics community, there exist implicit assumptions concerning 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 this paper, 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.

Contents

[*] Corresponding author. Tel.: +972 52 2234470.
E-mail addresses: shilona@cs.biu.ac.il, asafshiloni@gmail.com (A. Shiloni), segaln@cs.biu.ac.il (N. Agmon), galk@cs.biu.ac.il (G.A. Kaminka).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.06.026

1. Introduction

Investigations of multi-robot systems, from a computational perspective, often focus on algorithms for specific tasks and applications [1,2]. 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.

This paper defines two important computational classes of robots: *robot ants* and *robot elephants*. We examine the computability of these classes. 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.

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 [3,4] which often share some computation restrictions with ants, but similarly to elephants, do not have pheromones. We restrict our focus herein to ants and elephants.

Robot *ants* [5–8] are usually memoryless (or have severe memory limitations) and have relatively weak sensing abilities, if any [9,10]. 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 [9–11] and foraging [5,12].

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., [1,2]).

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 paper, 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.

2. Background and related work

Ant robots are usually described as memoryless or more formally as finite state machines [10,7] 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 [9,6]. Common to all previous work is the assumption that the ants are not able to use conventional planning methods such as POMDPs [6]. 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 [5], heat [13], markings [9], RFID [14], etc., and are sometimes evaporative [7,13].

This usage of pheromones as an indirect communication through the modification of the environment is the very basis of stigmergy [15]. 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 [16,15]. Nevertheless, in this paper 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 [7] and non-evaporative [17] markings. While some of these pheromones are laid by the robots themselves [17], others are a part of the given workspace [10]. They considered simple robots with a bounded amount of memory [10,8] for their model of ant robots. Their works and additional works by Koenig et al. [18] 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 [3,4,19]. 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 [3,4]. We do not investigate these in this paper.

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 [20]. 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 [21] produced a model for comparing the power of robot based on sensory abilities, but did not address computational and memory differences.

Several papers investigated classes of semi-synchronous [22] and asynchronous [23–28] 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.

3. Definitions

In this section, we provide formal definitions of the ant and elephant models used throughout our work (Section 3). 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 (Section 4) 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 paper, we handle three types of grids: (1) ClearGrid — an infinite grid with no obstacles. (2) RectangleGrid — an infinite grid with a bounded-sized rectangular obstacles. (3) ObstacleGrid — an infinite grid with an unbounded-sized any-shape obstacles.

Grid decomposition of the environment is a well known approximation for problems solving of this kind [2,10]. 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 homogeneous; they all have the same capabilities, and run the same algorithm.
- *Localization:* Ants cannot recognize their location.
- *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.
- *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.

Actions:

- *Move:* in four directions, *north, south, east, west*. In other words, they can only move between neighboring cells.
- *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 paper, 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/written 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.

[1] Shiloni et al. [29] show how to modify algorithms to a sense radius of zero, where an ant must physically move to a cell in order to learn about its content.

To focus the comparison between the ant and the elephant models on issues rather than sensing (already handled by [21]), 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 homogeneous 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 4.2).
- *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"). As the ants, they can only move between neighboring cells.
- *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 [21]. 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.

4. Elephants imitating ants

In this section, we begin to compare between the computational power of the models, using a first algorithm (Section 4.1) that allows multiple elephants to execute an algorithm for multiple ants. Moreover, we show an algorithm for a group of a more restricted variant of elephants, which achieves the same property (Section 4.2).

4.1. The anteater

First, 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.

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

```
1: Initialize Map $M$ with current global position $(x, y)$.
2: Initialize pointer $p$ to point to first instruction in $\mathcal{A}$.
3: while $\mathcal{A}$ has not stopped do
4:   if step in $p$ is to write pheromone $l$ in location $(x, y)$ then
5:     write $l$ in $M(x, y)$
6:   else if step in $p$ is read pheromone $l$ from location $(x, y)$ then
7:     read value $l$ from $M(x, y)$
8:   else if step in $p$ is sense location $(x, y)$ then
9:     Sense location $(x, y)$ in space
10:  else if step in $p$ is calculate values $(z_0, \dots, z_n)$ then
11:    Simulate calculation of $(z_0, \dots, z_n)$
12:  Broadcast $M$ to all $r \in R$
13:  Update $M$ with maps received
14:  if step in $p$ is move to $(x, y)$ then
15:    Move to location $(x, y)$ in space
16:  Set $p$ to point to next instruction in $\mathcal{A}$
```

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 3, 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. [17], 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 guarantees 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 area will be complete.

The Mark-Ant-Walk algorithm is given below (Algorithm 2). This algorithm is called continuously by each 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 $x \leftarrow \mathrm{argmin}_{q \in R(r,2r,p)} \sigma(q)$
2: if $\sigma(p) \le \sigma(x)$ 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-Ant–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 Section 3, where N is the number of robots, then* $\text{LF-ANT}_N \trianglerighteq \text{ANT}_N$ *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. □

4.2. 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.

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:     $r \leftarrow 1$ {The number of robots traveling in the group}
4:     **while** $r < |R| - 1$ **do**
5:        Move within a clockwise spiral {recording movements}
6:        **if** $\exists$ robot $r_i$ in current location $(x, y)$ **then**
7:           Send $(-x, -y)$ to robot $r_i$
8:           $r \leftarrow r + 1$
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 $(\mathcal{A}, R, M)$
```

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 it own origin, and then return to it 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.[2]

Therefore, we can show that a group of N NF-Ants computationally dominate 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* $\text{NF-ANT}_N \trianglerighteq \text{ANT}_N$ *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 $NF\text{-}ANT_N \trianglerighteq ANT_N$. □

5. 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 5.1) and on an ObstacleGrid (Section 5.2). Then, we discuss two examples of problems that can be solved by both ants and elephants (Section 5.3) followed by two problems that can be solved by elephants, yet not by ants (Sections 5.4 and 5.5).

[2] The spiraling movement will work for ClearGrid, but for ObstacleGrid the NF-Ant leader needs to move in a memoryless Depth First Iterative Deepening (DFID) similarly to the movement in the General Obstacle Algorithm (GOA) in [29].

5.1. 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 LF-Ants 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:

- **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.
- **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.

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.[3]

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*.

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 toward the *memory position* it will not change any pointers, but follow the pointers that already exist.

We refer the reader to Appendix for a formal definition of the finite state machine ElephantGun, which simulates the execution of an elephant algorithm when being run by an ant.

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

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: $x \leftarrow x + 1$
```

An ant running ElephantGun is given the Turing machine representation of *ElephantGunExample* as an input. Fig. 1 shows the snapshot of the world 4 steps after the last line of the algorithm was executed (before the ant has 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 mark the direction in which the *physical position* is (here, the directions are all "west"). Once, finished simulating the first line it

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

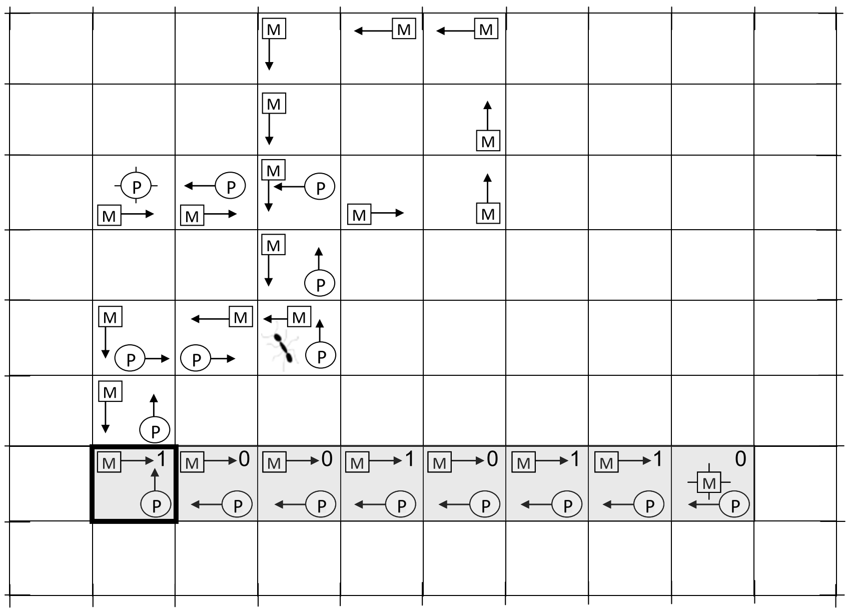

Fig. 1. An example of ElephantGun. The shaded region is the simulated Turing machine. *P* symbols are the *physical position* pointers. *M* symbols are the *memory position* pointers. The bold cell is the ant's starting location.

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 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, toward contradiction, that there is a set of pointers $X = (x_1, x_2, \ldots, x_n)$ pointing to head a such that $\forall i,\ i = 1] \ldots n,\ x_i \rightarrow x_{i+1}$ mod n (w.l.o.g), forming a loop. Thus, there exist no pointer x_i which points to the inside nor the outside of the loop. Also, none of x_i is the head itself, otherwise it would not be a loop. If the loop was created by head a itself, then the last pointer in the loop will replace the first one and will point toward head a and thus, break the loop. Otherwise, if the loop was created by head b, the first pointer in the loop will not be replaced and will still point toward head a. But head a is not a part of the loop, leading to a contradiction. □

Lemma 5. *At every instance in* ElephantGun *there is a path of pointers between the two heads in each direction (not necessarily the same path).*

Proof. Assume, toward contradiction, that there is no path of pointers from head a to head b (w.l.o.g). Thus, there exist at least one pointer in the path of pointers from a to b that does not lead to b. Since the two heads start from the same place and since there is no action of erasing, we can deduce that there was a path until the above pointer was replaced, and not by b. But, although both heads can add pointers, each of them can only replace its own 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 toward 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 4 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 $A = move, write, move, \ldots, write$ such that the movements and writing actions are toward opposite directions. Suppose that the ant's simulated tape is written from north to south and all moves in A are moves in the north 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.[4]

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 $\mathcal{O}(n)$ pheromones for a task that requires n steps for the original NF-Ant. $\square$

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 $NF\text{-}ANT_1$ can be solved by ANT_1. Thus, $ANT_1 \trianglerighteq NF\text{-}ANT_1$ on a ClearGrid. $\square$

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 $NF\text{-}ANT_N \trianglerighteq ANT_N$ for $N \geq 1$, then $NF\text{-}ANT_1 \trianglerighteq ANT_1$. Also, we have shown in Theorem 7 that $ANT_1 \trianglerighteq NF\text{-}ANT_1$. Therefore, $ANT_1 \equiv NF\text{-}ANT_1$ on a ClearGrid. $\square$

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.

5.2. 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 and therefore, might find itself with no empty cells to store new memory in them. Thus, assuming an infinite grid with an unknown number of obstacles which still maintain connectivity within the area (ObstacleGrid), in order to create a virtual Turing machine, we would like to have an algorithm that would iterate the grid in such a way which produces a new empty cell in each step. Fortunately, the General Obstacle Algorithm (GOA) from [29] 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.

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 empty 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, 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 bounded Depth First Search is being run repeatedly, such that in each run the depth limit is increasing by one. Therefore, a variable d must be maintained in order to decide when to backtrack (when we reach the current limit). 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 and could not increase anymore beyond a certain level. 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:

[4] 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.

- *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 when the Turing machine head is moving over the current cell.
- *direction***:** While the *parent* field at cell *c* points to the cell that the ant 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 4 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:  $state \leftarrow p$
3:  while true do
4:    if $current.direction = \text{NULL}$ then
5:      $current.direction = \text{EAST}$
6:    $last\_direction \leftarrow current.direction$
7:    $nbr = \text{sense}(current.direction)$
8:    if $nbr.parent = \text{NULL}$ then
9:      move($current.direction$)
10:     $current.parent, current.direction \leftarrow last\_direction + 180^\circ$
11:     break
12:   else if $nbr.parent = current.direction + 180^\circ$
      $\vee\ current.direction = current.parent$ then
13:     move($current.direction$)
14:     $current.direction \leftarrow current.direction + 90^\circ$
15:   else
16:     $current.direction \leftarrow current.direction + 90^\circ$
```

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 will first write 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 just toggles the *direction* field of the current cell clockwise (line 15).

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 toward 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 toward 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.

Algorithm 6 LeftMovement (symbol b, state p)

```
1:  write($b$)
2:  $state \leftarrow p$
3:  if $current.parent \neq$ TM_END then
4:     $last\_direction \leftarrow current.direction$
5:     $current.parent \leftarrow$ NULL
6:     move($last\_direction$)
7:     $first\_direction \leftarrow current.direction$
8:     while $true$ do
9:        $current.direction \leftarrow current.direction - 90^\circ$
10:       if $current.direction = first\_direction$ then
11:          break
12:          $nbr =$ sense($current.direction$)
13:       else if $nbr.direction = current.direction + 180^\circ$ then
14:          move($current.direction$)
15:          $first\_direction \leftarrow current.direction$
```

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.[5] 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 toward 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.

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 toward depth $d+1$. At that point, one needs to backtrack d steps toward the root of the tree and then $d+1$ steps toward 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 4 available actions is at the worst case as follows: $cost(move \rightarrow move) = 1$, $cost(write \rightarrow write) = 2w + 1$,

[5] 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.

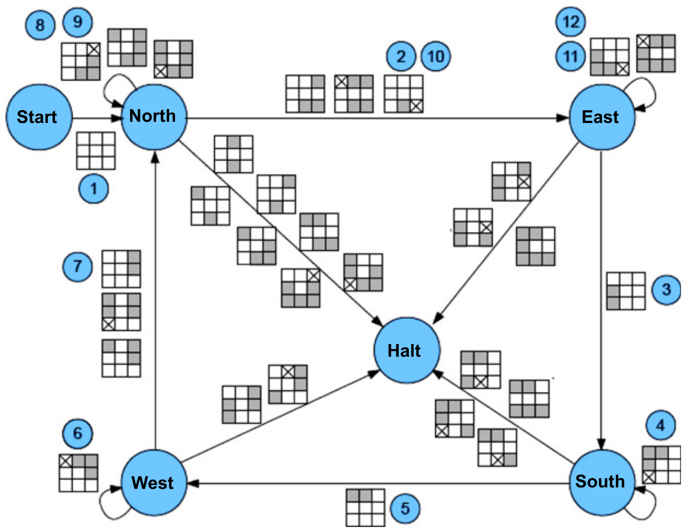

Fig. 2. NOA finite state machine representation.

$cost(move \rightarrow write) = m + 3w + 1$, 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 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 $\mathcal{O}(n)$ pheromones for a task that requires n steps for the original NF-Ant. □

5.3. 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 two-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 midpoint $p$ of the Manhattan distance between $(x_i, y_i)$ and $(x_{1-i}, y_{1-i})$
4: move toward $p$
```

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 that time complexity as $\mathcal{O}(m)$. Moreover, by Algorithm 3, an NF-Ant can simulate Rendezvous in Algorithm Manhattan. Nevertheless, there is also an ant algorithm which solves Rendezvous on a ClearGrid, called the No-Obstacle Algorithm (in [29]), using a state machine to create a spiral and to meet the other ant (see Fig. 2).

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 toward the other ant.

As we can see in [29], 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 [30].

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.

Algorithm 9 AntStretcher (subspace S)

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

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.

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 Section 5.4–5.5.

5.4. *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 strictly dominate N ants. In order to do this, we will define the following problem we call LimitedKServer.

Definition 8 (LimitedKServer)**.** Let $R = r_1, \ldots, r_N$ be a set of mobile robots with sensing radii of $\frac{M}{2N}$ each, all are positioned on a finite $M \times M$ grid such that all sensing radii are disjoint and their union covers the whole grid. Let $C = c_1, \ldots, c_x$ be a set of requests 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 requests are made such that no two requests are made in parallel. Assume that every robot $r_i \in R$ can handle a request immediately only within its sensing radius and that each request lasts for one time cycle only. We say that an algorithm $\mathcal{A}$ succeeds $\iff$ for every sequence of x requests, $\mathcal{A}$ handles exactly y. Otherwise, $\mathcal{A}$ fails.

Note that the above problem is a variant of a physical k-server [31] problem where there are a finite number of requests and the servers need to collectively handle only a certain number of these requests (see Fig. 3). This problem represents an abstraction of problems related to the tragedy of the commons such as overfishing [32]. 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.

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

Proof. Let X be the finite set of requests. Since the N NF-Ants cover the $M \times M$ entirely, there exist no request that is overlooked by all of the NF-Ants. Also, since each NF-Ant reigns over a disjoint territory, no request is being handled by two NF-Ants. In addition, since an NF-Ant is only occupied for one cycle per request and no two requests are made in parallel, there is no situation in which an NF-Ant is occupied and cannot handle a new request. Thus, after y requests the herd of NF-Ants have handled exactly y requests and therefore, all NF-Ants break from the while loop upon receiving the yth 'REQUEST' 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. □

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

```
1: $requests \leftarrow 0$
2: while $requests < y$ do
3:   if a message ‘REQUEST’ has been received then
4:     $requests \leftarrow requests + 1$
5:     if $requests = y$ then
6:       break
7:   else if $\exists$ request $c$ within sensing radius then
8:     broadcast ‘REQUEST’ to all $r \in R$
9:     $requests \leftarrow requests + 1$
10:    handle request
```

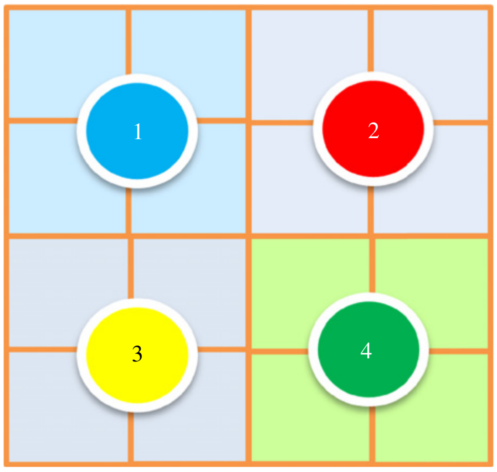

Fig. 3. An illustration of LimitedKServer with four robots.

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 request 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.*

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 toward 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 handle the $(y + 1)$th request for it cannot know that there were y requests 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 requests to handle 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, $NF\text{-}ANT_N \rhd ANT_N$, 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.

5.5. Duplicate robot patrol problem

The LimitedKServer problem might strike the reader as just a latency problem, i.e, if the requests were to hold a bit longer then the ants might have been able to both handle them all and move around the area in order to propagate the total number of requests. Therefore, we would like to introduce another counter example, 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 [2,33,34], 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 robots, each having a unique *id*. Therefore, we assume that if a robot enters the area and there is another robot already in the area with the same *id*, then the entering robot is a duplicate robot 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 robot crosses the area's perimeter.

We define the Duplicate problem formally as follows:

Definition 9 (Duplicate)**.** Let $R = r_1, \dots, r_N$ be a set of mobile robots which are positioned on a perimeter fence p, which surrounds a secluded area, and let $G = g_1, \dots, g_M$ be a set of mobile robots, which move about the area while trying to enter and leave the secluded area by crossing p. Each of the robots is having a not necessarily unique *id* marked $g_i.id$. Let g_i and g_j be two robots such that $g_i.id = g_j.id$ and assume g_i crossed p at time t_i. Then, robot 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 robot g_{dup} advances toward 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 [34,33] as a baseline. We propose the following DuplicateFinder algorithm for elephants (Algorithm 11).

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

```
1:  robot_list ← NULL
2:  run PA
3:  while no duplicate found do
4:    if received 'DUPLICATE' message then
5:      break
6:    if received robot.ID message then
7:      add robot.ID to robot_list
8:    if encountered robot then
9:      if robot.ID ∈ robot_list then
10:       broadcast 'DUPLICATE' to all r ∈ R
11:       break
12:     else
13:       add robot.ID to robot_list
14:       broadcast robot.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 robots and runs a given patrol algorithm. Each time an NF-Ant encounters an robot (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 robot (line 6). Therefore, suppose there exist an robot g_1 that approaches NF-Ant r_1 at time t and a corresponding duplicate robot 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 robot g_{dup} approaches r_2 at time $t' < t + P/2$ then it cannot be spotted as a duplicate and the algorithm will fail. □

Table 1
Models Dominance Relationship.

Model dominance	Algorithm	Time complexity
(1) $LF\text{-}ANT_N \trianglerighteq ANT_N, N \geq 1$	AntEater	$\mathcal{O}(n)$
(2) $NF\text{-}ANT_N \trianglerighteq ANT_N, N \geq 1$	NFantSpiral	$\mathcal{O}(n)$
(3) $ANT_1 \trianglerighteq NF\text{-}ANT_1$	ObstacleGun	$\mathcal{O}(n^2)$
(4) $ANT_1 \equiv NF\text{-}ANT_1$	(2) and (3)	
(5) $ANT_N \ntrianglerighteq NF\text{-}ANT_N, N > 1$	LimitedKServer	Counter Example
(6) $NF\text{-}ANT_N \triangleright ANT_N, N > 1$	(2) and (5)	

6. Conclusions

It was proposed that ant robots can perform difficult computational tasks despite their weak computational abilities [10]. 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. 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 1.

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 [21]), 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.

Appendix

An NF-Ant algorithm is a Turing machine ElephantAlgorithm such that:

$$\text{ElephantAlgorithm} = (Q, \Sigma, c, \Gamma, \delta, s, F)$$

where Q is the set of states, Σ is the input's alphabet, c 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 [20]), such that:

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

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; also transitions δ' will be 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 5 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, $\Sigma' = \Sigma$. Assume (w.l.o.g), that the ElephantGun simulates the Turing Machine tape from east to west.

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 head
- $q_{setmem(W)}$ — an intermediate state to update the current slot as the memory head
- $q_{setloc(Z)}$ — an intermediate state to update the current slot as the robot's location
- q_{find} — an intermediate state to find the robot's location
- $q_{find(Z)}$ — an intermediate state to find the robot's location and move one slot to $Z \in \{E, W, S, N, H\}$.

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

- $q_{a,b,q'',E}$ — an intermediate state to find the memory head's location and perform the $(q, a) \rightarrow (q'', b, E)$ transition
- $q_{a,b,q'',W}$ — an intermediate state to find the memory head's location and perform the $(q, a) \rightarrow (q'', b, W)$ transition.

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

- $(q, (a, y, z)) \to (q_{a,b,q\text{''},E}, (a, y, z), y)$ — for the case that the ant is not on the memory head
- $(q, (a, H, z)) \to (q\text{''}_{setmem(E)}, (b, E, z), E)$ — for the case that the ant is on the memory head.

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

- $(q_{a,b,q\text{''},E}, (a, y, z)) \to (q_{a,b,q\text{''},E}, (a, y, z), y)$ — continue searching the memory head in the pointed direction
- $(q_{a,b,q\text{''},E}, (a, H, z)) \to (q\text{''}_{setmem(E)}, (b, E, W), E)$ — found memory head, process transition, and move to the east
- $(q_{setmem(E)}, (a, \sqcup, \sqcup)) \to (q, (a, H, W), S)$ — update memory head pointer to “here” and pointer to physical head
- $(q_{setmem(E)}, (a, y, z)) \to (q, (a, H, z), S)$ — update memory head pointer to “here”.

Likewise, every transition in the form $(q, a) \to (q\text{''}, b, W)$ will be replaced by the following transitions:

- $(q, (a, y, z)) \to (q_{a,b,q\text{''},W}, (a, y, z), y)$ — for the case that the ant is not on the memory head
- $(q, (a, H, z)) \to (q\text{''}_{setmem(W)}, (a, W, z), W)$ — for the case that the ant is on the memory head.

Also, we will add the following transitions:

- $(q_{a,b,q\text{''},W}, (a, y, z)) \to (q_{a,b,q\text{''},W}, (a, y, z), y)$ — continue searching the memory head in the pointed direction
- $(q_{a,b,q\text{''},W}, (a, H, z)) \to (q\text{''}_{setmem(W)}, (b, W, E), E)$ — found memory head, process transition, and move to the east
- $(q_{setmem(W)}, (a, \sqcup, \sqcup)) \to (q, (a, H, E), S)$ — update memory head pointer to “here” and pointer to physical head
- $(q_{setmem(W)}, (a, y, z)) \to (q, (a, H, z), S)$ — update memory head pointer to “here”.

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

- $(q, (a, y, z)) \to (q_{find(Z)}, (a, y, z), z)$ — for the case that the ant is not on the physical robot head
- $(q, (a, y, H)) \to (q_{setloc(Z)}, (a, y, Z), Z)$ — for the case that the ant is on the physical robot head.

Together with the following new transitions:

- $(q_{find(Z)}, (a, y, z)) \to (q_{find(Z)}, (a, y, z), z)$ — continue searching the physical robot head in the pointed direction
- $(q_{find(Z)}, (a, y, H)) \to (q_{setloc(Z)}, (a, y, Z), Z)$ — found physical robot head, update pointer, and move to the desired direction $Z \in \{E, W, S, N\}$
- $(q_{setloc(Z)}, (a, \sqcup, \sqcup)) \to (q, (a, \overline{Z}, H), S)$ — update physical robot head pointer to “here” and memory head pointer to where you came from
- $(q_{setloc(Z)}, (a, y, z)) \to (q, (a, y, H), S)$ — update physical robot head pointer to “here”.

And lastly, for any action or sensing need to be done while the ant is in its own physical location:

- $(q, (a, y, z)) \to (q_{find}, (a, y, z), z)$ — for the case that the ant is not on the physical robot head
- $(q, (a, y, H)) \to (q, (a, y, H), S)$ — for the case that the ant is on the physical robot head.

Together with the following new transitions:

- $(q_{find}, (a, y, z)) \to (q_{find}, (a, y, z), z)$ — continue searching the physical robot head in the pointed direction
- $(q_{find}, (a, y, H)) \to (q, (a, y, H), S)$ — found physical robot head, the ant can sense or act.

References

[1] N. Hazon, F. Mieli, G.A. Kaminka, Towards robust on-line multi-robot coverage, in: ICRA, 2006.
[2] Y. Elmaliach, N. Agmon, G.A. Kaminka, Multi-robot area patrol under frequency constraints, in: ICRA, 2007.
[3] E. Şahin, Swarm robotics: from sources of inspiration to domains of application, in: Swarm Robotics: SAB 2004 International Workshop, in: Lecture Notes in Computer Science, vol. 3342, Springer, 2005, pp. 10–20.
[4] A.J. Sharkey, Robots, insects and swarm intelligence, Artificial Intelligence Review 26 (4) (2006) 255–268.
[5] R. Russell, Ant trails: an example for robots to follow? in: ICRA, vol. 4, 1999, pp. 2698–2703.
[6] I.A. Wagner, 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) (2001) 1–5.
[7] I. Wagner, M. Lindenbaum, A. Bruckstein, Distributed covering by ant-robots using evaporating traces, IEEE Transactions on Robotics and Automation 15 (5) (1999) 918–933.
[8] V. Yanovski, I.A. Wagner, A.M. Bruckstein, Vertex-ant-walk: a robust method for efficient exploration of faulty graphs, Annals of Mathematics and Artificial Intelligence 31 (1–4) (2001) 99–112.
[9] S. Koenig, Y. Liu, Terrain coverage with ant robots: a simulation study, in: Autonomous Agents, ACM, 2001, pp. 600–607.
[10] I.A. Wagner, Y. Altshuler, V. Yanovski, A.M. Bruckstein, Cooperative cleaners: a study in ant robotics, International Journal of Robotics Research 27 (1) (2008) 127–151.
[11] A. Sempe, F. Drogoul, Adaptive patrol for a group of robots, in: IROS, vol. 3, 2003, pp. 2865–2869.

[12] T.H. Labella, M. Dorigo, J.-L. Deneubourg, Division of labor in a group of robots inspired by ants' foraging behavior, ACM Transactions on Autonomous Adaptive Systems 1 (1) (2006) 4–25.
[13] R. Russell, Heat trails as short-lived navigational markers for mobile robots, in: ICRA, vol. 4, 1997, pp. 3534–3539.
[14] M. Mamei, 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, ACM, 2005, pp. 1353–1354.
[15] G. Theraulaz, E. Bonbeau, A brief history of stigmergy, Artifical Life 5 (2) (1999) 97–116.
[16] O. Holland, C. Melhuish, Stigmergy, self-organization, and sorting in collective robotics, Artifical Life 5 (2) (1999) 173–202.
[17] E. Osherovich, A.M. Bruckstein, V. Yanovski, Covering a continuous domain by distributed, limited robots, in: ANTS Workshop, 2006, pp. 144–155.
[18] S. Koenig, B. Szymanski, Y. Liu, Efficient and inefficient ant coverage methods, Annals of Mathematics and Artificial Intelligence 31 (1–4) (2001) 41–76.
[19] R. Cohen, D. Peleg, Local spreading algorithms for autonomous robot systems, in: structural Information and Communication Complexity, SIROCCO 2006, Theoretical Computer Science 399 (1–2) (2008) 82.
[20] M. Sipser, Introduction to the Theory of Computation, International Thomson Publishing, 1996.
[21] J.M. O'Kane, S.M. LaValle, On comparing the power of robots, International Journal of Robotics Research 27 (1) (2008) 5–23.
[22] I. Suzuki, M. Yamashita, Distributed anonymous mobile robots: formation of geometric patterns, SIAM Journal on Computing 28 (1999) 1347–1363.
[23] G. Prencipe, CORDA: Distributed coordination of a set of autonomous mobile robots, in: Proc. 4th European Research Seminar on Advances in Distributed Systems, 2001, pp. 185–190.
[24] 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, 2001, pp. 185–190.
[25] R. Klasing, E. Markou, A. Pelc, Gathering asynchronous oblivious mobile robots in a ring, Theoretical Computer Science 390 (1) (2008) 27–39.
[26] J. Czyzowicz, L.G. Cedilla]sieniec, A. Pelc, Gathering few fat mobile robots in the plane, in: principles of Distributed Systems, Theoretical Computer Science 410 (6–7) (2009) 481–499.
[27] M. Yamashita, I. Suzuki, Characterizing geometric patterns formable by oblivious anonymous mobile robots, Theoretical Computer Science 411 (26–28) (2010) 2433–2453.
[28] P. Flocchini, D. Ilcinkas, A. Pelc, N. Santoro, Remembering without memory: tree exploration by asynchronous oblivious robots, in: structural Information and Communication Complexity, SIROCCO, Theoretical Computer Science 411 (14-15) (2008) 1583–1598.
[29] A. Shiloni, A. Levy, A. Felner, M. Kalech, Ants meeting algorithms, in: AAMAS, 2010, pp. 567–574.
[30] A.S. Fukunaga, A.B. Kahng, Cooperative mobile robotics: antecedents and directions, Autonomous Robots 4 (1997) 226–234.
[31] M.S. Manasse, L.A. McGeoch, D.D. Sleator, Competitive algorithms for server problems, Journal of Algorithms 11 (2) (1990) 208–230.
[32] R.M. Turner, The tragedy of the commons and distributed AI systems, in: Proceedings of the 12th International Workshop on Distributed Artificial Intelligence, 1993, pp. 379–390.
[33] Y. Elmaliach, A. Shiloni, G.A. Kaminka, A realistic model of frequency-based multi-robot fence patrolling, in: AAMAS-08, vol. 1, 2008, pp. 63–70.
[34] N. Agmon, S. Kraus, G.A. Kaminka, Multi-robot perimeter patrol in adversarial settings, in: ICRA-08, 2008.