Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Adversarial ANTS

Master's Thesis

David N. Stolz

stolzda@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Tobias Langner, Jara Uitto
Prof. Dr. Roger Wattenhofer

December 2, 2013

Acknowledgements

First of all, I would like to thank Luchin Doblies for his elaborate support, input and comments. Also, I would like to thank my supervisors for their advice and support. Furthermore, I would like to thank Yannick Rothacher, Lukas Rothacher and Anatol Schauwecker for their valuable insights on this thesis from a biological perspective.

Finally, I am grateful for the numerous colleagues, that made all of this much more interesting. I can obviously only mention a very incomplete set, but certainly included are: Thomas Bürli, Alexander Grest, Fabio Heer, Moritz Hoffmann, Andreas Marfurt, Jonas Pfefferle, Oliver Probst, Philipp Schmid, David Sommer, Pascal Studerus, Andreas Tschofen, and Simon Wright.

Abstract

We study the *Ants Nearby Treasure Search (ANTS)* problem that has been introduced by Feinerman, Korman, Lotker, and Sereni (PODC 2012). The ANTS problem consists of n ants, initially placed at the origin of the infinite grid, which have to collaboratively locate food that is placed by an adversary. We follow the restrictions on the ants' computational power introduced in the paper of Emek, Langner, Uitto, and Wattenhofer (2013). In particular, the ants are controlled by a randomized finite state machine and have the ability to locally exchange constant-size messages.

In this thesis, we extend the previous work by studying which fundamentals a nature inspired problem must comprise; we argue that the two most crucial features of such a problem are the need for collaboration, and the element of uncertainty. Based on those insights, we introduce the *Adversarial ANTS (AANTS)* problem that extends the existing model by a need for fault-tolerance.

We provide a lower bound on the number of ants that are required to successfully locate the treasure, assuring that ants indeed have to collaborate. To the best of our knowledge, this is the first work in regard to the ANTS problem that actually proves the necessity for collaboration. We also establish a lower bound on the runtime of the AANTS problem, and we present an algorithm that is $o(\log(n))$-competitive w.h.p., where n is the number of ants. Additionally, we investigate the implications of making ants computationally more powerful, by allowing them to store information on an unbounded stack. Our results show that a single ant with a stack is computationally equivalent to one without a stack.

Contents

Chapter 1

Introduction

1.1 Biology Meets Theoretical Computer Science

In recent years, interdisciplinary research has gained increased attention. Traditionally, the collaboration between different fields of science has been unilateral; for example medical scientists using findings of biology, pharmacy profiting from advances in chemistry, and of course, nearly every field of study benefiting from mathematics. Even though this unilateral collaboration accelerates research for one of the parties, it would be even better if scientists collaborated in such a way, that **both** parties can benefit mutually. Of course, this is not an easy task, as it requires a certain problem to be relevant and unsolved for both areas of research; nevertheless, certain areas of research turn out to be well suited for such a fruitful collaboration. One of the maybe more surprising conjunctions is the one of biology and theoretical computer science, which motivated this thesis.

The question of how animals solve complex tasks is fundamental for behavioral biology. Of particular interest are tasks which include an inherent unpredictable element that has a significant impact on how the problem can be solved, or tasks that are solved with collaboration. It is well-known that there exist many different species of ants that collaboratively manage a colony, which includes various jobs to be taken care of; those jobs range from breeding of offspring over cleaning the hive to foraging. In this thesis we investigate the **ant foraging process**, as it is a good example of a task that features both the element of collaboration and the element of unpredictability.

From a theoretical computer science point of view, the ant foraging process resembles a typical distributed problem. It relates to various multi-robot problems, such as *gathering* a team of robots, or *graph exploration*. The ant foraging problem itself raises a number of questions: How much computational power does each individual ant need to have, so that foraging can be done efficiently? Can communication between the ants benefit their searching behavior, and what form of communication do the ants require? Are messages of constant size sufficient, or do they need to exchange, e.g., positional information?

Thinking about the ant foraging process, it quickly becomes evident that the problem is a fine example of a topic that suits our demand for mutual interdisciplinary research. Hence, we decided to extend the existing work on ant foraging, by modifying the model introduced by Emek et al. [1] in such a way, that it resembles nature more closely. We decided to add an element of uncertainty to the ant foraging process; namely we added an adaptive adversary that is allowed to fail ants during the process. We claim that such an extension to the model allows us to get an important step closer to nature, as the element of uncertainty lies at the heart of problems that are solved in nature; in particular, a major problem of algorithms that are designed in the explicit absence of uncertainty is, that they are prone to miss the essence of the actual problem.

1.2 Related Work

The main inspiration of this thesis was the work from Emek et al. [1] who studied the *Ants Nearby Treasure Search (ANTS)* problem, which was introduced by Feinerman et al. in [2, 3]. Feinerman et al. studied the ant foraging process from a theoretical computer science point of view, by asking the question of how the foraging problem can be solved efficiently. In their model, ants are represented by randomized turing machines, that initially start at a hive, and collaboratively discover the area around the hive to find the *treasure* – i.e., the food – located by an adversary. The goal is to find the treasure as fast as possible. Emek et al. adopted this model, but they argue that modeling ants to have unbounded memory might not be an accurate model of ants; instead, they show that the ANTS problem can be solved with finite memory, if ants are allowed to make use of a very simple form of communication. Lenzen and Radeva [4] encourage this claim, presenting an algorithm which requires only finite memory and the possibility to mark locations with pheromones.

Other work that has influenced this thesis can be grouped roughly into five categories: Graph exploration, swarm robotics, fault tolerance, interdisciplinary research and studies on ants. In the following we point out the related work of those five categories.

Graph exploration has been studied thoroughly; i.e., graph exploration by a single agent is studied in [5, 6, 7]. Of particular relevance for this thesis is the restriction to grids. An extensive survey of the recent work on grid exploration by a single agent can be found in Wernli [8]. Since we are modeling a natural environment, it might be reasonable to consider obstacles that hinder the movement of ants; how a single agent can discover an environment containing obstacles was studied, e.g., in the work of Betke et al. [9]. In respect to memory limitations, the use of only logarithmic memory has been studied in [10] and the use of no memory at all in [11]. How a pebble (which could correspond to pheromones) can be used to explore a graph is studied in [12].

Multi-agent graph exploration has also been studied in various settings; Bender and Slonim show in [13] how two agents can collaboratively explore a graph. The impact of communication on agent collaboration in general is studied in [14]. Graph exploration with an arbitrary number of agents is studied for trees, e.g., in [15, 16, 17, 18] and for general graphs in [19, 20, 21]. Multi-agent grid exploration with obstacles has been studied in a recent work of Ortolf and Schindelhauer [22].

Solving tasks with a large number of robots has been studied, for example, in the context of *swam robotics* [23]. Covering a finite graph in optimal time is closely related to the ANTS problem; a survey on different approaches to the problem can be found in [24]. The coverage problem is studied for a swarm of robots in [25], and for ant like robots in [26].

Fault tolerance in multi-agent systems has been studied in different contexts. Examples include: Fault-tolerance in actual robot systems [27], fault-tolerant robot gathering [28], fault-tolerant sorting with ant-like robots [29] and fault-tolerance in population protocols [30].

The idea of bringing biology and theoretical computer science closer is not new; ideas in such a direction can already be found in a survey from Wagner and Bruckstein [31]. A more recent survey is the one from Feinerman and Korman [32]. The idea of applying algorithmic concepts to problems from biology is also studied in the work of Chazelle, e.g., in the context of bird flocks [33].

A more mathematical point of view on ants can be found in [34]. In a similar fashion, Hirsh and Gordon analyze the collaboration in various types of insects in [35]. Very early approaches to simulating ant behavior based on observing ants can be found in [36, 37].

There exist many papers from biology on ants. Special attention deserves the work from Wehner et al., e.g., [38, 39, 40, 41]. The navigation capabilities of ants were studied in [37, 41, 42, 43, 44, 45, 46, 47, 48, 49]. Those studies present a lot of evidence, that suggests that ants make use of different navigational abilities, which can be grouped into two main categories: *Path-integration* and *landmark information*.

1.3 Thesis Outline

In Chapter 2 we start with introducing the main concepts used throughout this thesis. We also give a formal description of the used model, and introduce the *Adversarial Ants Nearby Treasure Search (AANTS)* problem. Then, in Chapter 3, we establish a lower bound on the number of ants required to solve the ANTS problem, giving insight on the computational power of ants in our model; in particular, we show that it requires at least three ants to discover the treasure, which shows that the chosen model features an inherent element of collaboration.

In Chapter 4 we proceed to investigate the AANTS problem, establishing a lower bound on the expected runtime, and presenting algorithms that solve the AANTS problem efficiently. Our main result is an algorithm with an $o(\log(n))$-competitive expected runtime w.h.p., where n is the number of ants.

In Chapter 5, we alter the model, studying the impact of making the ants more powerful, i.e., by allowing them to store information on a stack. We show that, even with this enhanced computational power, ants still need to collaborate in order to discover the treasure.

In Chapter 6, we compare the results with existing work on ants from biology. We conclude this thesis in Chapter 7, summarizing our contributions and pointing out possible directions of future work.

Chapter 2

Model

In this thesis, we investigate the ANTS problem introduced in the paper by Feinerman et al. [2] in different variants. Throughout this thesis, we make changes to the studied model, which are discussed in the respective chapters. In this chapter, we introduce the "basic" model, i.e., the problem statement and the basic computational model. Note that all the models used throughout this thesis are closely related to the model introduced by Emek et al. [1]. Additionally, we introduce the *Adversarial Ants Nearby Treasure Search* (AANTS) problem, which we will extensively study in the later chapters.

2.1 Preliminaries

Before we discuss the model that is used in this thesis, we introduce certain concepts from graph theory. The definitions of this section are illustrated in Figure 2.1.

Definition 2.1 (Manhattan distance) *For two points* $(x, y), (x', y') \in \mathbb{Z}^2$*, the* Manhattan distance *is defined as* $d((x, y), (x', y')) := |x - x'| + |y - y'|$. ◇

Definition 2.2 (infinite grid) *The* infinite grid $G^\infty = (V, E)$ *is a graph with* $V = \mathbb{Z}^2$*, containing an edge from* u *to* v *if and only if* $d(u, v) = 1$*, according to the* Manhattan distance. ◇

Remark 2.3 (field) *Throughout this thesis we refer to the vertices of the infinite grid as* fields*, which is a natural term when studying grids.*

Definition 2.4 (adjacent, neighboring) *Two fields* $u, v \in \mathbb{Z}^2$ *are called* adjacent *or* neighboring*, if and only if* $d(u, v) = 1$. ◇

Definition 2.5 (cardinal directions) *For a field* $(x, y) \in \mathbb{Z}^2$*, the four adjacent fields are referred to by their cardinal direction as follows:* North $(x, y + 1)$*,* East $(x + 1, y)$*,* South $(x, y - 1)$ *and* West $(x - 1, y)$. ◇

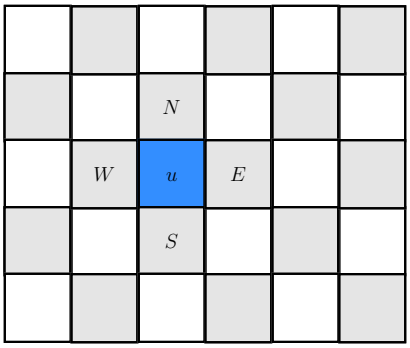


Figure 2.1: A part of the infinite grid G^∞. The four fields that are adjacent to field u are indicated with the letter for their respective cardinal direction.

2.2 General Model

The ANTS problem is specified as follows: A team of n identical mobile agents (referred to as *ants*) search for a treasure that was placed on the infinite grid G^∞ by an adversary. The goal of the search is to *discover* the field, on which the treasure is located, as fast as possible. We call a field *discovered*, when the field has been visited by at least one ant.

The search process is executed in *synchronous rounds*, and we say that each round lasts for one time unit. In the beginning (at time 0), all n ants are located at the *hive*, which is located at field $(0, 0) \in \mathbb{Z}^2$. All fields of the grid are identical, i.e., ants cannot distinguish them from each other. This means in particular, that an ant cannot derive any information from its current position; only an external observer would see on which field an ant currently stands. To facilitate navigation, all ants share the same sense of orientation; this means that the cardinal directions are consistent among all ants.

The Manhattan distance is used as the distance metric. The movement of ants is restricted to traversing at most a distance of 1 in each round, meaning that in each round, every ant can either move to an adjacent field, or stay put. Those five movement possibilities are abbreviated with N, E, S, W, P, where P denotes remaining stationary.

In this thesis, we assume that the ants are either *(randomized) finite state machines*, or *(randomized) pushdown automata*; which of the two computational models is in use is stated in the respective chapters. Note that in both cases, ants have, at any time, a current state. In the very same way as in the work of Emek et al. [1], we allow the ants to make use of a simple form of communication, which can be referred to as making *observations*. The observations that an ant can perform are limited to the following: In each round, every ant observes the presence of other ants on the same field. Note that it only gets very limited

information; namely for every state q, it observes if there is either no other ant currently present in state q, or if there are one or more ants present in state q. Hence it observes boolean information for every state q. This communication scheme is a special case of the one-two-many communication scheme introduced in [50].

Based on the observations and the computational model, the ants then proceed to compute their movement decision and their state transition. In the case where ants are pushdown automata, the ants may also alter the stack. We defer the definition of how the stack may be altered to Chapter 5, as for now, the ants are finite state machines.

In summary, every ant performs the following three steps in each round.

1. **Observe**. The ant observes other ants that are currently standing on the same field.
2. **Compute**. The ant performs a computation based on its state and the observed ants.
3. **Move and update state**. Based on the computation, the ant decides how to move. Additionally, the ant can update its state.

Regarding the computational power of the ants, it is crucial to note that their limitation to finite state machines or pushdown automata does not allow the ants to either have identifiers, nor to store their current coordinate[1]. Let us now formally define the protocol of an ant that has the computational power of a finite state machine[2]. The protocol of an ant is defined by the 3-tuple

$$M = (Q, q_0, \delta)$$

where Q is the finite set of states, q_0 is the initial state, and δ is the transition function. Initially, all ants are in state q_0; furthermore, all ants are positioned at $(0, 0)$, i.e., they are located at the hive. The transition function is defined as $\delta : Q \times 2^Q \to Q \times 2^{Q \times \{N,E,S,W,P\}}$. For every time t and ant a, the transition decision is made as follows: Let $q \in Q$ be the current state of a at time t, and let $c \in \mathbb{Z}^2$ be the field at which a is positioned. The state q' of a at time $t+1$ and the movement step $\tau \in \{N, E, S, W, P\}$ are chosen uniformly at random from pairs $(q', \tau) \in \delta(q, Q_a)$, where $Q_a \subseteq Q$ contains state $p \in Q$ if and only if there is at least one ant $a' \neq a$ in state p located at c at time t.

[1] Pushdown automata can store their coordinate, but they cannot access or alter it in a way a turing machine could. This limits the benefit of storing the coordinate on a stack heavily.

[2] The formal definition of an ant that has the computational power of a pushdown automaton can be found in Chapter 5.

2.3 The Adversarial ANTS Problem (AANTS)

Based on the model for the ANTS problem defined in the previous section, we extend the problem by adding an adversary, who is allowed to *fail* ants during the search process. We call this problem the *Adversarial Ants Nearby Treasure Search* (AANTS) problem. Since the only change to the model is the strengthening of the adversary, we will now specify her abilities.

At the end of each round, i.e., when the state transition and the movement decision of all ants are computed, but just before they are executed, the adversary may fail a certain amount of the ants. The ants that are failed by the adversary are "removed" from the grid, and they cannot influence the alive ants anymore. This means in particular, that failed ants do not leave a corpse behind, i.e., alive ants can only detect the failure of an ant by "knowing" that there should be an ant on a certain field, but their observation tells them that there is no ant. Since the adversary can decide in each round which ants she wants to fail, knowing the protocol executed by the ants, the ants have to deal with a so called *adaptive adversary*.

The number of ants that the adversary can fail in total is limited to f, where $f < n$, i.e., at least one ant must survive. Besides this restriction, there is no other restriction; in particular, algorithms should be able to tolerate up to $c \cdot n$ many failures, for a constant $c < 1$. The number of ants that the adversary can fail in a single round is not further constrained; an adversary could, for example, fail f ants in a single round.

Note that – besides initially positioning the treasure – failing ants is the only capability of the adversary; she cannot, for example, control the random bits that get created by the individual ants. This limitation of the adversary is necessary, as neither finite state machines nor pushdown automata could perform symmetry breaking without randomness or identifiers. Without symmetry breaking, all ants would perform the very same behavior; and thus, the resulting execution of any algorithm would be as if there was only one ant. In Chapter 3 (resp. Chapter 5) we show, that a single ant cannot solve the ANTS problem, hence AANTS would not be solvable if the adversary could control the random bits.

The goal of AANTS is the same as the goal of ANTS, namely to discover the treasure located by the adversary as fast as possible. The difference is only in how the performance is measured: In the original ANTS problem, the performance is measured as the runtime expressed in terms of the number of ants n, and the distance between the hive and the treasure, D. In AANTS, the runtime is also expressed in terms of f, the number of ants that the adversary can fail. Notice that all three parameters (D, n and f) are unknown to the ants.

2.4 Other Failure Models

In this section we present different failure models one can think of, and we explain why we think that they do not lead to interesting algorithms. Recall that we already discussed in Section 2.3 why an adversary cannot be allowed to control the random bits, which would be one of the different failure models that come to mind.

2.4.1 Oblivious Adversary

One way in which the adversarial setting could be modified would be to reduce the power of the adversary. Instead of ants being failed by an adaptive adversary, that is allowed to make decisions with knowledge about the algorithm, one could think of an *oblivious adversary*; i.e., the f failures would occur randomly throughout the search process. Such an adversarial model can easily be motivated by events that occur randomly in nature; for example a large animal walking by and crushing a couple of ants, ants dying from age, or ants dying from hostile weather conditions.

A question that arises naturally when deciding between an oblivious and an adaptive adversary model is: Why decide for the weaker model, if you can design algorithms for the stronger model? Even though it seems straightforward to pick the stronger model – as long as the problem remains solvable – one must keep in mind, that picking the weaker model is potentially justifiable; a stronger model usually leads to more complex solutions, and if the constraints of the model are not adequately backed by the actual problem, it might be reasonable to weaken to model, in order to obtain nicer solutions.

However, if the goal is to develop an algorithm that guarantees to solve the problem (a so called Las Vegas algorithm), deciding for an oblivious adversary model usually does not provide any significant benefits. Additionally, an oblivious adversary model might even complicate the analysis, without giving algorithms more degrees of freedom. Considering these potential drawbacks, and the fact that algorithms designed for the stronger model also work in the weaker model, we decided to choose the adaptive adversary model.

2.4.2 Byzantine Failures

There is another prominent concept when thinking of failure models: *Byzantine failures*. A possible application of the byzantine failure model would be to allow the adversary to control a certain amount of ants. For example, she could let them perform incorrect movement steps or incorrect state transitions. On the first glance, such a failure model seems to be very reasonable, as in nature a certain degree of lack of precision will occur, and byzantine failures could nicely

model such an imprecision. The following example illustrates such a byzantine failure: An ant wants to walk a single field north, but accidentally walks two fields north, due to shortcomings in its movement precision.

Even though such failures might occur in nature, it can easily be argued that modifying the ANTS problem by adding byzantine failures will either force any algorithm to **always** use a redundancy of $\Omega(n)$ – meaning that every field must be discovered by $\Omega(n)$ many ants –, or the algorithm cannot guarantee to discover the treasure. The argument can be sketched as follows: If an algorithm does not use redundancy at all, the adversary could make the algorithm fail to discover the treasure by simply shifting the ant that would discover the treasure around it (see Figure 2.2). It is therefore clear, that any algorithm that guarantees to discover the treasure, must use a form of redundancy. Since ants may not be able to realize that a certain byzantine failure occurred, redundancy must always be employed, and not only after a failure triggered its necessity. To make the problem interesting, one would parametrize the number of byzantine failures, or the number of ants that perform byzantine failures; else the number of byzantine failures would be constant in regard to the number of ants, which is obviously not of interest. But since the algorithm on one hand would be required to deal with this (potentially large) amount of byzantine failures, while on the other hand the ants are not capable of counting the number of ants on a certain field, the algorithm would be forced to use a search strategy with a redundancy factor of $\Omega(n)$, to make sure that every field is discovered. Since employing such a heavy redundancy does not make good use of the large amount of ants, the resulting algorithms would not be interesting.

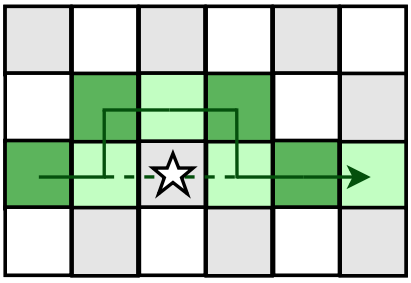


Figure 2.2: The star represents the treasure. The ant that should discover the treasure thinks that it walks the straight (dashed) line, but the adversary forces the ant to walk the solid line; the algorithm then assumes that the field with the treasure is discovered, and will not revisit it, hence failing to discover the treasure.

2.4.3 Location Bound Failures

Another variation of how failures could occur would be to have one or multiple adversaries who are themselves located on the grid. A possible motivation for such adversaries, who are bound to a certain location on the grid, could be anteaters, which eat all ants that step on the field on which the anteater is located. One could extend this model in various ways, e.g., by varying the number of adversaries, by allowing them to move, or by altering their size[3].

Let us discuss the simplest variation, the anteater with a fixed location. Such an anteater is equivalent to a *black hole*, a concept that is studied e.g., in [51, 52, 53]. In those papers, the following problem is addressed: There exists a graph and a group of mobile agents. The graph contains a black hole, i.e., one node of the graph is special in the sense that any mobile agent that steps on that node is extinguished. The goal is to traverse the graph with the mobile agents in such a way that the position of the black hole is learned.

The difference between the black hole search problem and the ANTS problem extended with anteaters is, that in ANTS it would not be sufficient to discover the black holes, but the ants would need to walk around them and still discover the treasure. In the following we will argue why the anteater failure model is not promising to yield interesting algorithms.

Having only a small (i.e., constant) number of static anteaters, the ants would easily be capable of walking around them, hence the anteaters would not have an interesting impact on the search process. Increasing the number of anteaters would at first not have any effect, as the anteater could always be avoided for the cost of sacrificing a handful of ants, until at a certain point the problem would become unsolvable, as simply too many ants die due to "discovering" the locations of the anteaters. A similar reasoning applies for allowing anteaters to move: They quickly become so powerful, that the ants cannot discover the treasure anymore. A different problem that arises is, that one must somehow limit the anteaters in the way that they can "defend" the treasure; for example they cannot be allowed to completely surround the treasure.

With those considerations in mind, we do not claim that there is no possibility that introducing a concept like anteaters might lead to interesting algorithms, but we think that justifying the exact parameters for the model (e.g., the number of anteaters) would be difficult. Therefore we did not study such a failure model.

[3] It seems reasonable to model the anteater with a different size than an ant, or allowing it to move faster than ants.

Chapter 3

Minimal Number of Ants to Discover the Grid

In this chapter we want to investigate the minimal number of ants that are required to deterministically find the treasure. The computational model for the ants is described in Chapter 2, but we do neither allow any failures in this chapter, nor the use of probabilistic bits. The latter restriction is due to the goal discovering the entire grid in a deterministic fashion, meaning that we want to have a guaranteed runtime to discover each field. It is clear that once the ants are discovering a distance far away from the hive, probabilistic bits will become useless as the number of fields to discover increases beyond any bounds – and the ants are not even aware of this situation due to their lack of memory. Thus the question that we pursue in this chapter can be stated as follows: *What is the minimal number of ants to deterministically discover the entire grid?* To answer this question, we want to establish the following theorem.

Theorem 3.1 *It takes at least three ants to deterministically discover the entire grid. Additionally this bound is tight, i.e., there exists an algorithm discovering the entire grid with three ants.*

In this chapter we assume that each of the ants has its own finite state machine. This alleviates the problem in that fashion, that we can have different roles; in the original model we could simulate this behavior with random bits, but as we decided to omit randomness in this chapter, we think it is clearer to allow the ants to have different finite state machines directly at the beginning. Note that this does not give the ants any advantage compared to ants in the original model.

3.1 One Ant

Lemma 3.2 *One ant cannot discover the entire grid deterministically.*

Proof Let us first make two important observations regarding the ant. First, as we are looking at a single finite state machine, communication cannot affect the behavior of the ant during the execution of the algorithm. And second, the number of states of the finite state machine is constant.

We use the following notation: q denotes the number of states of the finite state machine, and $\mathcal{P}(t)$ denotes the position of the ant at time t. We count the time starting from $t = 0$.

Let k ($1 \leq k \leq q$) be the first state that the ant enters a second time during the execution of the algorithm. We denote by t_{k_1} the first time where the ant is in state k, and analogously by t_{k_2} the second time and so on. Note that $t_{k_1} < q$ and $t_{k_2} < q + 1$.

As the ant decides in each time step for both the direction to move and a state transition based on solely its current state and any communication – and there is no communication – the ant will start repeating the exact movement steps and state transitions between any two t_{k_i} and $t_{k_{i+1}}$, for $i \geq 1$. Thus we can conclude that since the ant enters state k twice, it will enter state k an infinite number of times.

Note that for all $i \geq 1$ it holds that $t_{k_{i+1}} - t_{k_i} < q$, i.e., the time between the ant enters the state k again is bounded by a constant. This follows as there are at most $q - 1$ different states which the ant could enter, and since we know that the ant enters state k an infinite number of times, no different state can be entered twice between two consecutive t_{k_i}.[1]

Let us distinguish two cases:

1. $\mathcal{P}(t_{k_1}) = \mathcal{P}(t_{k_2})$

As the ant moves a cycle and ends up in the same configuration again (same position, same state), it will infinitely repeat performing the very same cycle. As we know that the time difference between two adjacent t_{k_i} and $t_{k_{i+1}}$ is bounded by a constant, the size of the cycle is also bounded by this constant. Thus the ant only walks over a constant number of fields and does not discover the entire grid.

[1] If there existed a state j that is entered twice between two consecutive t_{k_i}, the ant would perform the cycle between the two times in state j and never enter state k again, contradicting the fact that the ant enters state k an infinite number of times.

2. $\mathcal{P}(t_{k_1}) \neq \mathcal{P}(t_{k_2})$

Let $\Delta_k = \mathcal{P}(t_{k_{i+1}}) - \mathcal{P}(t_{k_i})$ be the movement that the ant performs between two consecutive times where it is in state k. Note that Δ_k is independent of i, as the ant must be repeating its movement behavior. It holds that $\mathcal{P}(t_{k_i+c}) = \mathcal{P}(t_{k_i}) + c \cdot \Delta_k$, for $c \geq 0$ and $i \geq 1$. Since the ant enters state k infinitely often, the ant will eventually move infinitely far in direction Δ_k. As the time difference between two adjacent t_{k_i} and $t_{k_{i+1}}$ is bounded by a constant, the ant can only discover fields that lie inside a band of constant width around the line of points $\mathcal{P}(t_{k_i})$. Since the direction of this band (Δ_k) does not change, the ant will eventually discover at most all fields inside this band, missing infinitely many fields outside of this band, and thus the ant does not discover the entire grid. Figure 3.1 illustrates this case.

As in both cases the ant fails to discover the entire grid, the lemma follows. ∎

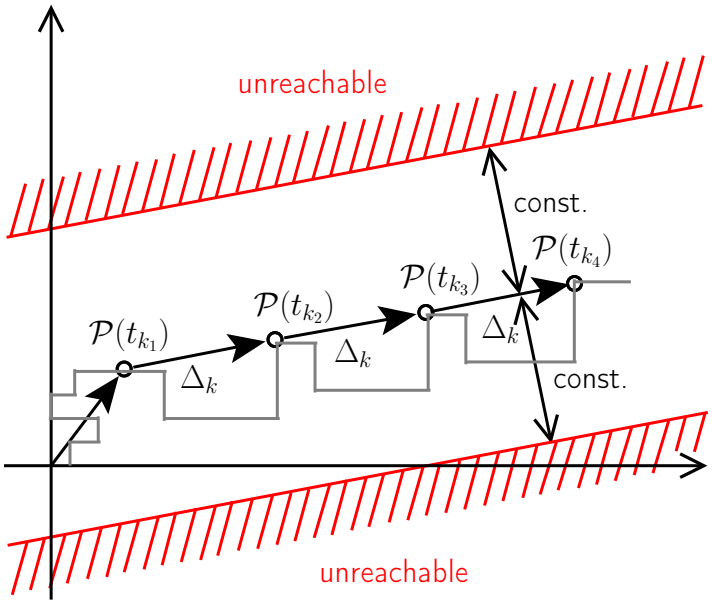


Figure 3.1: Movement pattern of a single ant; the gray line are the actual moves, the black arrows indicate direction Δ_k. All fields outside of the band around the points $\mathcal{P}(t_{k_i})$ cannot be discovered by the ant.

3.2 Two Ants

Lemma 3.3 *Two ants cannot discover the entire grid deterministically.*

Proof In order to prove this lemma we want to distinguish two different cases:

1. **The two ants meet a finite number of times.** As the two ants meet only a finite number of times, there exists a time t_s (t_s being a constant) after which they will never meet again. As they can discover at most $2 \cdot t_s$ fields up to time t_s, they have only discovered a constant amount of fields at that time. Afterwards both ants operate solely, meaning that both of them will either discover a constant amount of fields, or a band, as shown by Lemma 3.2. It is straightforward to see that two bands of finite width cannot cover the entire grid, hence we can conclude that the two ants will fail to discover the entire grid.

2. **The two ants meet an infinite number of times.** As the proof for this case is slightly more involved, we defer it and prove Lemma 3.4 first.

Lemma 3.4 *Given two ants that meet an infinite number of times, the time between two meetings is bounded by a constant.*

Proof Observe that instead of looking at the two ants moving independently, we can fix the second ant at the hive, and apply its movement steps inverted to the first ant. For example if in a given timestep the first ant moves north, and the second ant moves south, we can let the second at at its position and say that the first ant moves two steps north. Note that the relative distance between the two ants are equivalent in both scenarios.

In the transformed way of observing the two ants, the lemma can be rephrased as: *When an ant revisits a certain field an infinite number of times, the time between two meetings is bounded by a constant.* Observe that the moving ant now is able to perform movement steps of distance two in one round, but this does not have a non-constant effect on the behavior of the ants.

Assume that in time $t = 0$ the ant stands on the specific field (i.e., the first ant is currently on the same field as the second ant). As we are looking at two finite state machines, it is no longer sufficient to look at a single state s, but we rather must look at the *combined state* (s, s') that describes in which state both of the ants are at a certain time. As both ants do only have a constant number of states, we know that the total number of combines states is constant as well (and it can be upper bounded by $q_1 \cdot q_2$, where q_1,q_2 are the number of states of the respective ants). Let (k, k') be the first combined state that is entered a second time, and let $t_{(k,k')_i}$ denote the time when the ants are for the i-th time

in the combined state (k, k'). As in the previous section, we denote by $\mathcal{P}(t)$ the position of the ant at time t, and $\Delta_{(k,k')} = \mathcal{P}(t_{(k,k')_{i+1}}) - \mathcal{P}(t_{(k,k')_i})$.

Let us distinguish four different cases:

1. The ant arrives on the field before or at $t_{(k,k')_2}$

As we know that there are at most $q_1 \cdot q_2$ many combined states, it follows that $t_{(k,k')_2} < q_1 \cdot q_2 + 1$, and thus $t_{(k,k')_2}$ is constant. Hence the ant arrives on the field again after a constant amount of time.

2. $\Delta_{(k,k')} = 0$

A consequence of case 1 is, that we know that the ant has not moved onto the specific field until time $t_{(k,k')_2}$. As we know that the position for all combined states (k, k') is equal, the ant will perform the same cycle that it did between $t_{(k,k')_1}$ and $t_{(k,k')_2}$ over and over again, never going back to the specific field. As the ant never visits the specific field again, the condition that the ant visits the specific field an infinite number of times is violated.

3. $\Delta_{(k,k')} > 0$, the ant arrives before or at $t_{(k,k')_1} + q_1 q_2 \left(t_{(k,k')_1} + 2q_1 q_2\right)$

It is obviously sufficient to show that the stated time is constant in order to prove that the ant arrives after constant time on the specific field. Since $t_{(k,k')_1} < q_1 q_2$ (as there are at most $q_1 q_2 - 1$ other states to enter) and q_1,q_2 are constants, we showed that all terms in the stated time constraint are constant, thus the ant arrives after a constant time.

4. $\Delta_{(k,k')} > 0$, the ant arrives later than $t_{(k,k')_1} + q_1 q_2 \left(t_{(k,k')_1} + 2q_1 q_2\right)$

Recall that the cycle duration (the duration between two consecutive $t_{(k,k')_i}$) is at most $q_1 q_2$.

Observe that at time $t_{(k,k')_1}$ the ant is at most $t_{(k,k')_1}$ fields away from the specific field. When the ant continues to perform $t_{(k,k')_1}$ many cycles (consuming a total time of at most $q_1 q_2 \cdot t_{(k,k')_1}$) in which of them it always moves $\Delta_{(k,k')}$, we know that the ant is in a position where the direction $\Delta_{(k,k')}$ is pointing **away** from the specific field.

Having completed these cycles, the ant performs again $2q_1 q_2$ many cycles; therefore the ant is, after an additional time of at most $2q_1^2 q_2^2$, at least distance $2q_1 q_2 \Delta_{(k,k')}$ away from the specific field. As within a single cycle the ant can only achieve to move a distance of at most $2q_1 q_2$ away from both $\mathcal{P}(t_{(k,k')_i})$ and $\mathcal{P}(t_{(k,k')_{i+1}})$ – or it would not complete the cycle properly – the ant is now too far away to reach the specific field in the next cycle. As the distance between the following $\mathcal{P}(t_{(k,k')_i})$ is only increasing (as $\Delta_{(k,k')}$ is

pointing away from the specific field), the ant can never reach the specific field again in all the following cycles. Thus the ant does not reach the specific field again at all.

The discussed four cases cover all possible arrival times, and we showed that in cases 1 and 3 the ant arrives after a constant time, but in cases 2 and 4 the ant does not arrive again at all. Hence it follows that if the ant arrives at the specific field again, it must do so after a constant amount of time. Since the transformed problem and the original statement are equivalent problems, the lemma follows. ∎

We are now ready to complete the proof of Lemma 3.3. Let (m, m') be the first combined state in which the two ants meet for the second time. Note that they can meet in between those two meetings being in different combined states; but as the time between two meetings is constant (Lemma 3.4) and the number of meetings in different combined states is at most $q_1 q_2$ (else (m, m') would not be the first combined state in which they meet for the second time), we can conclude that the time difference between two consecutive meetings in (m, m') is bounded by a constant.

Let $\Delta_{(m,m')}$ denote the positional difference between two consecutive meetings in the combined state (m, m'). As the ants will repeat their exact behavior after meeting in this state, **both** ants will continuously walk into direction $\Delta_{(m,m')}$. As the time difference between those meetings is bounded by a constant, both ants will only deviate a constant distance from the meeting points. Hence they will never discover any point that lies outside of a band of constant width around the meeting points in state (m, m'), and thus they fail to discover the entire grid.

Essentially their movement behavior will resemble the movement of a single ant, even though they will continuously meet during their joint-execution of the discovering of the band.

As this concludes the case where the two ants meet an infinite number of times, Lemma 3.3 follows.

3.3 Three Ants

Lemma 3.5 *Three ants can discover the entire grid deterministically.*

Proof We prove the lemma by providing an algorithm that accomplishes to discover he entire grid. The three ants have different roles: Let us call one the Explorer, one the HorizontalGuide and one VerticalGuide. In the beginning both the Explorer and the HorizontalGuide step one field east, and the VerticalGuide moves one field north.

Starting from this configuration the three ants start to execute their individual algorithms:

Explorer The Explorer walks diagonally from one guide to the other one. Always when it hits a guide, it alters its movement direction by 90° counter clockwise. At the end of a complete circle (i.e., when turning for the fourth time), it also moves one step outwards. Observe that the Explorer movements resemble a spiral.

HorizontalGuide The HorizontalGuide moves only on the West-East axis. At the beginning it starts with the Explorer by walking into western direction. Note that it only moves with speed $1/2$, i.e., after each step into a direction it waits for one additional step. Once it meets the Explorer, it turns around and walks back into eastern direction (as well with speed $1/2$). When they meet on the eastern side, the HorizontalGuide steps one step outwards together with the Explorer and then repeats itself, i.e., it starts to walk into western direction with speed $1/2$.

VerticalGuide At the beginning, the VerticalGuide waits until the Explorer arrives for the first time. When it arrives, the VerticalGuide starts to walk south with speed $1/2$. When it meets the Explorer again, it turns around and walks north with speed $1/2$. When it meets the Explorer again on the northern side, it repeats its behavior, i.e., it starts to walk south again. Note that the VerticalGuide does not explicitly walk one field outwards, but only implicitly as the Explorer arrives later (as the Explorer is performing a larger circle).

The execution of the algorithm can be envisioned as follows: The Explorer continuously walks circles around the hive, increasing the radius of the circle by one each time it finishes a circle. The two guides move like two pendulums, one from North to South and the other one from West to East. Observe that moving one half of the circle takes exactly twice the amount of time as walking the straight line between those points. Thus when the guides walk with speed

1/2, they will arrive exactly at the same time as the Explorer. See Figure 3.2 for an illustration of the algorithm.

As the Explorer performs all circles around the hive, it is follows that it will eventually discover every field of the grid; and since the distance that the Explorer moves is always twice the distance that the guides move, it is guaranteed that they never fail to meet and the algorithm works correctly. Hence the lemma follows. ■

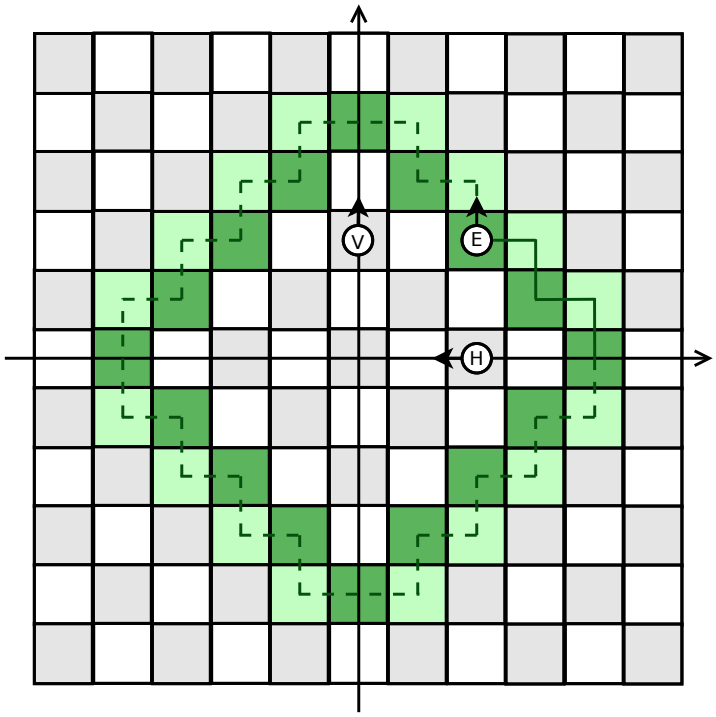


Figure 3.2: Three ants discovering the entire grid. The next step of the Explorer (E) is north, and it is currently discovering the darkgreen circle.

3.4 Conclusion

Theorem 3.1 follows from lemmas 3.3 and 3.5.

Chapter 4

Algorithms for the Adversarial ANTS Problem

In this chapter we analyze the Adversarial ANTS problem (AANTS) as it is specified in Chapter 2. Throughout this chapter, we use the model exactly as it is described in Section 2.3; i.e., we study the AANTS problem with probabilistic finite state machines.

We first establish a lower bound on the expected runtime. Then, we proceed to present multiple algorithms that accomplish to discover the treasure. For all presented algorithms, we analyze both correctness and runtime; additionally, we compare the runtime with the lower bound, investigating both the tightness of the lower bound and the quality of the algorithms. At the end of the chapter, we put the presented algorithms into relation with nature. In particular, we discuss whether one can expect to observe the presented algorithms in nature, and we interpret the implications of our findings on the understanding of ant behavior.

4.1 Lower Bound on the Runtime

In this section, we present a lower bound on the expected runtime. We first present a lower bound that is based on previous work, and we then proceed to improve this lower bound.

4.1.1 Derived Lower Bound from the Fault-Free Model

A tight lower bound on the runtime complexity for the of the original ANTS problem has already been established by Feinerman et al. in [2]. Let us look at how this lower bound of $\Omega\left(D + D^2/n\right)$ was established. Since the time it requires to walk directly to the treasure is D, it is self-evident that the lower bound must be in $\Omega\left(D\right)$. For the latter term the following reasoning can be used; assume there exists an algorithm that finds that treasure in expected runtime

$E[T] < \frac{D^2}{4n}$. Observe that at most n new fields can be discovered in each timestep, thus at time $2 \cdot E[T]$ the algorithm has discovered at most $2 \cdot E[T] \cdot n < D^2/2$ fields. As the probability of having discovered the treasure is equal to the fraction of fields in distance D that have already been discovered, and since there are more than D^2 many fields in distance D, it follows that $P\Big[T < 2 \cdot E[T]\Big] < \frac{D^2/2}{D^2} = 1/2$. This leads to a contradiction with Theorem B.2, thus we can conclude that the expected runtime cannot be smaller than $\frac{D^2}{4n}$. Combining these two bounds we get a lower bound for the expected runtime of $\Omega\left(D + D^2/n\right)$.

In the original model (using turing machines with no communication after the first round), it has been shown that the lower bound is tight by providing an algorithm with the same runtime complexity. Recall that the same lower bound is also tight for the model considering communicating finite state machines (that has been introduced by Emek et al. in [1]).

Based on this lower bound, it is straightforward to derive a lower bound for the AANTS problem with finite state machines, as it is described in 2.3: We know that the adversary can fail at most f ants, hence a possible strategy for an adversary would be to fail all the f ants in the beginning, i.e., at $t = 0$. The proof for the lower bound can thus be repeated by replacing n by $n - f$, yielding a lower bound of $\Omega\left(D + \frac{D^2}{n-f}\right)$ for the AANTS problem.

4.1.2 A Better Lower Bound

The main goal of this section is to establish a lower bound that is better (i.e., tighter) than the one that could directly be derived from the original model. Before the derive a better lower bound, we first want to establish the following lemma.

Lemma 4.1 *A lower bound of $\frac{D^2 \cdot (f+1)}{n}$ is at least as tight as a lower bound of $\frac{D^2}{n-f}$.*

Proof In order to prove the lemma, we need to show that for all possible $D \geq 1$, $n \geq 1$ and $0 \leq f < n$, the following statement is fulfilled

$$\frac{D^2 \cdot (f+1)}{n} \geq \frac{D^2}{n-f}.$$

It can be seen directly that the two sides are equal for the case $f = 0$, hence leaving only the case $f > 0$ to be shown.

$$
\begin{aligned}
n &\geq f+1 && \mid \cdot f \\
n \cdot f &\geq f^2 + f && \mid +n - f^2 + f \\
n \cdot f + n - f^2 - f &\geq n \\
(n-f)\,(f+1) &\geq n && \mid \cdot n^{-1} \cdot (n-f)^{-1} \\
\frac{f+1}{n} &\geq \frac{1}{n-f} && \mid \cdot D^2 \\
\frac{D^2 \cdot (f+1)}{n} &\geq \frac{D^2}{n-f}
\end{aligned}
$$

■

Theorem 4.2 (lower bound) *For any algorithm $\mathcal{A}$ that discovers the treasure in expected runtime $E[T]$, it holds that*

$$E[T] \geq \frac{D^2 \cdot (f+1)}{4n}.$$

Proof Let us look at a very simple adversary, namely one that keeps waiting and only fails ants once they would discover the treasure in the following step. It is clear that as soon as the number of ants walking onto the field with the treasure is $f+1$, the treasure is discovered. From this observation follow two bounds on the number of ants that need to walk onto each field: the first one is a lower bound, i.e., if fewer than $f+1$ ants walk onto a certain field, the adversary can put the treasure onto this field and the algorithm will never manage to find the treasure. Thus every algorithm that succeeds to find the treasure eventually, will need to discover each field with at least $f+1$ ants. The second bound is an upper bound: It does not provide any benefit to send more than $f+1$ ants onto a given field; since the algorithm is guaranteed to discover the treasure already after the $f+1$ ant discovered the field, every additional step onto the same field might be a wasted step.

To reach a contradiction, assume that the expected runtime is $E[T] < \frac{D^2 \cdot (f+1)}{4n}$. From Theorem B.2 we know that

$$P\Big[T < \frac{D^2 \cdot (f+1)}{2n}\Big] \geq 1/2\,. \tag{4.1}$$

As Theorem B.1 states, the number of fields up to distance $D \geq 1$ is at least D^2+1. In order to discover the treasure in the presence of the simple adversary previously described, each of those D^2 field must be covered by at least $f+1$ ants. If we look at the steps that a single ant can perform in each round, we can call a step onto a field that has not yet been covered $f+1$ times a *useful step*. Let $U(t)$ denote the number of useful steps in round t. Observe that $\forall t \geq 0 : U(t) \leq n$ and $\forall t : \sum_{i=1}^{t} U(i) \leq t \cdot n$.

Let $F(t)$ denote the number of fields that have been discovered at time t. As it takes $f+1$ useful steps to discover a field, we know that

$$F(t) \leq \frac{\sum_{i=1}^{t} U(i)}{f+1} \leq \frac{t \cdot n}{f+1}.$$

Since we know that there are at least D^2+1 fields on which the treasure could be located, it is now possible to derive an upper bound for the success probability after t rounds

$$P[T < t] \leq \frac{F(t)}{D^2+1} < \frac{F(t)}{D^2} \leq \frac{t \cdot n}{D^2 \cdot (f+1)}.$$

By setting $t = 2 \cdot E[T]$, the upper bound yields

$$P\Big[T < \frac{D^2 \cdot (f+1)}{2n}\Big] < \frac{1}{2}. \tag{4.2}$$

Since the Equations 4.1 and 4.2 contradict each other, the theorem follows. ∎

Corollary 4.3 *Any algorithm locating the treasure has an expected runtime of* $\Omega(D + \frac{D^2 \cdot (f+1)}{n})$.

Corollary 4.4 *If* $f = c \cdot n$, *for a constant* $c > 0$, *the treasure can only be located in* $\Omega(D^2)$ *time.*

Corollary 4.4 is quite surprising. Recall that already a constant number of ants can achieve an expected runtime of $\mathcal{O}(D^2)$. And if $f = c \cdot n$, the number of ants that remain can still be in $c' \cdot n$, i.e., the ant colony has still a lot more than constant many ants. Nevertheless the runtime is asymptotically equal to the runtime of a constant number of non failing ants.

4.2 General Concepts

In this section we first present a formal definition of robustness. With the help of these definitions, we rephrase our goal to develop algorithms that tolerate up to a constant fraction of ants being failed by the adversary. Then, we proceed to describe several concepts that are useful when designing algorithms. Note that all three concepts base on Theorem B.3; but since they apply the theorem to achieve different goals, three different concepts are derived.

4.2.1 Formalizing Robustness

In order to reason about robustness, we want to introduce the notation of $g(n)$-robustness. Note that the following definitions do not only apply to algorithms for the AANTS problem, but to multi-agent algorithms in general.

Definition 4.5 ($g(n)$-robustness) *Let $\mathcal{A}$ denote an algorithm. We say that $\mathcal{A}$ is* $g(n)$-robust*, if and only if $\mathcal{A}$ is successful for any execution in which the adversary fails $f \leq g(n)$ agents.* ◇

Observe that an algorithm in general can be at most $(n-1)$-robust. Regarding the AANTS problem, we know that algorithms can be at most $(n-3)$-robust, since it requires at least three ants to discover the treasure (see Theorem 3.1). Even though this definition allows to precisely capture the robustness of an algorithm, it is useful to extend this definition to also capture asymptotic robustness.

Definition 4.6 (asymptotic $g(n)$-robustness) *We call an algorithm $\mathcal{A}$* asymptotically $g(n)$-robust*, if and only if there exists a constant $c_{\mathcal{A}} > 0$ such that $\mathcal{A}$ is successful for any execution in which the adversary fails $f \leq c_{\mathcal{A}} \cdot g(n)$ agents.* ◇

Recall that our goal is to develop algorithms that discover the treasure, even when up to a constant fraction of the ants are failed (see Section 2.3). Formalizing this goal with the help of the introduced definitions, it follows that we want to develop algorithms that are asymptotically n-robust. Note that asymptotic n-robustness is the strongest (asymptotic) form of robustness.

Throughout this thesis, we will present multiple algorithms that are asymptotically n-robust. However, we will omit to exactly determine the constant factor $c_{\mathcal{A}}$ for each of them. Observe that from the construction of the algorithms, it is clear how the respective constant factors could be derived; but since we do not think that it is of any particular interest to exactly determine the factors, we decided to leave out a detailed derivation.

4.2.2 Multiple Algorithms in Parallel

When developing different algorithms, it might be the case that one algorithm achieves an optimal runtime for a certain choice of parameters f, n, D, while another algorithm achieves an optimal runtime for a different choice of parameters. A useful mechanism to create an algorithm that is optimal for the union of the parameter choices is to run all algorithms in parallel.

Theorem 4.7 *A constant number of algorithms can be executed in parallel without affecting the asymptotic bounds on f, w.h.p. The achieved runtime asymptotically corresponds to the runtime of the fastest algorithm.*

Proof Theorem B.3 states that we can split $c \cdot n$ many ants into two different clusters, where each of the clusters contains at least $\frac{c}{3} \cdot n$ many ants w.h.p. Note that this process can be repeated for a constant number of times. Therefore, we can combine a constant number of algorithms by initially letting every ant choose in which of the algorithms it wants to participate; afterwards the different algorithms can be executed in parallel. Note that the treasure is discovered as soon as one of the algorithms discovers it. Note that the different algorithms run with at least $c' \cdot n$ ants w.h.p., where the constant c' can be derived from the number of algorithms that are executed in parallel. Thus, the achieved runtime corresponds asymptotically to the runtime of the fastest algorithm, concluding the theorem. ■

For a more detailed derivation of the constant c', refer to the following section on Super Ants, where we apply Theorem B.3 in a similar way. Note that the constant c' influences the constant factor $c_{\mathcal{A}}$ for every algorithm that makes use of the idea to execute multiple algorithms in parallel, but it does not affect the asymptotic robustness of the algorithm.

4.2.3 Super Ants

In this section we introduce the concept of *super ants*, which play a key part in the proposed algorithms. A super ant can either be regarded as a special kind of ant, that is guaranteed to work properly, i.e., cannot be failed by the adversary, or as a big cluster of ants that fulfill a certain task together. From an algorithmic perspective, a super ant can easily be simulated by clustering many (i.e., more than f) ants together, and let them execute the same (non-probabilistic) protocol. Since more than f ants work on the same task, the task is guaranteed to be successfully completed, i.e., the super ant as a whole cannot be failed.

Regarding ants in nature, it might be more reasonable to actually think of a tougher kind of ant, as it seems rather unnatural having a big cluster of ants, all moving and behaving exactly the same. Nevertheless, we will show that in our model the two views are actually interchangeable, and thus we can use the concept of super ants in the proposed algorithms, without making any changes to the computational model.

The first important aspect to note is that for any algorithm there can only be a constant number of super ants. Let us now look at how super ants are created in an algorithm. Assume that there are c (where c is a constant) many super ants[1]. At the beginning of the execution of an algorithm, before anything

[1]Note that we build one additional cluster of ants – the cluster in which all the ants are, that do not represent a super ant. Observe that this additional cluster has been omitted for the sake of clarity in this section; it can easily be looked at as an own super ant, that does not

else is performed, each ant tosses $\lceil \log(c) \rceil$ many coins, and based on its outcome it will become part of the cluster representing the respective super ant. Based on Theorem B.3, we can split kn many ants into a constant number of clusters, each containing at least $c' \cdot kn$ many ants w.h.p.; thus it follows that c super ants can be constructed this way.

Nevertheless, this procedure immediately yields a bound on the number of failures that we can tolerate, since we require that at least one ant survives in each of the super ant clusters. There are c many super ant clusters, and in each one are at least $n \cdot \left(\frac{1}{3}\right)^c$ ants w.h.p. Hence the maximum number of failures we can tolerate, so that for no super ant cluster is wiped out completely, is at most $f \leq n \cdot \left(\frac{1}{3}\right)^c - 1$. Note that $f \geq c_{\mathcal{A}} \cdot n$ still holds for a constant $0 < c_{\mathcal{A}} < 1$, but the number of super ants introduces an upper bound on $c_{\mathcal{A}}$. Note that even though this upper bound affects that constant factor of the asymptotic robustness, algorithms that use super ants are still asymptotically n-robust.

Since the super ant clusters must behave "as one", i.e., each of the ants in the cluster must behave identically as all the other ants[2], it follows that super ants cannot directly make use of probabilistic functionality, without having an impact on the runtime; a single ant can use probabilistic functionality, and use it for its decisions within a single time unit. In contrast, a super ant would need to solve, e.g., consensus, after using probabilistic functionality, in order to prevent the cluster from splitting up. Since solving consensus in our model (featuring only a very limited form of communication) can take a long time, we decided to disallow super ants from using probabilistic functionality.

Using the described procedure we can make use of super ants in any algorithm without changing the model. Let us quickly summarize the properties of super ants and the implications of using an algorithm with super ants.

- The use of super ants restricts the number of failures, i.e. $c_{\mathcal{A}}$ (see Definition 4.6), but the algorithm is still asymptotically n-robust.
- There can only be a constant number of super ants.
- Super ants cannot be failed.
- Super ants cannot use probabilistic functionality.

need to follow the restrictions of the super ant (i.e., ants in this last cluster do not need to stay together etc.).

[2] If certain ants of the cluster behave differently than other ants, the cluster would essentially split into two different clusters, leading to a decrease of the cluster size. As the size of the cluster is the critical property that prevents the cluster from failing, such a split violates the assumption that they cannot be failed and is thus not feasible.

4.2.4 Quarterplane Instead of Grid

Theorem 4.8 *Any algorithm that discovers the treasure located in one specific quarter plane in expected time $\mathcal{O}(g(D, n, f))$ can be adapted to find treasures in the entire grid in expected time $\mathcal{O}(g(D, c_n \cdot n, c_f \cdot f))$ w.h.p., for constants c_n and c_f.*

Proof Assume that we have an algorithm $\mathcal{A}$ that discovers the entire quarter plane Q, i.e., it finds treasures that are located in this quarter plane. We can now transform $\mathcal{A}$ into an algorithm $\mathcal{A}'$ that discovers the treasure for any quarter plane in the following way. First every ant randomly decides for a quarter plane Q'. Then, it changes its internal sense of orientation in such a way, so that Q' becomes what Q has been before. Now the ant starts to execute $\mathcal{A}$, but it only communicates with other ants that have decided to search Q' as well.

Since there are only four quarter planes, we know that each of these partitions that execute one instance of $\mathcal{A}$ contain at least $c_n \cdot n$ many ants w.h.p, see Theorem B.3. As $\mathcal{A}$ may impose restrictions on the number of failures that it can tolerate, we need to adapt the number of failures that the $\mathcal{A}'$ can tolerate proportionally; additionally, the runtime of $\mathcal{A}$ may depend on f, thus the runtime of $\mathcal{A}'$ will depend on $c_f \cdot f$. This is due to now having only a fraction of n performing $\mathcal{A}'$, but the adversary could fail all f ants within one specific $\mathcal{A}'$, forcing a particular quarter plane to deal with a larger ratio of failures to total number of ants compared to the original algorithm.

Note that all these adaptations only require changes that are in the order of constants, thus the theorem follows. ∎

Drawing Method

When we draw an illustration of an algorithm that works solely on a quarter plane, it might be handy to draw the two rays parallel next to each other instead of them being aligned in a right angle. Figure 4.1 illustrates this drawing transformation. Note that the distance between the rays increases the farther away from the hive – i.e., the farther downwards in the illustration – even though this is not directly visible from the transformed illustration.

4.2.5 Combining the Concepts

Note that all three presented concepts impose upper bounds on the constant $c_{\mathcal{A}}$, which determines the fraction of ants that the adversary can fail. The upper bound of failures that an algorithm can tolerate, that uses two or three of those concepts can determined by multiplying the individual upper bounds. Since the

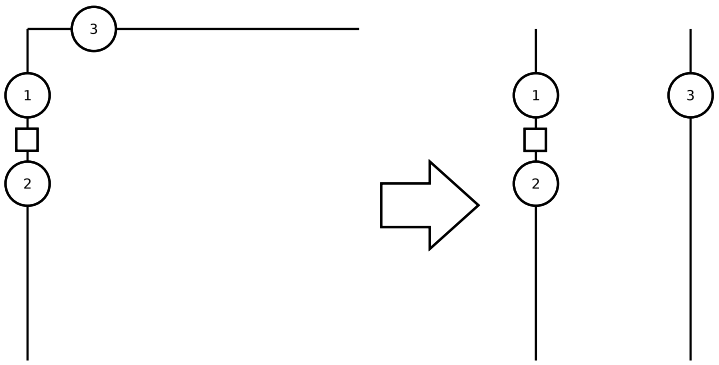


Figure 4.1: Drawing method for algorithms that work on a quarter plane. On the left: Actual positioning of ants on the quarter plane. On the right: Transformed illustration.

resulting upper bound is still a constant fraction in respect to n, algorithms that use the presented concepts are still asymptotically n-robust.

4.3 Three-Ant Search

We have shown in Section 3.3 that three ants can discover the entire grid. It is clear that we can use this algorithm – with super ants instead of single ants – to get an algorithm that will reliably discover the treasure. Since correctness has already been shown in Lemma 3.5 and the robustness is a consequence of that we only have three super ants, and no other ant participating in the algorithm, it only remains to analyze the runtime.

Theorem 4.9 *Three-ant search is an asymptotically n-robust algorithm that discovers the treasure in $\mathcal{O}(D^2)$ time.*

Proof To derive an upper bound for the runtime, we only count fields as discovered, once the Explorer super ant steps onto them[3]. Since the movement of the Explorer consists only of circles of a given distance around the hive, the treasure is discovered at the latest when the Explorer completes the circle of distance D. Observe that in order to perform a circle of radius r, an ant does not only have to step onto all the fields in distance r, but also on all the fields in distance $r-1$ (due to the diagonal movement on the grid). Thus between any two fields in distance r that are newly discovered, there is always at most one additional field that must be stepped onto.

[3]Note that in the actual execution of the algorithm, one of the guides may step onto the treasure first (if the treasure is located on one of the two axes). It is clear that such an event would only yield a lower runtime, thus looking only at the Explorer will yield an upper bound.

As shown in Theorem B.1, the total number of fields up to distance D is $F(D) = 2D^2 + 2D + 1 \in \Theta(D^2)$. Since we know that it takes at most two steps to discover a new field, the total runtime to discover the entire area up to distance D is in $\Theta(D^2)$ and the theorem follows. ∎

4.4 FastSpread with Failures

Emek et al. introduce in [1] a procedure called FastSpread. FastSpread is a means to distribute a set of ants onto fields in a line, such that eventually there is exactly one ant on each field. Recall that the first $s/6$ fields are ready (i.e., contain exactly one ant) after $s + c \cdot \log(n)$ time w.h.p, where c is a constant. The purpose of FastSpread is to allow a later procedure to make use of the sequential distribution of the ants; for example, the algorithm of Emek et al. [1] makes use of a second procedure to build teams. Let us now look at how failures affect FastSpread by establishing the following theorem.

Theorem 4.10 *Failures will never slow down FastSpread; i.e., it holds that for every surviving ant, the time until it is ready cannot be increased by another ant that fails.*

Proof Let us look at what happens when one ant fails. In Figure 4.2 we illustrated a flow chart that illustrates the different effects a failure can have, based on the state in which the failing ant was in. Observe that even though there are multiple states in which an ant can fail, they only lead to three different effects on the runtime of the surviving ant. Let us look at the three possible effects in more detail.

The first possible effect is that a gap is created by an ant that fails, which remains there (i.e., is not fixed by another ant arriving on this field) until FastSpread terminates. This might happen if an ant that is ready fails, or if an ant fails exactly at the moment when other ants move forward and the ant that fails was the only one that stayed on the field. Observe that since no other ant that is not ready will ever move on the field where the ant failed, the failure does not affect the runtime of a not ready ant[4].

The second possible effect is that an ant that is alone on its field fails, i.e., one that is ready or about to become ready. Note that difference to the first effect is only that at a later point in time ants will arrive on the field of the failed ant. Let us distinguish three groups of ants that could be affected by the failure, omitting all ready ants, as they can anyway not be affected anymore. The first group is the one consisting of ants that are not ready and will eventually move over the field of the failed ant, but at the time when they move over the field, a

[4]Note that we do not try to deal with the resulting gap, but we leave this task to the procedure that follows FastSpread.

different ant is ready on this field again. Note that those ants do not even realize that a failure has occurred, and behave oblivious to the failure. Nevertheless there is one ant fewer, thus ants of this group will eventually compete with one ant fewer, leading to a decrease in their runtime[5].

The second group of ants are those that move onto the field and there is no ready ant on the field again yet. If the ant did not fail, all those ants would have needed to step over this field, before they could participate in any decision process again. Since they can now start this process again already on the failed field, they do not need to do this one step, thus their runtime is reduced. Also observe that they potentially must compete with fewer ants (as they do not need to walk to the first field where there are already other ants competing), which potentially will lead to an additional decrease in their runtime.

The third group of ants is the one that has already passed the field where the ant failed. The only effect on these ants is, that there are arriving ants that walked past the field of the failing ant. As previously observed, not only will one fewer ant arrive, but also will the other ants arrive at a later stage of the algorithm (as they are busy with the decision process on the field of the failed ant). Both these consequences will potentially decrease the runtime of ants from the third group, but never increase it.

As the second possible effect also does not lead to an increase of the runtime for any ant, it only remains to look at the third possible effect: the ant dies during the decision process, but no field is empty after the failure. It is directly clear that the decision process on the field where the failed ant would have been is decreased, while the other fields are not affected directly. As the indirect consequences have already been discussed, the runtime is only decreased for the ants of this group as well.

Since the runtime might only be decreased for all the possible effects that a failing ant can trigger, the theorem follows. ∎

Corollary 4.11 *For every positive integer $s \leq n$ and for every constant c, after $6s + c\log(n)$ time, there is exactly 0 or 1 ant on each field up to distance s w.h.p.*

4.4.1 Linking FastSpread with Other Procedures

Let us now look at how we can keep track of the progress made by FastSpread. In the original procedure, the innermost ant was used in order for the ready ants to start participating in later stages of the overall algorithm. Since the innermost

[5]Note that the smaller the number of not ready ant on a field is, the quicker they will become ready.

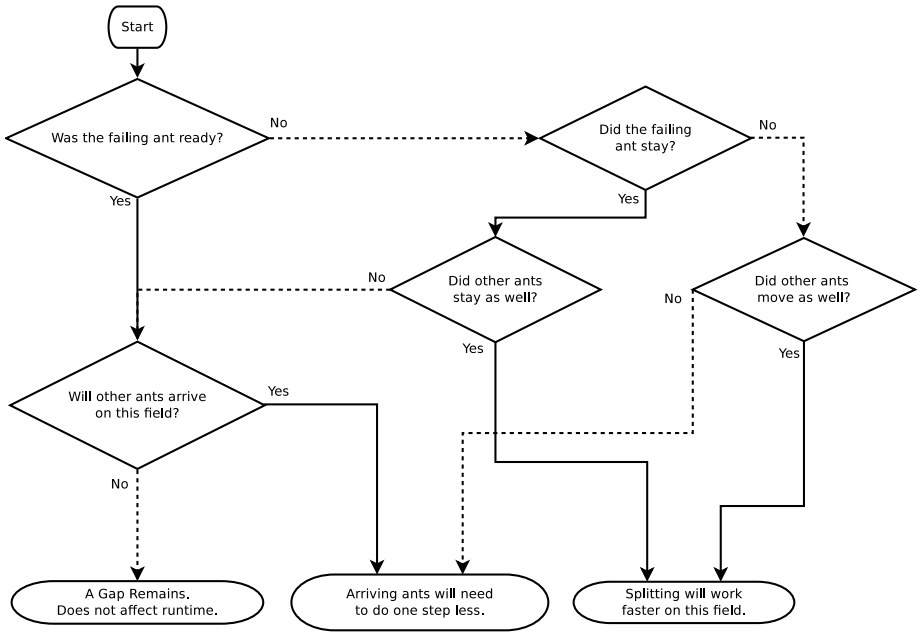


Figure 4.2: The different cases in which an ant may fail during FastSpread and the respective consequences.

ant can now fail, it is evident that we need to come up with a different means to tell the ready ants what they need to do afterwards. We can achieve this by adding two super ants to FastSpread. One super ant follows the "head" of the ants that spread out, i.e., it always moves outwards if at least one ant decides to move outwards. The second super ant initially stays on the first field. Afterwards it repeats the following procedure: As soon as the ant, that stands on the same field as the super ant, is ready (or there is no ant anymore, due to failures), the super ant tells it what to do and moves one step outwards. If there is no ant on the field, it will continue to move until it meets at least one ant. Upon meeting the next ant, it will wait again until this ant becomes ready. The second super ant is a replacement for the innermost ant in the original algorithm, and the first super ant can prevent the second super ant from walking outwards indefinitely. Note that once FastSpread is completed, the two super ants may participate in a different part of the overall algorithm.

We call this extended procedure Reliable FastSpread.

Observation 4.12 *Using two super ants, Reliable FastSpread can initialize the first s fields in $6s + c \cdot \log(n)$ time w.h.p. (c is a constant), independently of the failures forced by an adversary*[6]*.*

[6]Note that the number of failures that an adversary can force is bounded by the constraints for having super ants.

Observation 4.13 *Reliable FastSpread terminates (i.e., all the ants are either initialized or have been failed by the adversary) in at most*[7] $6n + c \cdot \log(n)$ *time w.h.p., independent of* f*.*

4.5 Segment Sweep

In this section we introduce the procedure Segment Sweep. Since this procedure is an essential part for the later algorithms, we discuss it here in depth. The purpose of Segment Sweep is to discover an area of a certain size, e.g., to discover all fields (in one quarter plane) with distances $[x, x + l]$ from the hive. Note that the procedure only discovers a single quarter plane, but as stated by Theorem 4.8, the procedure can be extended to discover the entire grid. In this section we only discuss the highlevel idea of Segment Sweep, for additional details refer to Section C.1.

4.5.1 Procedure Description

Segment Sweep assumes to start in a certain configuration, namely: On the left ray[8] there are two groups of super ants, and in between them are n worker ants lined up. Note that each worker ant stands alone on its field. On the right ray there is another group of super ants, having the same distance to the hive as the group of super ants on the left ray that is closer to the hive. The top half (without the dashed part) of Figure 4.3 illustrates this configuration; squares represent worker ants, circles represent super ants.

Starting from this configuration, the super ants start a coordinated "segment-attempt", in which all the worker ants (and some of the super ants) walk over the quarter plane. Every worker ant performs a quarter-circle and stops on the right ray, with the same distance to the hive that it had at the beginning of the segment-attempt (where it stood on the left ray). Note that all starting and stopping of worker ants is performed by meeting super ants, thus no worker ant will ever walk past the right ray.

After all worker ants arrived on the right ray, there exists a super ant that knows whether or not every worker ant arrived successfully. If all worker ants arrived, we call it a *successful segment-attempt*, if at least one worker ant failed to arrive (i.e., died) we call it a *failed segment-attempt*. In both cases the worker ants and some super ants turn around and walk back to the left ray in the same fashion as before. Once they arrive, they will perform one of two procedures, based

[7] Since the ants that are part of a super ant cluster do not participate in the initialization, there are less ants to be initialized, which speeds up the process.

[8] Since we are only looking at quarter planes, we call the parts of the two axes that bound the quarter plane *left (right) ray*.

on whether it was a successful segment-attempt or not. If it was a successful segment-attempt, the super ants and the worker ants all move outwards, in such a way that they will get into the same configuration as at the beginning, but with every ant being farther outwards. They walk outwards exactly by such a distance, that the following segment starts one field farther outwards than the current one ended. The Segment Sweep procedure can now be called again, to discover fields in this new segment.

However, if it was a failed segment-attempt, we know that at least one worker ant failed. To deal with this problem, the worker ants are condensed towards the inner super ant, with the goal to create a segment of worker ants that starts at the same position as before, having again one worker ant on every field of the segment.

Let us quickly explain the condensing procedure. The condensing procedure runs for a certain amount of time, in which all worker ants repeat the following simple procedure, that consists of four steps: At time t, all ants with an odd distance to the hive walk to the adjacent field closer to the innermost super ant. Checking at time $t+1$, if there is already another worker ant on that field; if so, they move back to the field where they came from, but if there is no ant, they remain. At time $t+2$ and $t+3$ the same steps are performed by worker ants that stand on an even distance to the hive. Repeating to perform such steps long enough will ensure that the ants form a continuous segment again (if the adversary does not fail any ants during or at the end of the condensing).

Obviously this segment will be shorter; therefore, the outer super ant on the left ray moves inwards too. Now we have the same configuration as in the beginning, with the same start of the segment of the worker ants, but with fewer worker ants. Now the Segment Sweep procedure can be called again, in order to retry a (smaller) segment and discover the fields in this area.

A very detailed explanation of the procedure can be found in Section C.1.

4.5.2 Runtime of a Successful Segment-attempt

Figure 4.3 illustrates the distances over which the worker ants and the super ants must move in order to perform a Segment Sweep. d is the smallest distance between the two rays, and x is equal to the number of the worker ants plus the two super ant groups. Note that the distance between the two rays increases by two for each field farther outwards that an ant stands.

The runtime of a successful segment-attempt is only depending on the parameters d and x. Let us look at the runtime of the different steps of a successful segment-attempt individually, in order to derive the runtime for the entire segment-attempt. We distinguish four different steps, as described in C.1. The runtime of the different steps is calculated in Table 4.1, yielding a total runtime

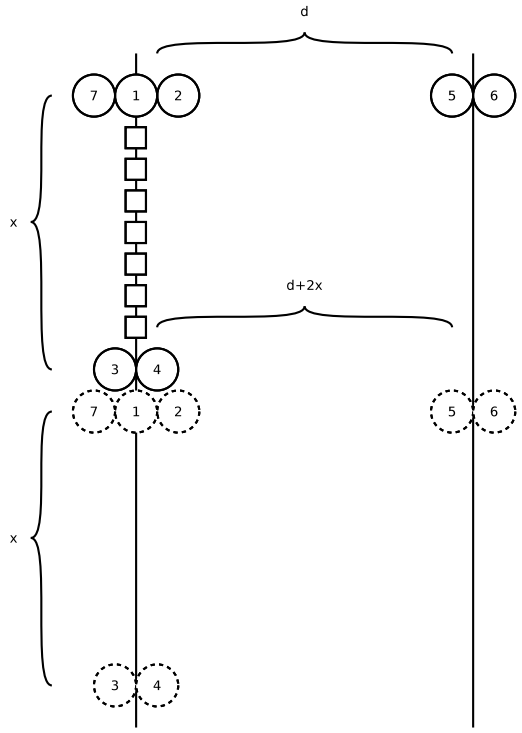


Figure 4.3: The distances that are involved in movements of a single Segment Sweep.

of $2d + 9x \in \Theta(d + x)$ for a successful segment-attempt.

Segment-attempt	$x + (d + 2x) = d + 3x$
Super ant repositioning	x
Walking back	$x + (d + 2x) = d + 3x$
Moving outwards	$x + x = 2x$
Total runtime	$2d + 9x$

Table 4.1: Runtime of a successful segment-attempt where x is the size of the segment and d is the shortest distance between the two rays.

4.5.3 Runtime of a Failed Segment-attempt

In order to analyze the runtime of a failed segment-attempt, we again refer to the distances x and d, as illustrated in Figure 4.3. The runtime of the different steps involved in the failed segment-attempt (see Section C.1) is calculated in Table 4.2, yielding a total runtime of $2d+13x \in \Theta(d+x)$ for a failed segment-attempt.

Segment-attempt	$x + (d + 2x) = d + 3x$
Super ant repositioning	x
Walking back	$x + (d + 2x) = d + 3x$
Odd-even repair	$x + 4x + x = 6x$
Total runtime	$2d + 13x$

Table 4.2: Runtime of a failed segment-attempt where x is the size of the segment and d is the shortest distance between the two rays.

4.5.4 Discovery Rate of Segment Sweep

Definition 4.14 (discovery rate) *The discovery rate $R_{\mathcal{P}}$ of a procedure $\mathcal{P}$ is defined as the average number of new fields that are discovered by $\mathcal{P}$ in each timestep between the start and the end of the procedure, i.e.*

$$R_{\mathcal{P}} = \frac{\#\textit{new fields discovered by } \mathcal{P}}{\textit{runtime of } \mathcal{P}}.$$

Observation 4.15 *Let n be the total number of ants; for any procedure $\mathcal{P}$ it holds*

$$R_{\mathcal{P}} \leq n.$$

Observation 4.16 *Any procedure $\mathcal{P}$ with discovery rate $R_{\mathcal{P}} \geq c \cdot n$, for a constant $c > 0$, discovers its area in asymptotically optimal runtime.* ◇

Let us first look at the number of fields that are discovered during a failed segment-attempt. As a failed segment will later be repeated by a following Segment Sweep, we consider all the fields to be **not** discovered at the end of the failed segment-attempt. Obviously some of the fields have been discovered, but by enforcing that all fields must be discovered by a successful segment-attempt, we can facilitate the analysis; and since a later Segment Sweep will be performed on the very same fields, the algorithm guarantees that eventually all fields will be discovered.

Lemma 4.17 *During a successful segment-attempt the discovery rate $\mathcal{R}_{SSA}$ is in $\Omega(x)$, where x is the length of the segment, i.e., the number of worker ants plus two.*

Proof Note that the variables x and d are used as defined before and as illustrated in Figure 4.3. Due to the diagonal movement pattern of ants when

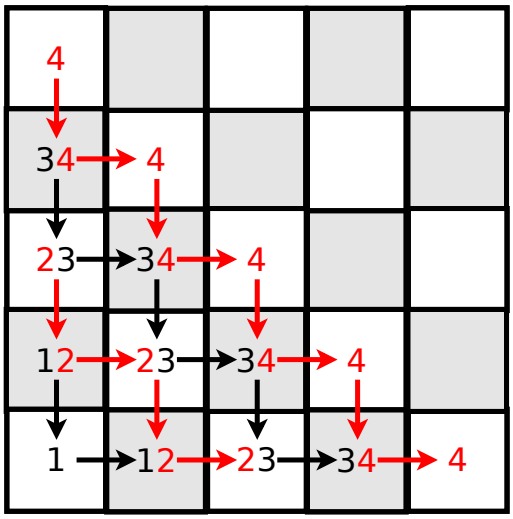


Figure 4.4: When ants move diagonally from one ray to another, at most two ants walk over the same field. The ants are numbered based on their starting distance to the hive and walk from the northern ray to the eastern ray. A certain number on a certain field means, that the ant with that number crossed it.

walking from one ray to another, every field is crossed by at most two ants. Refer to Figure 4.4 for an illustration of the movement pattern of four ants.

Observe that the innermost ant of a segment may cross fields that have already been discovered by a previous segment. Also note that if a worker ant stands in such a distance to the hive, that the distance between two rays is d, there are $d+1$ fields discovered by moving from one ray to the other one.

If we omit the innermost ant, i.e., the ant that discovers distance $d+1$, and divide the total number of fields discovered by two (since every field is discovered at most twice), we can lower bound the fields that are discovered by a successful segment-attempt as follows

$$\begin{aligned}
\#\text{new fields} &\geq \frac{1}{2}\left(\sum_{i=1}^{x}(d+2i+1)\right) \\
&= \frac{1}{2}\left(x + xd + 2\cdot\sum_{i=1}^{x} i\right) && \mid \sum_{i=1}^{x} i = \frac{x^2+x}{2} \\
&= \frac{1}{2}\left(x + xd + x^2 + x\right) \\
&= \frac{1}{2}\left(x\cdot(x+d+2)\right).
\end{aligned}$$

We can now combine the number of discovered fields with the runtime of a successful segment-attempt to calculate the discovery rate.

$$\begin{aligned}
R_{\mathcal{SSA}} &\geq \frac{\frac{1}{2}\left(x\cdot(x+d+2)\right)}{9x+2d} \\
&\geq \frac{1}{2}\cdot\frac{x\cdot(x+d)}{9x+2d} \\
&\geq \frac{1}{2}\cdot\frac{x\cdot(x+d)}{9x+9d} \\
&= \frac{x}{18}
\end{aligned}$$

Since $R_{\mathcal{SSA}} \geq \frac{x}{18} \in \Omega(x)$, the lemma follows. ∎

Observation 4.18 *Let $\mathcal{S}$ denote a successful segment-attempt with $c \cdot n$ many worker ants (for any constant $c > 0$). It holds: $R_{\mathcal{S}} \geq c' \cdot n$. Thus, it discovers the segment in asymptotically optimal runtime (see Observation 4.16).* ◇

4.5.5 Conclusion

The presented procedure Segment Sweep allows us to discover a certain segment of a quarter plane within asymptotically optimal runtime, if the adversary does not fail any ants. Failures that occur are detected and dealt with. Observe that at the end of procedure, the ants are in a configuration that allows a next execution of Segment Sweep, making it easy to repeatedly execute this procedure.

4.6 Basic Segment Sweep

In this section we introduce the algorithm Basic Segment Sweep, which uses the procedure Segment Sweep to iteratively search the entire quarter plane for the treasure. Recall Theorem 4.8, that allows us to only look at a quarter plane instead of the entire grid. The goal of this section is to establish the following theorem.

Theorem 4.19 *Basic Segment Sweep is asymptotically n-robust algorithm (tolerating $f = c_f \cdot n$ many failures, for a constant $c_f > 0$) which discovers the treasure in $\mathcal{O}\left(n + \frac{D^2+n^2}{n-f} + f \cdot (D+n)\right)$ time w.h.p.*

4.6.1 Algorithm Description

With the procedures Reliable FastSpread and Segment Sweep already being established, Basic Segment Sweep can be described quite easily. In the beginning, several super ants are elected. The remaining worker ants perform Reliable FastSpread, in order to distribute themselves on a continuous segment on one ray. As soon as all the worker ants are distributed (which is determined by a super ant), the ants start to perform Segment Sweep. After each execution of Segment Sweep, they simply perform Segment Sweep again, until they discover the treasure.

4.6.2 Runtime Analysis

In order to analyze the runtime, we first need to determine some key variables. Note that at the beginning the number of worker ants is reduced to $c \cdot n$ w.h.p. as we split the ants into a constant number of groups. During the execution of the algorithm the number of worker ants can shrink at most by f, thus at any time during the execution the number of worker ants is in $\Omega(n - f)$; and so is the length of the segment, as the length of the segment is equal to the number of worker ants.

Lemma 4.20 *The distance that must be discovered by successful segment-attempts is at most in $\mathcal{O}(D + n)$.*

Proof We show that the distance that must be discovered by segments is at least D and at most $\max(n, 2D)$. The lower bound of D is straightforward to see, and the upper bound can be seen as follows: The number of worker ants is at most n. If the number of worker ants is larger than D, there will only one successful segment-attempt be performed. Since this segment contains at most n worker ants, the segment has length at most n, thus at most the distance up to n

is discovered. If the number of worker ants is smaller than D, multiple successful segment-attempts are necessary, of which the largest segment that does not reach D has size at most $D-1$; since the following segment can have at most the same size, the distance that will be discovered is $2D-2 < 2D$. Hence the lemma follows. ∎

As the number of successful segment-attempts that is necessary can be upper bounded by the distance that must be discovered divided by the minimum size of each segment, we can state the following observation.

Observation 4.21 *The total number of successful segment-attempts that must be performed is at most in* $\mathcal{O}\left(\frac{D+n}{n-f}\right)$.

Recall that the runtime of a successful segment-attempt with segment length of x and smallest distance between two rays d is in $\Theta(x+d)$. As we need to discover at most all fields up to distance $\mathcal{O}(D+n)$ (Lemma 4.20), the maximal segment length is in $\mathcal{O}(D+n)$ as well. Note that the maximum distance that the innermost super ant will walk is also at most in $\mathcal{O}(D+n)$, as the diagonal distance is equal to two times the distance from the hive. Therefore we know that $x \in \mathcal{O}(D+n)$ and $d \in \mathcal{O}(D+n)$. This yields the following observation.

Observation 4.22 *The runtime of every successful segment-attempt is at most in* $\mathcal{O}(D+n)$.

Combining the upper bound on the number of successful segment-attempts (Observation 4.21) with the upper bound on the runtime for each of those segments (Observation 4.22), yields the following corollary.

Corollary 4.23 *The total runtime for all successful segment-attempts is at most in* $\mathcal{O}\left(\frac{D+n}{n-f} \cdot (D+n)\right) = \mathcal{O}\left(\frac{D^2+2Dn+n^2}{n-f}\right) = \mathcal{O}\left(\frac{D^2+n^2}{n-f}\right)$. ◇

So far we we omitted the additional cost that arises from failures during the execution. Observe that failures have no impact on the runtime of either the splitting in the beginning, nor on the runtime of Reliable FastSpread. During the executions of the Segment Sweep procedures, a single failure may already force the algorithm to repeat this segment. Since we know that the runtime of the largest execution of Segment Sweep is at most $\mathcal{O}(D+n)$ (Observation 4.22), and there are at most f failures, we can state the following corollary.

Corollary 4.24 *The total additional runtime due to failing ants is at most in* $\mathcal{O}(f \cdot (D+n))$. ◇

Combining these results we can derive that the overall runtime of Basic Segment Sweep is at most $\mathcal{O}\left(n + \frac{D^2+n^2}{n-f} + f \cdot (D+n)\right)$ w.h.p.[9]. Refer to Table 4.3 for a summary of these results.

[9]Note that the runtime only holds w.h.p., as Reliable FastSpread is used to initialize the ants.

Splitting into subgroups	$\mathcal{O}(1)$
Initialization (Reliable FastSpread)	$\mathcal{O}(n)$ w.h.p.
All successful segment-attempts	$\mathcal{O}\left(\frac{D^2+n^2}{n-f}\right)$
All failed segment-attempts	$\mathcal{O}(f \cdot (D+n))$
Total runtime	$\mathcal{O}\left(n + \frac{D^2+n^2}{n-f} + f \cdot (D+n)\right)$ w.h.p.

Table 4.3: Summary of the runtime analysis of Basic Segment Sweep.

4.6.3 Failure tolerance

The only thing that remains to be shown is that Basic Segment Sweep can tolerate $f = c_f \cdot n$ many failures. Observe that both the splitting in the beginning, as well as the use of super ants, only require that $n - f = c_f \cdot n$, i.e., that $c_f \cdot n$ many ants survive. Since neither the splitting nor Reliable FastSpread cares about ants being failed, failure tolerance is fulfilled for those procedures.

Looking at the procedure Segment Sweep, we observe that it can tolerate up to all worker ants are being failed, as long as the super ants are not extinguished. Since we already assumed that super ants do not fail, Segment Sweep does not impose any new restrictions on f.

Hence the only restriction on f is the one arising from the splitting into different subgroups, i.e., Basic Segment Sweep can tolerate $f = c_f \cdot n$. Note that this implies that Basic Segment Sweep is asymptotically n-robust.

Combining the runtime analysis with the failure bound, Theorem 4.19 follows.

4.6.4 Lower Bound vs. Basic Segment Sweep

In order to analyze the performance of Basic Segment Sweep, we want to compare it with the previously established lower bound. As a first step, we want to facilitate the analysis by upper bounding the runtime of Basic Segment Sweep.

Lemma 4.25 *Given a runtime T, with $T \in \mathcal{O}\left(n + \frac{D^2+n^2}{n-f} + f \cdot (D+n)\right)$. It holds that T is also in $\mathcal{O}\left(n + \frac{D^2 \cdot (f+1)}{n} + f \cdot (D+n)\right)$.* ◇

Proof Note that if we can show that

$$\frac{D^2+n^2}{n-f} \leq \frac{D^2 \cdot (f+1)}{n} + nf + n \tag{4.3}$$

holds, the lemma will follow as a direct consequence. Hence, we proceed to show inequality 4.3. Recall Lemma 4.1 from which directly follows that

$$\frac{D^2+n^2}{n-f} = \frac{D^2}{n-f} + \frac{n^2}{n-f} \leq \frac{D^2 \cdot (f+1)}{n} + \frac{n^2}{n-f}.$$

Therefore it only remains to show that $\frac{n^2}{n-f} \leq nf + n$. Observe that if $f = 0$ the two sides are equal. Hence we show the inequality for $f \neq 0$, where we assume that at least 1 ant survives.

$$\begin{aligned}
f + 1 &\leq n && \mid \cdot nf \\
nf^2 + nf &\leq n^2 f && \mid -nf^2 - f^2 \\
0 &\leq n^2 f - nf^2 - nf && \mid +n^2 \\
n^2 &\leq n^2 + n^2 f - nf^2 - nf \\
n^2 &\leq (nf + n)(n - f) && \mid \cdot (n-f)^{-1} \\
\frac{n^2}{n-f} &\leq nf + n
\end{aligned}$$

■

Having simplified the runtime of Basic Segment Sweep, we now compare it with the lower bound. To allow for an easier comparison, we distinguish different parameter ratios between n and D. Table 4.4 shows the asymptotic bounds.

Bound on n	Basic Segment Sweep	Lower Bound
$n \in o(D)$	$\mathcal{O}\left(\frac{D^2 \cdot (f+1)}{n}\right)$	$\Omega\left(\frac{D^2 \cdot (f+1)}{n}\right)$
$n \in \Theta(D)$	$\mathcal{O}\left(D \cdot (f+1)\right)$	$\Omega\left(D \cdot (f+1)\right)$
$n \in \omega(D)$	$\mathcal{O}\left(n \cdot (f+1)\right)$	$\Omega\left(D + \frac{D^2 f}{n}\right)$

Table 4.4: Runtime of Basic Segment Sweep compared with the lower bound.

Observe that for $n \in \mathcal{O}(D)$ the upper bound on Basic Segment Sweep meets the lower bound, hence we can conclude that at least for such values of n and D the lower bound is tight. It also follows that Basic Segment Sweep is asymptotically optimal for such values. Only in the case where the number of ants is large in comparison with the distance to the treasure, the runtime deviates from the lower bound; thinking about the algorithm, it quickly becomes clear that this problem arises from initializing all ants, and then trying to perform a huge segment (with a size linear in n), even though the treasure is very close. In the following sections we will introduce different algorithms that try to overcome this issue.

4.6.5 Conclusion

Basic Segment Sweep is an algorithm that discovers the treasure within asymptotically optimal runtime if the treasure is far away from the hive in comparison with the number of ants. For other scenarios the algorithm still successfully discovers the treasure, but not within optimal runtime.

4.7 Exponential Initialization

As one of the shortcomings of Basic Segment Sweep is the costly initialization, we introduce an algorithm called Exponential Initialization that can be used to initialize the ants, while already discovering smaller segments at the same time. As the name suggests, the main concept of the algorithm is to exponentially increase the segment size that is used to perform Segment Sweep, beginning with a constant segment size.

4.7.1 Algorithm Description

In the beginning, we elect a couple of super ants, and the remaining worker ants start to do Reliable FastSpread. The ray that is used for Reliable FastSpread is the same as the one on which the ants later start the iterations of Segment Sweep, but they perform Reliable FastSpread on the **opposite** side of the hive as compared to the side on which they perform Segment Sweep[10]. During Reliable FastSpread, a super ant is used to keep track of the outermost worker ant up to which all worker ants are already initialized[11]. Note that this super ant marks the progress of Reliable FastSpread. Additionally to the procedure Reliable FastSpread as it is described, the ants perform odd-even condensing[12] towards the hive at the same time. This means that the ants perform one step of Reliable FastSpread, and then perform 4 steps of odd-even condensing towards the hive. This assures us, that the ants participating in Reliable FastSpread are never too far away from the hive. Note that this extended version of Reliable FastSpread takes five times the time of the original version. For simplicity, we change every other part of Exponential Initialization in such a way, that the ants perform one step of the algorithm, and then wait for four rounds. In the further description and analysis of the algorithm we omit this detail, as it only complicates the description and does not have any non-constant impact; instead we will perform an analysis as if the five rounds used in Reliable FastSpread plus condensing could be done in a single round. Note that in order to prevent an ant from walking farther than

[10] Refer to Figure 4.5 for an illustration; the used ray is the vertical one, the ants below the hive perform Reliable FastSpread and the ants above Segment Sweep.

[11] Where initialized means that the ant stands alone on its field and is in the "ready" state.

[12] A description of how odd-even condensing works can be found in *Phase 4: Segment Fix Odd-even* of Section C.1.2.

to an adjacent field in one such super round, a condensing ant will not condense from an odd **and** an even field in the same super round (even when it would be possible to condense two fields).

On the opposite side of the hive is the quarter plane that the ants should discover. At the beginning, several super ants position themselves, waiting for the first worker ant that participates in Reliable FastSpread to become ready. Once this worker ant is ready, they bring the worker ant into position to perform an iteration of Segment Sweep. Note that the segment size is only 3, as only one worker ant participates in the Segment Sweep. From there on, the super ants start to perform a repeating behavior: After each successful segment-attempt, the super ants double the segment size and bring in the necessary worker ants from the other side of the ray. Note that the segments always start at the hive, i.e., the first segment covers the distance in the interval $[1, 3]$, the second segment covers $[1, 4]$, the third one $[1, 7]$ and so on. As only the number of worker ants that are used is doubled, the segment sizes become $1+2=3$, $2+2=3$, $4+2=7$ and so on. When a segment-attempt fails, the segment size is not increased, instead the ants are condensed **outwards**, and the missing worker ants are replenished with new worker ants that are waiting on the other side of the hive. Figure 4.5 illustrates a snapshot of the algorithm.

There are two key remarks that we want to state here. First, the segment size is always doubled after a successful segment-attempt, allowing the algorithm to quickly perform a segment-attempt with segment size larger than D, i.e., a segment-attempt that discovers the treasure. Second, as the segment size is only increased after **successfully** discovering a segment, we know that the largest segment that is performed before discovering the treasure is at most $2D$ large, i.e., the ants do not perform an unnecessary huge segment, as it might happen during Basic Segment Sweep. Observe that if $n < D$, it is not possible to discover the treasure, but we can solve this issue by transitioning into Basic Segment Sweep as soon as all worker ants are participating in the iterations of Segment Sweep.

Let us summarize the main parts of the algorithm:

- Initialize worker ants using Reliable FastSpread, outside of the quarter plane that is being searched, and condense them immediately towards the hive.
- Iteratively perform Segment Sweep, doubling the segment size after each successful segment-attempt.
- Once all worker ants are participating in iterations of Segment Sweep, transition into Basic Segment Sweep.

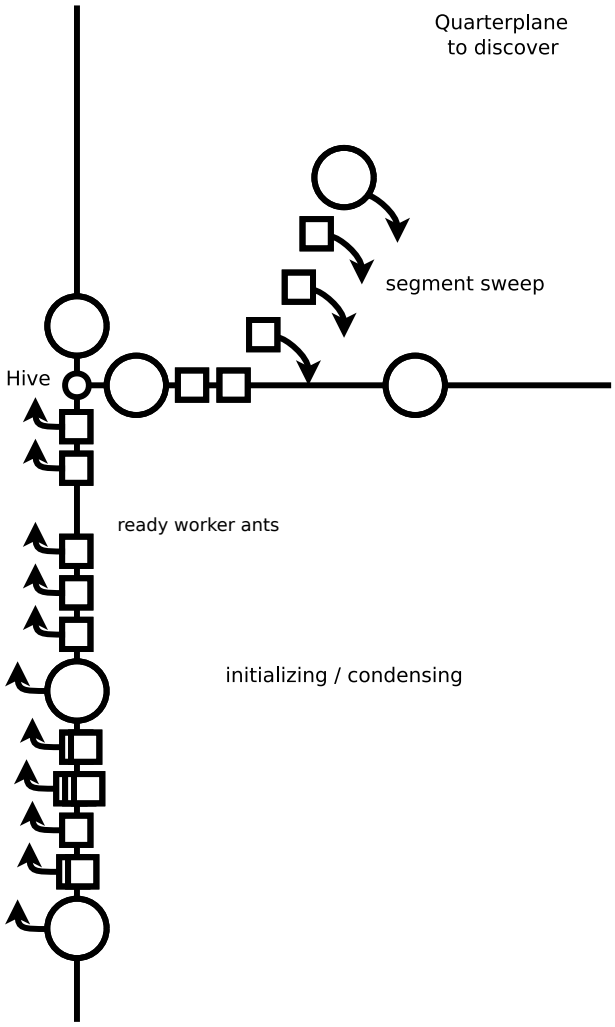
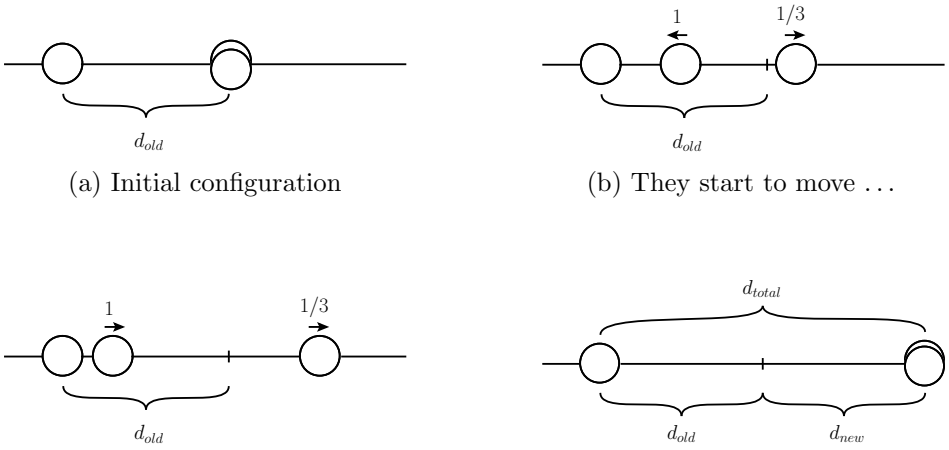


Figure 4.5: The Exponential Initialization algorithm. Circles represent super ants, squares represent worker ants. Ants above the hive perform Segment Sweep, while the ants below the hive perform Reliable FastSpread and condense towards the hive. Note that only the quarter plane above and to the right of the hive must be discovered by this group of ants.

4.7.2 Doubling a Distance

In this section we explain how a given distance between two super ants can be doubled in a exact and deterministic way. Note that we need one additional super ant for this process. Figure 4.6 illustrates the process. In the beginning, one super ant marks the left end and two super ants stand on the right end. The distance between left and right is now to be doubled. This can be achieved by letting one of the right super ants walk right with speed $1/3$ and the other one walking left with normal speed. Once the ant walking left meets the waiting super ant marking the left end, it turns around walks right. When the two super ants walking right meet again, both stop and the process is finished.

As one of the ants walks $2 \cdot d_{old} + d_{new}$ with regular speed, and the other ant walks only d_{new} with speed $1/3$, it follows that $2 \cdot d_{old} + d_{new} = 3 \cdot d_{new}$, i.e., $d_{new} = d_{old}$. As the $d_{total} = d_{new} + d_{old}$, the distance has been exactly doubled.



(a) Initial configuration

(b) They start to move ...

(c) ... the left ant catches up ...

(d) Final configuration

Figure 4.6: Three super ants doubling a distance.

4.7.3 Bringing in Worker Ants

In this section we describe how the algorithm can bring in the appropriate amount of worker ants. With the term "bringing in" we refer to the process of transferring worker ants, that are participating in Reliable FastSpread and that are ready, to the other side of the hive so that they are integrated in the subsequent iterations of Segment Sweep. Note that during this section we assume that enough worker ants are ready and waiting next to the hive, so that super ants can just come and take the ants with them. We will later prove that this assumption holds by the design of the algorithm. Note that ready ants could fail whilst they are waiting, or during the process of switching between Reliable

FastSpread and Segment Sweep; the algorithm however does not try to fix this issue directly, but this problem will simply be discovered after the next iteration of Segment Sweep, and is therefore equivalent to a worker ant failing during the segment-attempt.

There exist three different cases of bringing in worker ants: The first one is after a successful segment-attempt, when additional worker ants are needed to double the segment size. The second one is when a segment-attempt has failed, and we need to bring in worker ants to replenish the failed ants. The last case occurs, when we would need to bring in worker ants, but there are not enough worker ants anymore, i.e., the algorithm must transition into Basic Segment Sweep. To facilitate the presentation, we look at these three cases individually.

After a Successful Segment-attempt

As described in Section 4.7.2, the super ants first double the segment size of Segment Sweep, and tell all the worker ants that are already participating to move outwards. Hence the starting configuration for the bringing-in procedure is a segment, in which the half of the fields that are closer to the hive do not yet contain a worker ant, and the half that is farther outwards already contains a worker ant.

In order to bring in new worker ants, three super ants position themselves in the same way as if they would double a distance: Two super ants are on the hive and one super ant stands on the field that marks the middle of the segment. They double the segment in direction of the ants that are doing Reliable FastSpread, so that when the doubling procedure is finished, the distance between the super ants and the hive is equal to half the new segment size. When the super ants now walk towards the hive, and tell every worker ant they meet to start moving towards the quarter plane that is to be discovered, they will bring in the exact amount of worker ants that are needed.

After a Failed Segment-attempt

Instead of doubling the segment size, the super ants tell the worker ants to condense outwards, leaving a continuous gap directly at the hive. The super ants then position themselves at the hive and at the end of this gap (i.e., at the beginning of the segment of alive worker ants). From this configuration, they perform the exact same steps as after a successful segment-attempt, i.e., they double their distance in such a way, that they can bring in just the right amount of worker ants.

Not Enough Worker Ants

The super ants can easily detect that there are not enough worker ants remaining. Namely they can do so, by walking outwards on the ray where Reliable FastSpread is performed, and when they meet the super ant that marks the outermost ant of Reliable FastSpread, they know that there are not enough worker ants anymore. After this occurs, they bring in the remaining worker ants, but additionally tell all the worker ants that are performing Segment Sweep (i.e., all worker ants, since there are no more left performing Reliable FastSpread), that they should condense towards the hive. When this condensing is complete, the ants are in starting configuration for Basic Segment Sweep, as they are lined up as a single large segment, starting from the hive. From this point, they will perform Basic Segment Sweep as it is described in Section 4.6.

4.7.4 Correctness

We now want to show that the algorithm as it is described successfully locates the treasure in the respective quarter plane. Note that the correctness of Reliable FastSpread and Segment Sweep has already been shown in Sections 4.4 and 4.5. As it is straightforward to see that the procedure to double a distance works (when performed by super ants), we omit a more detailed analysis of the procedure. Note that also the correctness of the procedure of bringing in ants (under the assumption that enough ants are initialized and next to the hive) is immediately clear, as it essentially only uses the functionality of doubling a distance. Therefore there are only two parts of the algorithm for which it remains to be shown that they work correctly, both concerning the transition between Reliable FastSpread and Segment Sweep: First we need to show that **enough** ants are ready, so that they can be brought in. And second, we need to show that the ready ants are waiting directly at the hive, and are not farther outwards.

Enough Ants are Ready

Lemma 4.26 *Reliable FastSpread initializes worker ants faster than they are needed by Segment Sweep; i.e., each time super ants try to bring in worker ants, the required amount is ready (or failed) w.h.p.*

Proof From Corollary 4.11 we know that the first s worker ants that perform Reliable FastSpread are ready after $6s + c\log(n)$ time (or they are failed). As the procedure of bringing in worker ants does not distinguish between ready and failed ants, we can omit caring about failed ants in the remainder of the proof; recall that failures are only relevant during Segment Sweep.

By the design of Exponential Initialization, the super ants performing Segment Sweep initially wait until the first worker ant is ready, e.g., they wait until the

$c\log(n)$ time of Reliable FastSpread, that is required for the first worker ant to be ready, is passed. It is therefore sufficient to prove that once the iterations of Segment Sweep start, they will require fewer than $t/6$ ants at any time t.

In Section 4.5 we did a detailed runtime analysis of Segment Sweep, from which we know that the runtime of a segment-attempt is at least $9x$, where x is the size of the segment[13]. Observe that enough worker ants are ready for the first segment-attempt that is performed, as the super ants indeed wait for the first worker ant to become ready. Let us now show that enough worker ants are ready for the following segment-attempts inductively. If a segment-attempt of segment size x fails, we know that at most all (i.e., $x-2$) worker ants have failed. As the same amount of worker ants that have failed must be brought in, it follows that it is sufficient if $x-2$ worker ants have become ready during the execution of the segment-attempt. Since the segment-attempt required at least $9x$ time, we know that $9x/6 \geq x$ new worker ants have become ready w.h.p. Since already more than the required amount of worker ants has become ready just during the execution of the segment-attempt (and other ready ants may already have been there before), it follows that enough worker ants will be ready after each failed segment-attempt.

When a segment-attempt of size x is successful, the number of worker ants participating in Segment Sweep is doubled, i.e., an additional $x-2$ worker ants are required. As the time for a successful segment-attempt is also at least $9x$, it follows directly from the proof for the failed segment-attempt that enough ants are ready w.h.p.

Since we have shown that both after a failed segment-attempt and after a successful segment-attempt there are enough ants ready w.h.p., the lemma follows. ∎

Ants are Close to the Hive

Lemma 4.27 *Whenever super ants arrive to bring in new worker ants, the required worker ants form a continuous segment next to the hive (possibly containing failed ants).*

Proof As in the previous proof, we want to omit dealing with failed ants, as this does not lead to a problem with the overall algorithm. For that reason, we only require the worker ants to form a continuous segment when there have no ants been failed; in other words, in an execution where the adversary does not fail any ants participating in Reliable FastSpread, we assure that the required amount of ants will be in position.

[13]Since x includes the super ants marking start and end of the segment, a segment of size x requires $x-2$ worker ants.

Combining this restriction with Lemma 4.26 which states that the required amount of ants will be in the ready state, it sufficient look only at the condensing procedure and show that each time the super ants arrive to bring in further worker ants, the respective worker ants have condensed in such a way, that they form a continuous segment next to the hive.

Note that when looking at the fields on which Reliable FastSpread is executed, one might see different things on the different fields: For example, a single worker ant that is in the ready state, a group of worker ants that are deciding which ants is going to stay, or a super ant, which is marking the end of the segment. Let us call those different combination of ants *groups*; i.e., during the execution of Reliable FastSpread, we refer to the ants that stand together on **one** field as a *group*, independent of their number and their states. This facilitates reasoning about the **condensing** procedure, as we can separate the two concerns: The fact that the groups are close enough to the hive, i.e., that the condensing works, and the fact that the right "type" of group stands next to the hive, i.e., a ready worker ant. Observe that the second concern has already been taken of care of by Lemma 4.26, thus it only remains to be shown that the groups condense quickly enough.

Let us specify what remains to be shown a bit more formally: At every time t when the super ants want to bring in x worker ants, the first x fields next to the hive must be occupied by groups.

We first want to look at how the condensing affects the distance between adjacent groups performing Reliable FastSpread.

Observation 4.28 *The distance between two adjacent groups participating in Reliable FastSpread is at most* 2*.*

This observation follows directly from the way how the groups perform condensing: Initially all groups are adjacent, and the condensing only starts to have an effect, once a segment of groups is removed (i.e., a continuous segment next to the hive that is brought in to perform Segment Sweep). From this time on, the following behavior can be observed for all groups: If the adjacent group in direction of the hive is the direct neighbor (having distance 1), it might move at most one step[14], creating a distance of 2. In the following time steps either both groups will move synchronously (as they are both on a field with an even respectively odd distance to the hive), keeping the distance of 2, or only the group that is farther outwards will move, reducing the distance to 1 again. Note that the group that is farther outwards will move until it reduces the distance to 1 again, as only at this point it will be blocked from further condensing, whereas the group that is closer to the hive will be blocked earlier and hence will stop condensing earlier.

[14]It can only move, if there is not yet another group next to it in direction of the hive.

Observation 4.29 *Once a group starts condensing, it will only stop once it is part of the continuous segment that starts at the hive.*

We saw that the only event that triggers condensing is when super ants remove a continuous segment of groups next to the hive. It is clear that the remaining group that is closest to the hive starts condensing first, and only stops once it arrives at the hive. All other groups will start to condense later, namely one time step after the group before them started to condense. Recall that a group only stops to condense, once the group that is next to it in direction of the hive stops to condense (see explanations for Observation 4.28). Hence it follows by induction that every group will only stop to condense once it is part of the continuous segment of groups that starts at the hive.

Lemma 4.30 *Let the first x groups next to the hive be removed at time τ, and let $t(j)$ denote the time when the j-th remaining group*[15] *arrives at its position in the continuous segment starting at the hive. It holds $t(j) \leq \tau + 2j + x$.*

Proof Observe that with the j-th remaining group we denote the group for which only $j-1$ other groups are closer to the hive. Let us first look at how long it will take at most until the j-th group starts condensing: Recall that a group starts condensing one timestep after the adjacent group that is closer to the hive starts condensing. In the worst case, all $j-1$ groups form a continuous segment; in that case, the j-th group will start condensing after j timesteps, i.e., at time $\tau + j$.

Once the j-th group is condensing, it follows from Observation 4.29 that the time until the group is at its position in the continuous segment is equal to the distance that it needs to traverse. Observe that the distance to the hive at the time when the j-th group starts to condense is at most $x + 2j$; the term x is due to the x removed groups, and the term $2j$ comes from the fact that there are $j-1$ groups closer to the hive and the distance between each adjacent group is at most 2 (Observation 4.28). The distance of the j-th group to the hive is, once it is part of the continuous segment, of course j. Hence the distance between starting and end point is at most $x + 2j - j = x + j$.

Since the time it takes for the j-th group to reach its position in the continuous segment is the sum of the time it needs to wait plus the traverse time, Lemma 4.30 follows. ∎

Corollary 4.31 *When the first x groups next to the hive are removed at time τ, the next x groups have condensed completely towards the hive no later than at time $\tau + 3x$.*

[15] I.e., the group that is at distance $x + j$ from the hive at time τ.

Let us now prove Lemma 4.27 inductively. It is clear that the groups are initially next to the hive, as Reliable FastSpread starts at the hive and no ant has yet been removed. For all the following points in time when super ants try to bring in new worker ants, we know that Segment Sweep was just performed. Let x be the number of worker ants that participated in the iteration of Segment Sweep that has just been performed, i.e., the segment size minus the two super ants marking start and end. Since x worker ants participated, it follows that in the previous step of bringing in worker ants, at most x worker ants have been removed from the side performing Reliable FastSpread. Since the time it takes for the condensing process to form a continuous segment again is grows in the number of the removed ants, we assume that x worker ants have been removed to show the lemma for the worst case scenario. Let us now look at the maximum number of worker ants that might be brought in for the next iteration of Segment Sweep; as the previous segment contained x worker ants, it follows that, independent of whether the segment-attempt was successful or not, the number of additional worker ants that must be brought in for the next iteration is at most x.

Hence we know that in the previous iteration at most x worker ants have been removed, and now the super ants require at most x new worker ants. From Corollary 4.31 follows that these x new worker ants are in position after at most $3x$ time. As the runtime analysis in Section 4.5 shows, the runtime of an iteration of Segment Sweep with segment size x is at least $9x$, hence the required worker ants will already be in position when the super ants arrive and Lemma 4.27 follows. ∎

Conclusion

We have shown that the worker ants are ready in time (Lemma 4.26) and that they are close enough to the hive (Lemma 4.27). Hence it follows that the interface between Reliable FastSpread and Segment Sweep works correctly, and the correctness of Exponential Initialization follows.

4.7.5 Runtime Analysis

In order to analyze the runtime of Exponential Initialization, we first need to recall the runtime of Segment Sweep; in Section 4.5 we have shown that the runtime of Segment Sweep is in $\mathcal{O}(x + d)$, where x is the segment size and d is the inner distance, i.e., the smallest distance that a worker ant has to discover. Note that this runtime holds for both successful and failed segment-attempts. As during Exponential Initialization all iterations of Segment Sweep start directly at the hive, we know that $d = 1$, hence the runtime of Segment Sweep will be in $\mathcal{O}(x)$.

During Exponential Initialization, the super ants bring in worker ants after

each iteration of Segment Sweep; note that the procedure of bringing in new worker ants only requires the super ants to traverse distances that are in $\Theta(x)$, hence the combined runtime of executing Segment Sweep and bringing in new worker ants is still in $\mathcal{O}(x)$.

Recall that the iterations of Segment Sweep only start, once the first worker ant has been initialized by Reliable FastSpread, i.e., after $\mathcal{O}(\log(n))$ time w.h.p.; hence this term must be considered for the total runtime of the algorithm.

In order to analyze the runtime, we want to split the total runtime into two parts and analyze them individually: First we analyze the total time consumed for the successful segment-attempts, and second we analyze the additional time that is needed to deal with ants that have been failed by the adversary. Note that in the following we assume that the treasure is reached **during** the exponential increasing of the segment size, and before the algorithm transitions into Basic Segment Sweep. We will later look at what runtime is achieved when the treasure can only be discovered with transitioning into Basic Segment Sweep.

To facilitate the proof, we will omit the fact that the super ants that mark the boundary of the segment-attempts discover fields themselves, i.e., we will only look at the worker ants when speaking of the segment size. Note that this assumption will only lead to a worse runtime compared to the runtime achieved by the actual execution of the algorithm, hence the runtime yielded by the following analysis will be an upper bound for the actual runtime.

Successful Segment-attempts

Lemma 4.32 *Let s be the smallest number of successful segment-attempts that are necessary to reach the treasure. It holds that $s \leq \lceil \log(D) \rceil + 1$.*

Proof Since all the segments are started directly at the hive, the i-th successful segment-attempt covers all the distances up to 2^{i-1}. Let us now derive the distance that is discovered by performing $s = \lceil \log(D) \rceil + 1$ segments.

$$\begin{aligned} 2^{s-1} &= 2^{(\lceil \log(D) \rceil + 1) - 1} \\ &= 2^{\lceil \log(D) \rceil} \\ &\geq 2^{\log(D)} \\ &= D \end{aligned}$$

As a distance of at least D is discovered, it follows that the treasure will be discovered, and the lemma follows. ∎

Lemma 4.33 *The total runtime for all successful segment-attempts is in $\mathcal{O}(D + \log(n))$ w.h.p.*

Proof Recall that the term $\mathcal{O}(\log(n))$ w.h.p. stems from the super ants initially waiting for the first worker ant to become ready. Let us now look at the remaining runtime. Let T be the total runtime of all successful segment-attempts, and recall that a segment-attempt of size 2^i (including bringing in the necessary worker ants) has a runtime of $\mathcal{O}(2^i)$. We have

$$\begin{aligned}
T &= \sum_{i=0}^{s-1} \mathcal{O}(2^i) \\
&= \mathcal{O}\left(\sum_{i=0}^{s-1} 2^i\right) && \mid s \leq \lceil \log(D) \rceil + 1 \\
&= \mathcal{O}\left(\sum_{i=0}^{(\lceil \log(D) \rceil + 1) - 1} 2^i\right) && \mid \lceil \log(D) \rceil \leq \log(D) + 1 \\
&= \mathcal{O}\left(\sum_{i=0}^{\log(D)+1} 2^i\right) \\
&= \mathcal{O}\left(\sum_{i=0}^{\log(D)+1} 2^i\right) \\
&= \mathcal{O}\left(2^{\log(D)+2} - 1\right) \\
&= \mathcal{O}\left(2^{\log(D)}\right) \\
&= \mathcal{O}\left(D\right).
\end{aligned}$$

Since the total runtime is the sum of the initial waiting cost and T, the lemma follows. ∎

Failed Segment-attempts

Lemma 4.34 *The total runtime for all failed segment-attempts is at most $\mathcal{O}(Df)$.*

Proof From the design of the algorithm we know that the only procedure that is influenced by failing ants is Segment Sweep. Recall that a single failure can already force an additional iteration of Segment Sweep. Since the largest segment that is performed has a size of at most $2D$, the largest runtime of a single iteration of Segment Sweep is at most $\mathcal{O}(D)$. As the adversary can force at most f additional iterations of this segment, the total runtime is in $\mathcal{O}(Df)$, hence the lemma follows. ∎

Runtime for Discovery During the Exponential Increasing

Combining Lemma 4.33 and Lemma 4.34, the following corollary follows.

Corollary 4.35 *If the treasure is discovered during the exponential increasing, the runtime is in* $\mathcal{O}(D + Df + \log(n))$.

Is the Treasure Discovered during the Exponential Increasing?

Corollary 4.35 yields a runtime for Exponential Initialization, but in order to make the corollary useful, we need to investigate the question on what conditions are necessary so that the treasure is discovered during the exponential increasing, i.e., before transitioning into Basic Segment Sweep. Thus the goal of this section is to establish the following lemma.

Lemma 4.36 *The treasure is discovered during the exponential increasing if* $D < k \cdot n$, k *being a constant, and* $f \in o(n)$.

Proof Recall that the number of worker ants is not n, but only $c \cdot n$ w.h.p., for a constant $c > 0$, as, e.g., the algorithm makes use of super ants.

Observation 4.37 *If the adversary cannot prevent the ants from successfully discovering a segment of size* σ*, it cannot prevent the ants from discovering all the previous segments.*

When we think of what it means for an adversary not to be able to prevent a segment of size σ, it is clear that this is equal to the statement $f \leq cn - \sigma$. This follows from the fact that $cn - \sigma$ is the number of worker ants that can be used to replenish any failed ants when attempting a segment of size σ. When the adversary cannot fail all the worker ants from this pool plus one additional ant participating in Segment Sweep, the ants will eventually perform a successful segment-attempt. As all segments that are previous to the segment of size σ contain fewer ants, i.e., $\sigma' < \sigma$, the inequalities $f \leq cn - \sigma'$ hold and thus the adversary cannot prevent those segments to be successfully discovered.

Observation 4.38 *The largest successful segment-attempt during exponential increasing discovers a segment with at least size* $cn/4$ *w.h.p.*

This observation can be seen with the following reasoning: We can assume that the segment that is attempted has a size of less than $cn/2$, else the ants would have previously performed a segment of size $cn/4$ and afterwards doubled the segment size, and the observation would hold. Since the segment that is attempted contains fewer than $cn/2$ worker ants, there are at least $cn/2$ worker

ants that can be used to replenish any failed worker ants. Since $f \in o(n)$, it follows that the adversary cannot fail enough ants to prevent the algorithm from successfully discovering the segment.

Combining observations 4.37 and 4.38, it follows that during the exponential increasing, Exponential Initialization discovers a segment of at least size $cn/4 \in \Theta(n)$ w.h.p., if $f \in o(n)$. Therefore Lemma 4.36 follows. ∎

Runtime when Linked with Basic Segment Sweep

Let us now look at the runtime that is achieved when the treasure is **not** discovered during the exponential increasing, i.e., when the ants need to transition into Basic Segment Sweep. Let us first investigate how the configuration of ants on the grid looks like at the end of the exponential increasing: All worker ants have been brought in from Reliable FastSpread, and are lined up to perform Segment Sweep. This is just the configuration that is required for Basic Segment Sweep to start the iterations of Segment Sweep. Thus to analyze the overall runtime until the treasure is discovered, we need to compare the time it takes to initialize the ants in Basic Segment Sweep (i.e., by simply performing Reliable FastSpread) against the time it takes to initialize the ants with Exponential Initialization.

Lemma 4.39 *The asymptotic runtime of Basic Segment Sweep is not affected when using the initialization strategy of Exponential Initialization instead of using Reliable FastSpread to initialize the worker ants.*

Proof Corollary 4.35 states that the runtime of Exponential Initialization to reach distance σ[16] is in $\mathcal{O}(\sigma+\sigma\cdot f+\log(n))$. Given a total of n ants, it is clear that the maximum distance that can be reached during the exponential increasing is n, hence the time it takes to initialize all worker ants with Exponential Initialization is at most $\mathcal{O}(n+n\cdot f+\log(n)) = \mathcal{O}(n+n\cdot f)$. Adding this initialization cost to the runtime of Basic Segment Sweep (as stated by Theorem 4.19) does not affect the achieved runtime, thus the lemma follows. ∎

4.7.6 Conclusion

Summarizing the findings of the previous sections, we have established an algorithm that works at least as well as Basic Segment Sweep, with the additional benefit of being capable of discovering the treasure already during the initialization. Table 4.5 summarizes the achieved runtime for the different parameter combinations. Note that since the runtime analysis of Exponential Initialization requires to have a more fine-grained parameter distinction, we vary both D and f in the table.

[16] We replaced D with σ to clarify that the treasure is not yet discovered.

		f		
		0	≥ 1, $o(n)$	$c \cdot n$, $0 < c < 1$
n	$o(D)$	$\mathcal{O}(\frac{D^2}{n})$	$\mathcal{O}(\frac{D^2 \cdot f}{n})$	$\mathcal{O}(D^2)$
	$\Theta(D)$	$\mathcal{O}(D)$	$\mathcal{O}(Df)$	$\mathcal{O}(D^2)$
	$\omega(D)$	$\mathcal{O}(D + \log(n))$	$\mathcal{O}(Df + \log(n))$	$\mathcal{O}(n^2)$

Table 4.5: Asymptotic Runtime of Exponential Initialization with transitioning into Basic Segment Sweep.

Comparing these results with the runtime results of Basic Segment Sweep (see Table 4.3), we see that Exponential Initialization brings us closer to an optimal solution, but there remain several parameter combinations for which the algorithm does not match the lower bound.

4.8 Uniform Splitting

The previously presented algorithms all suffer from a weakness: While a rather small amount of ants is participating in iterations of Segment Sweep, many ants are waiting. The main issue is not the waiting itself, but the fact we use only a relatively small portion of the ants to discover segment, i.e., only a single ant stands on each field of a segment, allowing the adversary to pay very little in order to fail an entire segment. It follows from this observation, that an algorithm could reduce its runtime, if it forced the adversary to pay more in order to fail a segment; and the way to achieve that is to use multiple ants discovering the same segment, i.e., to use redundancy.

In this section we present an algorithm that achieves redundancy by using probabilistic means, combined with Segment Sweep to discover the segments.

4.8.1 Algorithm Description

Before explaining how the algorithm works, we want to point out an important difference to the previous algorithms, namely that Uniform Splitting does not require any initialization, e.g., as performed by Reliable FastSpread. One benefit of this is, that it allows us to get rid of the $\mathcal{O}(\log(n))$ w.h.p. initialization term.

Let us now describe Uniform Splitting. In the first timestep, multiple super ants are elected. One super ant marks the next segment size that must be discovered, where the segment size is doubled after each successful segment-attempt. This super ant initially waits on one of the rays of the quarter plane that must be discovered, with a distance of 3 to the hive. The doubling of the segment is performed as described in Section 4.7.2.

Starting from the second timestep, the following sequence of operations is performed repeatedly: First all the worker ants are distributed according to a **uniform distribution** on all the fields of the current segment; i.e., if the current segment size is σ, each worker ant positions itself with probability $1/\sigma$ on each of the fields of the segment. Afterwards, a Segment Sweep is attempted, where the uniform distribution might lead to multiple worker ants discovering the same distance to the hive. If the segment-attempt was successful, i.e., if **at least one** worker ant arrives on each distance, the segment size is doubled; note that doubling the segment in Uniform Splitting means to only move the super ant that marks the segment size twice as far from the hive. Independent of whether the segment-attempt was successful or not, all worker ants are brought back to the hive, and the next iteration begins, i.e., the worker ants get again uniformly distributed for the next segment.

Summarizing Uniform Splitting, the key points are as follows:

- One super ant marks the next segment-size to perform.
- The segment size is doubled after each successful segment-attempt.
- The worker ants are distributed uniformly on all the fields of the current segment, providing redundancy.

Since Segment Sweep has already been analyzed, the only part that remains to be analyzed is the distribution of the worker ants.

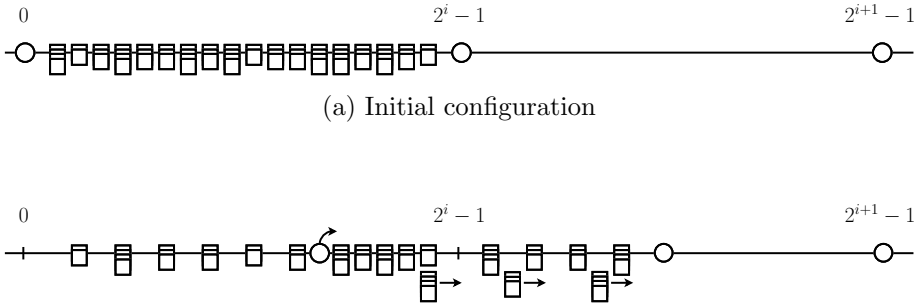
4.8.2 Distributing the Ants

In this section we explain how we can distribute worker ants uniformly on 2^i fields, for $i \in \mathbb{N}_0$; we call this procedure Distribute. Since Distribute works inductively, we will first establish the two base cases for $i = 0$ and $i = 1$. The case $i = 0$ is solved by simply guiding all worker ants onto the single field. For $i = 1$, the super ants first guide all worker ants onto the first field, then each of the worker ants tosses a coin and moves onto the field number two with probability $1/2$.

We now show the induction step: We start with the worker ants distributed uniformly over 2^i fields, and super ants are marking both the start and the end of the segment (i.e., are positioned at field 0 and $2^i - 1$). The goal is now to distribute the worker ants uniformly over 2^{i+1} fields. As a first step, we move one super ant onto the field $2^{i+1} - 1$, i.e., we double the segment size. Afterwards a super ant starts to walk from field 0 to field 2^i, and tells all worker ants that stand on **odd** fields to start moving outwards. The super ant that notifies all those worker ants makes a pause of 1 timestep on each odd field, to prevent that multiple worker ant groups end up moving on the same field. The worker ant

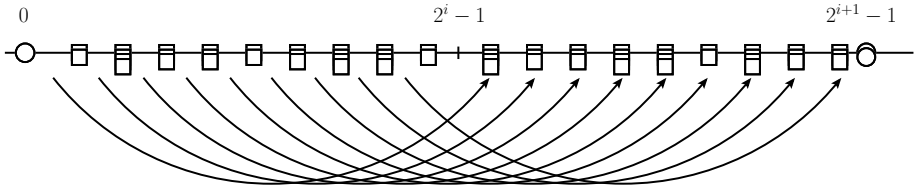
groups that are moving outwards walk until they meet another super ant, which is initially waiting on field 2^i, and then stop on this field. This super ant walks two fields outwards for each group of worker ants that arrives. Therefore the worker ant groups are stopped on the fields 2^i, $2^i + 2$, $2^i + 4$, ..., $2^{i+1} - 2$.

When all worker ants are distributed on the even fields, a super ant walks over all fields, telling each worker ant to toss a coin and move to the neighboring odd field (with a distance of one higher to the hive) with probability one half. Figure 4.7 illustrates the induction step of Distribute.

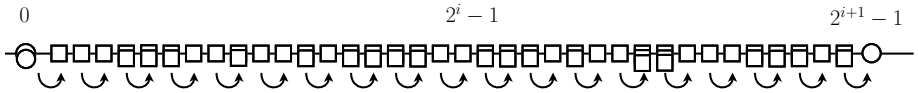


(a) Initial configuration

(b) Worker ants on odd fields move to the second half ...



(c) Overview which worker ant moved to which field



(d) Coin toss decides the final position

Figure 4.7: The induction step of Distribute.

Correctness

Lemma 4.40 *Distribute distributes the worker ants uniformly on 2^i fields, for $i \in \mathbb{N}_0$.*

Proof Since the cases $i = 0$ and $i = 1$ are straightforward to see, we will directly analyze the induction step. Instead of looking at all worker ants together, we will look at a single ant. Note that showing that a single worker ant stands on each field with probability $1/2^{i+1}$ implies the lemma, as all worker ants are independent. We know that at the beginning each worker ant stands on a field $\mathcal{F} \in [0, 2^i]$. Observe that $\mathcal{F}$ corresponds to an i-bit number. The first step of the induction step is to move an ant to the even field on the outwards half, if the ant currently is on an odd field. If we look at the bit representation of the position of the ant, we see that: If $\mathcal{F}$ is even, the position does not change, i.e., $\mathcal{F}' = \mathcal{F}$. If $\mathcal{F}$ is odd, the new position is even and the ant is moved by a distance of $2^i - 1$, i.e., $\mathcal{F}' = 2^i - 1 + \mathcal{F}$. Carefully looking at how the bits change during the transition from $\mathcal{F}$ to $\mathcal{F}'$, we see that the least significant bit is removed and prepended to $\mathcal{F}$, additionally the least significant bit is set to 0. Hence we know that an ant that stands on field $\mathcal{F}'$ still encodes a position where i bits have been chosen uniformly at random.

Distribute continues to let every ant toss a coin, and moving to the odd field with probability one half; this part of Distribute lets every ant choose the least significant bit. Therefore, the ant chooses in total $i + 1$ bits by coin tosses, and stands on the position that those $i + 1$ bits represent. Thus, the worker ant stands with equal probability on any of the 2^{i+1} fields, and the lemma follows.■

Runtime

Lemma 4.41 *Distribute distributes the worker ants on 2^i fields in $\mathcal{O}(2^i)$ time, for $i \in \mathbb{N}_0$.*

Proof The runtime for $i = 0$ and $i = 1$ is constant, thus we want to solely look at the runtime of the induction step. When thinking about Distribute, we see that the runtime for the induction step in which the ants stand initially on 2^i many fields, and they get distributed onto 2^{i+1} many fields, depends only on the super ants moving around and telling worker ants what to do. Since the super ants walk only a constant number of times between the fields 0 and $2^{i+1} - 1$ (once for doubling, once for notifying to move and once for telling each ant to toss a coin), the runtime for the induction step is at most $c \cdot 2^{i+1}$.

Let us now calculate the total runtime to distribute the ants on 2^i fields T_i, where c_{init} denotes the runtime up to $i = 1$.

$$\begin{aligned} T_i &\leq c_{init} + \sum_{k=2}^{i} c \cdot 2^k \\ &= c_{init} + c \cdot \sum_{k=2}^{i} 2^k \\ &= c_{init} + c \cdot (2^{i+1} - 1) \\ &= \mathcal{O}(2^i) \end{aligned}$$ ∎

Conclusion

Combining lemmas 4.40 and 4.41 yields the following corollary.

Corollary 4.42 *Distribute distributes any amount of worker ants according to a uniform distribution on 2^i fields, within $\mathcal{O}(2^i)$ time, for $i \in \mathbb{N}_0$.*

4.8.3 Analysis

The correctness of the overall algorithm Uniform Initialization basically follows directly from Corollary 4.42, as the algorithm only consists of calling Distribute, and afterwards performing Segment Sweep. Since we already proved that both parts work correctly, we now proceed to show that the algorithm indeed discovers the treasure and we analyze the runtime. For easier reading we split the analysis into two parts: First we show that the algorithm discovers the treasure w.h.p., and second we analyze the runtime that is required to do so. The goal of the section is to establish the following theorem.

Theorem 4.43 *Let c_d be a constant. For $D \leq c_d \frac{n}{\log(n)}$ and $f \in o(n)$, Uniform Initialization discovers the treasure in at most $\mathcal{O}(D + \frac{D^2(f+1)}{n})$ time w.h.p.*

Discovering the Treasure

Lemma 4.44 *Let c_d be a constant. For $D \leq c_d \frac{n}{\log(n)}$ and $f \in o(n)$, Uniform Initialization discovers the treasure w.h.p.*

Proof Observe that segment-attempts with different sizes are performed during Uniform Initialization, as the segment size is iteratively doubled. Nevertheless, we will always calculate probabilities as if the current segment-attempt has the segment size of the **largest** segment that is performed. Note that the probabilities

for smaller segments would only be better (in terms of redundancy), as there are fewer fields to distribute the ants onto. Thus, if we prove the lemma with this modification, it follows that the lemma also holds for the actual algorithm.

Observation 4.45 *Let σ be the size of the largest segment that is performed in order to discover the treasure. It holds that $\sigma \leq 2D$.*

Since the segment size is only doubled after a successful segment-attempt, the largest segment that does not contain D has size $D-1$. Thus the next segment would have size $2D-2$, implying the observation.

Observation 4.46 *Let c_a be a constant. At any time during the execution of Uniform Initialization, there exist at least $c_a n$ worker ants w.h.p.* ◇

Before the adversary fails any worker ants, there exist $c' \cdot n$ worker ants w.h.p., for a constant $c' > 0$, due to the use of, e.g., super ants. Since $f \in o(n)$, i.e., $f < cn$ for a selectable constant c, it follows that $c'n - f \geq c'n - cn = c_a n$; thus there remain at least $c_a n$ worker ants at any time w.h.p., where c_a is a selectable constant.

Lemma 4.47 *After executing Distribute, there are at least $\frac{c_a n}{4D}$ worker ants on any specific field w.h.p.*

Proof Let X be the random variable denoting the number of ants on a specific field. It holds

$$E[X] \geq \frac{c_a n}{\sigma} \geq \frac{c_a n}{2D} \geq \frac{c_a n \log(n)}{2c_d n} = \frac{c_a}{2c_d} \log(n).$$

Applying a Chernoff bound (Theorem A.2) with $\delta = 1/2$, it follows

$$P[X < (1-1/2)E[X]] \leq e^{-\frac{1}{2}\cdot\frac{1}{4}E[X]} \leq e^{-\frac{c_a}{16c_d}\log(n)} = \frac{1}{n^{c_a/16c_d}}.$$

Since $\frac{c_a n}{2D} \leq E[X]$, it follows that $P[X < \frac{c_a n}{4D}] \leq P[X < 1/2E[X]]$. Therefore $P[X < \frac{c_a n}{4D}] \leq \frac{1}{n^{c_a/16c_d}}$, respectively $P[X \geq \frac{c_a n}{4D}] \geq 1 - 1/n^c$, which concludes Lemma 4.47. ■

Let us call an execution of Distribute *good*, if after its execution, at least $\frac{c_a n}{4D}$ worker ants stand on each field (before the adversary has failed any of the ants); vice versa we call an execution of Distribute *bad*, if there is at least one field containing fewer than $\frac{c_a n}{4D}$ worker ants.

Lemma 4.48 *After executing* Distribute*, there are at least* $\frac{c_a n}{4D}$ *worker ants on* each *of the* σ *fields w.h.p.; i.e., the execution is good w.h.p.*

Proof Let E_i denote the event that field i contains fewer than $\frac{c_a n}{4D}$ worker ants, for $i \in [0, \sigma]$. From Lemma 4.47 we know that $P[E_i] \leq 1/n^c$ for a selectable constant c. Applying a union bound (Theorem A.3), we can derive that

$$\begin{aligned} P[\bigcup_i E_i] &\leq \sum_i P[E_i] \\ &\leq \sigma \cdot P[E_i] \\ &\leq 2c_d \frac{n}{\log(n)} \cdot P[E_i] \\ &\leq 2c_d \frac{n}{\log(n)} \cdot \frac{1}{n^c} \\ &\leq 2c_d \cdot n \cdot \frac{1}{n^c} \\ &\leq \frac{1}{n^{c'}}. \end{aligned}$$

Since $P[\text{Distribute is good}] = 1 - P[\bigcup_i E_i]$, Lemma 4.48 follows. ∎

Lemma 4.49 *Performing* $r = 1 + \log(\sigma) + \frac{4Df}{c_a n}$ *executions of* Distribute*, all* r *executions are good w.h.p.*

Proof Let us first derive an upper bound r,

$$\begin{aligned} r &= 1 + \log(\sigma) + \frac{4Df}{c_a n} \\ &\leq 1 + \log(\sigma) + \frac{4c_d n f}{\log(n) \cdot c_a n} \\ &= 1 + \log(\sigma) + \frac{4c_d f}{c_a \log(n)} \\ &\leq 1 + \log(\sigma) + \frac{4c_d}{c_a} \cdot f && \mid f \leq \frac{c_a}{4c_d} n, \text{ since } f \in o(n) \\ &\leq 1 + \log(\sigma) + n \\ &\leq 1 + \log(2D) + n \\ &= 2 + \log(D) + n \\ &\leq 2 + \log(n) + n \\ &\leq 4n. \end{aligned}$$

Let the R be the random variable denoting the number of executions of Distribute that are bad, and let D_i denote the event that the i-th execution is bad. Observe that different the executions of Distribute are independent (by design of the overall algorithm), and thus $\forall_i : P[D_i] = P[\text{Distribute is bad}]$. In the following we derive the probability that at least one of the r executions of Distribute is bad, by applying a union bound (Theorem A.3).

$$\begin{aligned}
P[R \geq 1] &= P[\bigcup_{i=1}^{r} D_i] \\
&\leq \sum_{i=1}^{r} P[D_i] \\
&= \sum_{i=1}^{r} P[\text{Distribute is bad}] \\
&= r \cdot P[\text{Distribute is bad}] \\
&\leq 4n \cdot P[\text{Distribute is bad}] && \mid \text{Lemma 4.48} \\
&\leq 4n \cdot \frac{1}{n^c} \\
&\leq \frac{1}{n^{c'}}
\end{aligned}$$

Since the probability that all executions of Distribute are good is $1 - P[R \geq 1]$, Lemma 4.49 follows. ∎

Observation 4.50 *The treasure is discovered after* $1+\log(\sigma)$ *successful segment-attempts.*

Recall that the segment size is always doubled after a successful segment-attempt. Thus when performing $1 + \log(\sigma)$ successful segment-attempts, the largest segment that is successfully performed has size σ. Since $D \leq \sigma$, the treasure is discovered.

Observation 4.51 *If the adversary does not fail more than* $\frac{4Df}{c_a n}$ *of the first* $1 + \log(\sigma) + \frac{4Df}{c_a n}$ *segment-attempts, the treasure is discovered within the first* $1 + \log(\sigma) + \frac{4Df}{c_a n}$ *segment-attempts w.h.p.*

From Observation 4.50 we know that after $1 + \log(\sigma)$ successful segment-attempts the treasure is discovered. Since all of the first $1 + \log(\sigma) + \frac{4Df}{c_a n}$ distributions are good w.h.p. (Lemma 4.49), it follows that the adversary must fail more than $\frac{4Df}{c_a n}$ segment-attempts to prevent the discovery.

Lemma 4.52 *Let ρ be the maximum number of segment-attempts (out of the first $1 + \log(\sigma) + \frac{4Df}{c_a n}$) that an adversary can fail. It holds $\rho \leq \frac{4Df}{c_a n}$ w.h.p.*

Proof Since we are looking at the first $1 + \log(\sigma) + \frac{4Df}{c_a n}$ segment-attempts, we know that all executions of Distribute are good w.h.p. (Lemma 4.49), i.e., on each field stand at least $\frac{c_a n}{4D}$ worker ants w.h.p. Recall that in order to fail a round, the adversary must fail at least all the ants on one field. Hence we can derive

$$\begin{aligned} \sum_{i=1}^{\rho} \frac{c_a n}{4D} &\leq f \\ \rho \cdot \frac{c_a n}{4D} &\leq f \\ \rho &\leq \frac{4Df}{c_a n}. \end{aligned}$$

■

Combining Observation 4.51 and Lemma 4.52 yields the following corollary.

Corollary 4.53 *Uniform Initialization discovers the treasure within the first $1 + \log(\sigma) + \frac{4Df}{c_a n}$ segment-attempts w.h.p.*

Note that this corollary implies Lemma 4.44, thus concluding its proof. ■

Runtime

The goal of this section is to upper bound the runtime of Uniform Initialization based on the analysis performed in the previous section, so that we can conclude Theorem 4.43. To facilitate the runtime analysis, we will look at the successful segment-attempts and the failed segment-attempts individually.

Lemma 4.54 *The total runtime Uniform Initialization requires for all successful segment-attempts is at most $\mathcal{O}(D)$ w.h.p.*

Proof Recall that the runtime of a segment-attempt which starts at the hive with size x is in $\mathcal{O}(x)$, as established in Section 4.5. Corollary 4.42 states that the runtime of Distribute to distribute the ants on x fields is in $\mathcal{O}(x)$; therefore the total runtime of first executing Distribute and afterwards performing Segment Sweep is in $\mathcal{O}(x)$.

Observation 4.50 gives an upper bound on the number of successful segment-attempts that must be performed w.h.p.; and since $\sigma \leq 2D$ (Observation 4.45), it

follows that the algorithm performs at most $1+\log(\sigma) \leq 1+\log(2D) = 2+\log(D)$ successful segment-attempts.

By construction of the algorithm, the segment size is doubled after each successful segment-attempt (starting with a segment size of 1). Thus we can derive the following bound for the runtime

$$\begin{aligned}\sum_{i=0}^{\log(\sigma)} \mathcal{O}(2^i) &= \mathcal{O}(2^{\log(\sigma)+1} - 1) && \mid \sigma \leq 2D \\ &= \mathcal{O}(2^{\log(D)+2} - 1) \\ &= \mathcal{O}(D).\end{aligned}$$

■

Lemma 4.55 *The total additional runtime an adversary can force Uniform Initialization to require by failing segment-attempts is at most $\mathcal{O}(\frac{D^2 \cdot f}{n})$ w.h.p.*

Proof Lemma 4.52 states that the maximum number of segment-attempts that an adversary can fail is at most $\frac{4Df}{c_a n} \in \mathcal{O}(\frac{D \cdot f}{n})$ w.h.p. It is straightforward to see that the most expensive segment-attempt to fail, in terms of additional runtime, is the largest segment that is performed, i.e., the segment of size $\sigma \in \mathcal{O}(D)$. Since the runtime required for executing Distribute and performing a segment-attempt of size σ is in $\mathcal{O}(\sigma)$, it follows that the total runtime for all failed segment-attempts is at most $\mathcal{O}(\frac{D \cdot f}{n}) \cdot \mathcal{O}(D) = \mathcal{O}(\frac{D^2 \cdot f}{n})$ w.h.p., concluding the proof. ■

Corollary 4.56 *The total runtime of Uniform Initialization is at most $\mathcal{O}(D + \frac{D^2 \cdot f}{n})$ w.h.p.*

Combining Lemma 4.44 and Corollary 4.56 establishes Theorem 4.43.

4.8.4 Conclusion

The presented algorithm Uniform Initialization discovers the treasure within asymptotically optimal runtime w.h.p., but it does not work for every choice of the parameters f and D. Note that the algorithm either discovers the treasure within optimal runtime w.h.p., or it does not discover the treasure at all; this separates the algorithm from previous ones, which work for all parameter choices, but only yield suboptimal performance for certain parameter combinations.

4.9 Combining the Algorithms

In the previous sections we have introduced four different algorithms:

- Three-ant Search
- Basic Segment Sweep
- Exponential Initialization
- Uniform Initialization

Since Exponential Initialization is a superior extension of Basic Segment Sweep, we do not need to consider Basic Segment Sweep when combining the algorithms. As stated by Theorem 4.7, we can develop a new algorithm that executes the three algorithms Three-ant Search, Exponential Initialization and Uniform Initialization in parallel, obtaining the asymptotically optimal runtime of the three. Since these three algorithms are optimal for different combinations of f and D, we recapitulate which algorithm is optimal for which parameter combinations in Table 4.6. The Table combines the different runtime analysis' of the previous sections; in particular Theorem 4.9, Table 4.5 and Theorem 4.43.

		f		
		0	≥ 1, $o(n)$	$c \cdot n$, $0 < c < 1$
D	$\Omega(n)$	EI	EI	EI, 3-ant
	$\omega(\frac{n}{\log(n)})$, $o(n)$	EI	$\star$	3-ant
	$\mathcal{O}(\frac{n}{\log(n)})$	UI	UI	3-ant

Table 4.6: Which algorithms are asymptotically optimal for which parameter combination. EI: Exponential Initialization, UI: Uniform Initialization, 3-ant: Three-ant Search.

Observe that for every parameter combination, except for the one marked with the $\star$, there is at least one algorithm that achieves an asymptotically optimal runtime.

4.9.1 The Last Combination

In this section we analyze the runtime achieved by Exponential Initialization for the parameter combination that is marked by a $\star$ in Table 4.6.

Theorem 4.57 *For $\omega\left(\frac{n}{\log(n)}\right) \leq D \leq o(n)$, and $1 \leq f \leq o(n)$, Exponential Initialization achieves a runtime that is only a factor of $o(\log(n))$ larger than the lower bound.*

Proof Recall that the runtime of Exponential Initialization is in $\mathcal{O}(Df+\log(n))$, for the given bounds on D and f (Table 4.5). Note that since $D \in \omega\left(\frac{n}{\log(n)}\right)$, it follows that $\log(n) \in \mathcal{O}(Df)$; i.e., the asymptotic runtime can be simplified to $\mathcal{O}(Df)$.

The lower bound for the given parameter combination is $\Omega\left(D + \frac{D^2 \cdot f}{n}\right)$. For simplicity we reduce the lower bound to $\Omega\left(\frac{D^2 \cdot f}{n}\right)$. Observe that this lower bound is at most as large as the actual lower bound; thus if we show the theorem for this simplified lower bound, the proof also holds for the actual lower bound.

Let us now derive the factor by which the runtime of Exponential Initialization is larger than the lower bound

$$\begin{aligned}\frac{\mathcal{O}(Df)}{\Omega\left(\frac{D^2 \cdot f}{n}\right)} &= \mathcal{O}\left(\frac{Df}{\left(\frac{D^2 \cdot f}{n}\right)}\right) \\ &= \mathcal{O}\left(\frac{n}{D}\right).\end{aligned}$$

To get the largest factor by which the runtime of Exponential Initialization is larger than the lower bound, we need to maximize this term. Since the only way of maximizing the derived factor is to minimize D, and $D \geq \omega\left(\frac{n}{\log(n)}\right)$, it follows that the largest possible factor is $\mathcal{O}\left(\frac{n}{\omega\left(\frac{n}{\log(n)}\right)}\right) = o(\log(n))$. ∎

4.9.2 Conclusion

Combining the three algorithms Three-ant Search, Exponential Initialization and Uniform Initialization yields an algorithm that discovers the treasure in asymptotically optimal runtime w.h.p. for most of the values of f, D and n; additionally the runtime is, w.h.p., at most by a factor of $o(\log(n))$ larger than the lower bound for any choice of parameters.

4.10 Relation to Biology

In this chapter we have explained how the problem of foraging can be solved in a hostile environment by ants that have only limited computation capability. The presented algorithm yields a nearly optimal result in respect to the asymptotic runtime, but remains at least one question to be asked: Does the presented algorithm resemble actual ant searching behavior?

In Chapter 6, we will survey the existing biological work on the topic, but for now we want to study the concepts introduced throughout this chapter in regard to their closeness to reality.

When we recall all the details required to get the algorithm to work, it quickly becomes clear that the algorithm does not represent any behavior as it can be observed by nature. In the following we point out some aspects that, even though being crucial to the developed algorithms, will most likely not be observed in nature.

4.10.1 Super Ants

Super ants, in the sense of large clusters of regular ants performing tasks in the exact same way – especially showing the exact same movement pattern – are unlikely to be observed. One aspect making it an implausible concept is, that it would require an exactly equal decision making process for all ants in the same super ant cluster, which contradicts the typical structure of nature that always contains a certain amount of variance. A different aspect is, that a large cluster of ants standing at the same location outside of the hive would be extremely vulnerable to an adversary that stands itself on a certain position and is capable of killing all ants on a certain field, e.g., an ant eater.

Nevertheless, the described problems might be alleviated by having different kind of ants, some being a single ant of a special kind, that is represented by a super ant in our model. But note that when thinking of the concept of a super ant, it is not sufficient to only look at the super ants, one must also look at the worker ants. Observe that in all of the presented algorithms and procedures the worker ants rely heavily on super ants, i.e., if the super ants failed, most of the worker ants (or even all of them) would remain at some position outside of the hive and never do anything useful anymore (e.g., they would not even return to the hive). This heavy dependency on super ants behaving reliably makes the ant colony extremely fragile, thus we can conclude that the concept of super ants is unlikely to appear in nature.

4.10.2 Waiting Time

In most of the described procedures, many of the ants spend a large part of their time waiting. For example worker ants wait until they are brought in to perform Segment Sweep during Exponential Initialization, or they wait during the doubling of the segment, or during Segment Sweep itself when they discover only a small distance and they have to wait for the larger distance to be discovered. All this time spent with ants waiting does not affect the asymptotic runtime, but it is self-evident that ants in nature will not wait for such a long time, simply for algorithmic reasons. A good example for unreasonable waiting behavior are certain super ants (those marking the start of the segment) that are used in Segment Sweep; their purpose is actually only to simulate a pebble, so that the segment-attempt can be repeated accurately. Using ants to simply mark a (nearly) static position seems not only to be a waste of resources, but as well to be quite fragile.

4.10.3 Synchronous Time

In all the presented algorithms we use synchronous time. Note that Segment Sweep, which is the main procedure of all presented algorithms, depends heavily on having synchronous time: The super ant, that collects the arriving ant on one ray and tells every arriving ant to stop, moves exactly with speed $1/3$, as it knows that every 3 timesteps the next worker ant will arrive. One one hand this is a simple means to know when the ants arrive, but on the other hand this introduces a heavy dependency on ants arriving exactly at the right time. Since this dependency makes the procedure fragile – every worker ant that misses the super ant is lost forever – it is implausible that such a procedure will be encountered in nature.

4.10.4 Conclusion

We have discussed some of the issues that indicate that the presented algorithms are unlikely to be encountered in nature. As the described "shortcomings" of the algorithms are necessary to discover the treasure, one can conclude that the chosen model is too restrictive, i.e., actual ants have more computational power than we assumed. This conclusion is confirmed by different biological studies, which are summarized in Chapter 6. The other implications one can derive from our work are deferred to Chapter 7.

Chapter 5

Discovering the Grid With Pushdown Automata

In this chapter we investigate how the ANTS problem can be solved when we use pushdown automata instead of finite state machines. The problem statement and the rest of the model stays the same. Note that we only allow **deterministic** behavior, i.e., no random bits, as we want to analyze the capabilities of pushdown automata in the same way as in Chapter 3. The main goal of this chapter is to establish the following theorem.

Theorem 5.1 *It takes at least two pushdown automaton ants to deterministically discover the entire grid. Additionally this bound is tight, i.e., there exists an algorithm discovering the entire grid with two pushdown automata.*

5.1 Changes to the Model

Since the only change to the model is that the replacement of finite state machines with pushdown automata, we now proceed to define the pushdown automata. A pushdown automaton is defined by the 5-tuple

$$M = (Q, \Gamma, \delta, q_0, \epsilon)$$

where Q is the finite set of states, Γ is the finite set of stack symbols, δ is the transition function, q_0 is the initial state and ϵ is the initial stack symbol. Note that the transition function is defined as $\delta : Q \times \Gamma \to 2^{Q \times \Gamma^* \times \{N,E,S,W,P\}}$, but only finite elements of Γ^* are allowed, i.e., the pushdown automaton can only push a constant number of stack symbols onto the stack within each transition.

Additionally, we limit ourselves to pushdown automata that in each timestep either only remove a symbol from the stack, or write some symbols on the stack; i.e., we do not allow a pushdown automaton to "replace" symbols on the stack in

a single timestep. Observe that any pushdown automaton that does not satisfy this condition can easily be transformed into one which does; the transformation simply consists of splitting any state in which the pushdown automaton both removes and pushes symbols from the stack into two consecutive states. It therefore follows that this limitation does not limit the power of the pushdown automaton.

5.2 One Pushdown Automaton

The goal of this section is to establish the following theorem.

Theorem 5.2 *A single pushdown automaton ant cannot discover the entire grid deterministically.*

Let us first introduce some notation: $S(t)$ denotes the number of elements on the stack at time t, and by $C(t)$ we denote the *configuration* of the pushdown automaton at time t. The configuration is defined as the 2-tuple consisting of the current state of the pushdown automaton and the topmost stack symbol, i.e., $C(t) = (q, \gamma)$ for $q \in Q$ and $\gamma \in \Gamma$.

Observation 5.3 *The total number of different configurations is $|Q| \cdot |\Gamma|$, i.e., constant.*

Lemma 5.4 *Let t_1, t_2 be two different times such that $C(t_1) = C(t_2)$, and let t_2 be the first time at which the pushdown automaton enters configuration $C(t_1)$ after t_1. If $S(t_i) \geq S(t_1)$, for all $t_1 < t_i \leq t_2$, then the pushdown automaton will repeat the transitions made between t_1 and t_2 forever.*

Proof Let γ be the topmost stack symbol at t_1; note that since $C(t_1) = C(t_2)$, γ is also the topmost stack symbol at t_2. We know that between t_1 and t_2 the stack never is never smaller than at time t_1, thus the part of the stack that is below γ is never accessed between t_1 and t_2. Therefore, all transitions that the pushdown automaton performed were either based on γ, or on stack symbols that have been pushed onto the stack since t_1. Refer to Figure 5.1 for an illustration of this scenario.

Recall that the configuration at t_2 is equal to the one at t_1. Therefore, the same sequence of transitions as between t_1 and t_2 will be performed again, since all transitions are solely based on the part of the stack that gets built up **during** the sequence of transitions between t_1 and t_2. Thus the lemma follows. ∎

Observation 5.5 *Let t_1, t_2 be two times with $C(t_1) = C(t_2) = c$, and let t_2 be the first time that c is entered after t_1. There exists a constant k, such that $t_2 - t_1 < k$.*

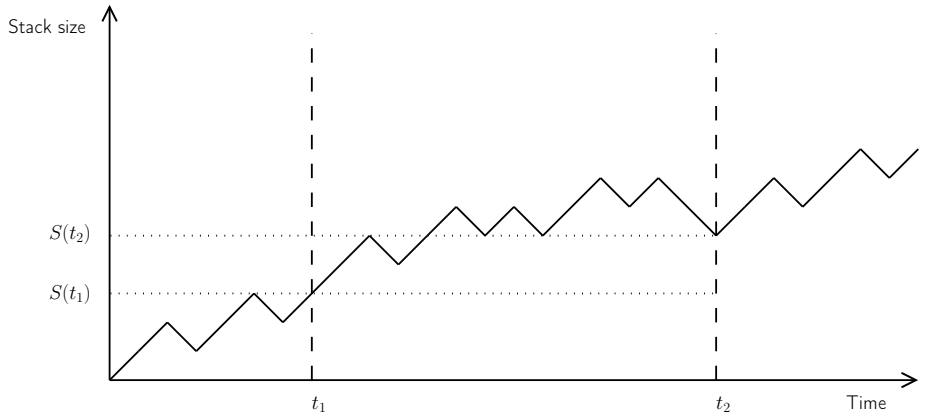


Figure 5.1: The evolution of the stack over time for a given pushdown automaton. Note that the content written onto the stack before t_1 is not accessed between t_1 and t_2.

Observe that there are two kinds of configurations: Some are entered only a finite number of times, and others are entered an infinite number of times. It is clear that there must be at least one configuration that is entered an infinite number of times, as the pushdown automaton must be able to run for an arbitrary long time.

Lemma 5.6 *Let t_∞ be the smallest time, for which holds: For every time $\tau > t_\infty$, $C(\tau)$ is entered an infinite number of times. It holds that t_∞ is finite.*

Proof Since there are only a finite amount of different configurations (Observation 5.3), there are also only a finite amount of configurations that are entered finitely often. For each of those configurations we can determine the last time when the configuration is entered. Since we can repeatedly apply Observation 5.5, it follows that for every configuration, the last time it is entered can be bounded by a constant. Therefore the maximum of those times will also be constant; we refer to this maximum as t_∞. By construction of t_∞, no configuration that is entered only a finite number of times is entered after time t_∞; thus the lemma follows. ∎

Since the pushdown automaton can only push a constant number of symbols onto the stack in each timestep, the following corollary follows.

Corollary 5.7 *$S(t_\infty)$ is finite.*

Lemma 5.8 *A single pushdown automaton will start to repeat its behavior after a finite amount of time, and the cycle length of the repeating is finite as well.*

Proof To prove the lemma, we ignore the first t_∞ timesteps, and directly start reasoning about the repeating behavior at time $t_\infty + 1$. To show the lemma, we will now show that after a finite time after t_∞, there will occur two times such that Lemma 5.4 is fulfilled.

At time $t_\infty + 1$, we observe configuration $C(t_\infty + 1)$ and we do the following: We know that every configuration will be entered again (Lemma 5.6); thus let t' denote the first time at which the pushdown automaton is in $C(t_\infty + 1)$ again. Recall that this occurs after a finite amount of time (Observation 5.5).

At t' we check if Lemma 5.4 is fulfilled. If it is fulfilled, we know that the pushdown automaton will repeat this behavior forever, and the proof is concluded. If it is not fulfilled, we know that there must be at least one time before t' at which the stack was lower than $S(t_\infty + 1)$. Let $t_{\min} \leq t'$ be the time at which the stack size was minimal. Figure 5.2 illustrates this case. We then apply the same reasoning iteratively, starting from $t_{\min}$ and $C(t_{\min})$. In each iteration either Lemma 5.4 must be fulfilled, or the stack size is decreased by at least 1.

Let us now assume that Lemma 5.4 is not fulfilled for a long sequence of iterations. Observe that each iteration only takes a finite amount of time (if the lemma is not fulfilled, $t_{\min}$ occurs before the same configuration is entered a second time, and it only takes a finite amount of time to enter the configuration a second time (Observation 5.5)). Since the stack shrinks at least by 1 in each iteration, the stack will be empty after a finite number of iterations (recall Corollary 5.7). Let t_{empty} be the first time after $t_\infty + 1$ where the stack is empty. Because every configuration is entered infinitely often, the configuration $C(t_{\text{empty}})$ will also be entered again; and since the stack is empty at t_{empty}, Lemma 5.4 will hold for t_{empty} and the next time $C(t_{\text{empty}})$ is entered.

Therefore we can conclude that Lemma 5.4 will either already be fulfilled at some time during the described iterations, or at latest when the stack is completely empty. Since t_∞ is finite, and there are only a finite number of iterations which themselves do also only take a finite amount of time, and the time until the lemma is fulfilled starting from t_{empty} is also finite, it follows that the repeating behavior starts after a finite amount of time. This concludes the lemma. ∎

Corollary 5.9 *A deterministic pushdown automaton with no input, i.e., one that works solely with its states and stack, is not more powerful than a deterministic finite state machine.*

Since we already showed that a single deterministic finite state machine cannot discover the entire grid (Lemma 3.2), Theorem 5.2 follows.

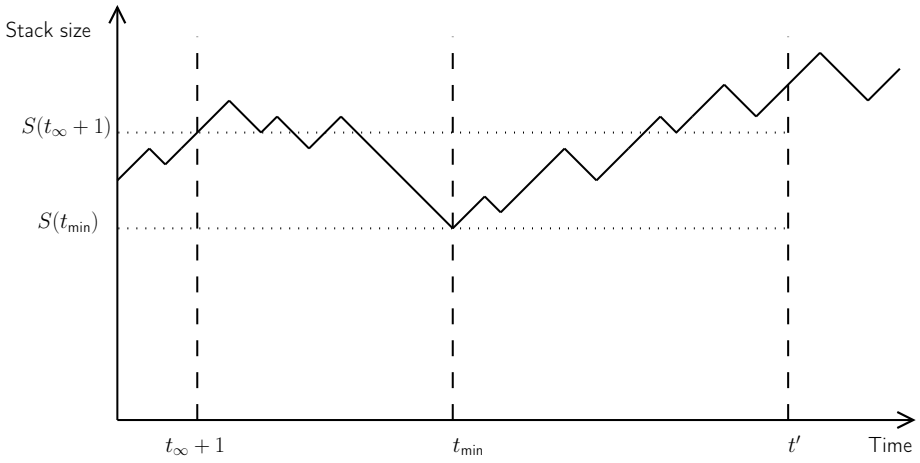


Figure 5.2: Lemma 5.4 is not fulfilled for $t_\infty + 1$ and t', because there exists a t_{min} with a smaller stack size in between. The next iteration will check if Lemma 5.4 is fulfilled starting from t_{min}.

5.3 Two Pushdown Automata

We have shown that the quarter plane (and hence the grid) cannot be discovered deterministically with a single pushdown automaton. In this section we present an algorithm that discovers the entire grid deterministically using only **two** pushdown automata. Recall that when using finite state machines we needed at least three finite state machines; hence we can conclude that pushdown automata using communication are more powerful than finite state machines in regard to the ANTS problem.

Lemma 5.10 *Two pushdown automaton ants can discover the entire grid deterministically.*

Proof We prove the lemma by providing an algorithm that discovers the grid. The algorithm is very similar to the 3-ant Search described in Section 3.3. Instead of having one Explorer and two Guides, we perform the same spiral search around the hive using one Explorer and only one Guide. The Explorer performs the exact same steps as in the algorithm 3-ant Search, i.e., it does not even use its stack. The Guide now replaces the two Guides, by using the stack in the following way: Initially it walks outwards in eastern direction together with the Explorer. Whenever the Guide walks outwards, it pushes one symbol onto the stack in each timestep. When it meets the Explorer, it turns around and walks back to the hive, removing symbols from the stack one by one. When the stack is empty, the Guide knows that it arrived at the hive; thus it turns by 90° degrees and starts

to walk outwards again. Note that the Guide walks with normal speed and not with speed 1/2.

Since both the Explorer and the Guide always walk the same number of fields towards the hive and outwards, just in different order, they always meet on the axis. Upon each meeting, both of them realize that they need to change their direction, and so they successfully manage to perform a spiral search. Hence the lemma follows. ∎

5.4 Conclusion

We have established that even though a single pushdown automaton cannot discover the entire grid deterministically, already two pushdown automata can solve this task. Since it takes at least three finite state machines to achieve the same task, it therefore follows that pushdown automata with the ability to communicate are more powerful than finite state machines in our computational model. One might think that since this model is more powerful, considering it might lead to different algorithms that are closer to what is observed in nature. We argue that this is not the case, since we have shown that in the very same way as for finite state machines, pushdown automata also rely strictly on having some sensory input in order to perform anything that is non-repeating (see Corollary 5.9). As one needs to guarantee that such input exists, one would be forced to use the concept of super ants again. And since super ants are limited to a constant number, the only algorithms that could be designed with pushdown automata would be algorithms in which the majority of the ants work together in a single huge team, in the same way as for finite state machines. We already discussed that such algorithms are unlikely to be observed in nature (see Section 4.10); thus we claim that algorithms that use pushdown automata instead of finite state machines do not get closer to nature, and we omit the design of such algorithms.

Chapter 6

The Biological Perspective

Since we discussed the computer science point of view of the ants foraging problem thoroughly in the previous chapters, we also want to shed light on the subject based on research performed by biologists. We therefore did a survey of the research performed over the last decades, with the goal of finding confirming or conflicting results in regard to the work on the ANTS problem. In this chapter we summarize the findings of the work that is most relevant to either the ANTS problem specification, the presented algorithms that solve the ANTS problem, or to ant navigation capabilities in general. We will see that the current state of research clearly suggests that ants are more powerful than finite state machines, as their navigation capabilities include remarkable features; for example, there are ants which are capable of performing the so called *path integration*, and many ants can store *landmark information* based on various cues.

6.1 Path Integration

The term *path integration* refers to the capability of an animal to keep track of its position, such that it can return to its nest in a straight line, even when the outbound path contains many turns. This could for example be achieved by memorizing the angle and distance relative to the nest. Figure 6.1 illustrates the movement pattern of an ant capable of performing path integration; note that the illustration is not based on actual data, it simply visualizes the concept of path integration.

The question of whether animals are capable of performing path integration was already raised by Darwin in 1873 [54]; note that he refers to the process as "dead reckoning", since the term path integration did not exist yet. Since then, a lot of research has been performed regarding the capabilities of animals in regard to navigation. Of particular interest for our problem is of course the research that has been performed with ants. Work that deserves attention includes especially the various studies performed by Wehner et al.; for example in [38, 39, 40, 41] one can find various information about the basic concepts of how ants use path

integration. Note that Grah et al. showed in [41] that certain species of ants are capable of reliably performing path integration in three dimensions, for example in a very hilly terrain.

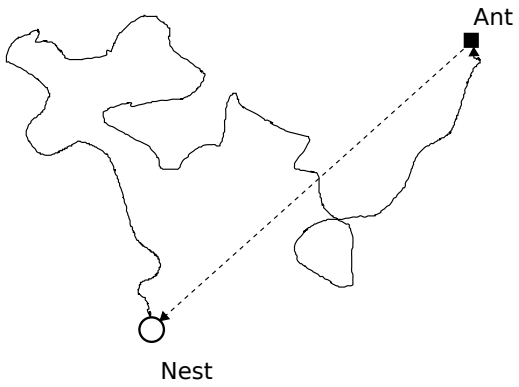


Figure 6.1: An ant left its nest for foraging (solid line). Due to the ability to perform path integration, the ant does not follow the outbound path backwards to reach the nest, but it walks in an (approximately) straight line home (dashed line).

6.2 Landmark Information

Besides path integration, many animals (including ants) make use of *landmark information*. An animal that makes use of landmark information stores various cues, with the help of which it can determine its position. A simple example of such a cue can be a large object that is visible from multiple locations. Just like path integration, landmark information is a self-contained means to enable navigation (in the absence of errors).

In the case of ants, there exist different types of cues that are memorized by ants. In [42, 43, 44, 45], it has been shown that ants use **visual** landmarks. Furthermore, [46, 47] show that ants also use **odor** information. In a more recent study, [48], it has been shown that ants can even make use of **magnetic** and **vibration** landmarks.

Summarizing the findings of these papers, it can be observed that ants are not only capable of memorizing multiple landmark cues, they can also make use of very different types of cues. Such cues are used to speed up both the foraging process, as well as the process of finding the way back to the nest. In experiments where the landmark information was altered, the ants quickly realized that a certain piece of their landmark information was outdated and started to adapt to the new situation. Such observed behavior suggests that the processing of landmark information in ants is quite evolved.

6.3 Combining Path Integration and Landmark Information

An interesting question that arises when looking at path integration and landmark orientation separately, is how they are combined. There are many studies that investigate this question. For example in the work of Collett and Graham [49], experiments with hamsters are conducted to test for the interactions between the two navigation mechanisms. Coming back to the ant setting, Wehner et al. [42, 45] investigate the effects of removing landmarks in desert ant foraging, in order to observe how well the switching between the two navigational means (landmark orientation and path integration) works. In [43], Cheng et al. experiment with rotating landmarks instead of removing them.

Summarizing these results, one can say that ants are capable of switching between the two navigation mechanisms such that they reliably find their way back to the nest. Landmark altering can have a negative effect on the speed with which the nest is found, nevertheless ants do not seem to get lost; this seems to be due to their ability to successfully switch between the two navigation mechanisms and a random exploration movement in such a way, that they eventually find their way home. Wehner and Srinivasan [37] describe and analyze the searching behavior of a single ant, trying to find its nest.

6.4 Conclusion

The referred work clearly suggests that ants use an intelligent combination of at least three different navigation means: Path integration, landmark information and randomness. Studies indicate that not only the individual mechanisms are quite evolved, but also the interaction between them. This results in a both robust and efficient movement behavior.

Chapter 7

Conclusion

We have studied a problem that lies at the heart of the emerging interdisciplinary research that consolidates biology and theoretical computer science. Based on the claim that only a model featuring an inherent element of uncertainty is suitable to accurately represent an everyday problem in nature, we extended the ANTS problem with an adversary that is allowed to fail some of the ants.

In this chapter we summarize our contributions, with a particular focus on the algorithmic perspective. Additionally, we interpret our results in regard to their implications for biology and we sketch an outlook for the arising collaboration between biology and theoretical computer science. Lastly, we present some ideas that could be studied in the future.

7.1 Summary of Contributions

In Chapter 3 we proved a tight lower bound for the number of ants that are required to solve the ANTS problem; namely, we showed that it requires at least three ants. To the best of our knowledge, this is the first work that investigates a lower bound on the number of ants. The importance of this lower bound lies not so much in the exact number, but in the fact that it requires more than one ant; in particular, the lower bound shows that the variant of the ANTS problem that has been introduced in Emek et al. [1] – which is the foundation of the problems studied in this thesis – is an inherent distributed problem. This distinguishes the model from both the model introduced by Feinerman et al. in [2] and the one introduced by Lenzen and Radeva [4]: The problems studied in those papers can be solved by a single ant; the only advantage they take of having multiple ants is to devise a trivial form of parallelism, accomplishing a linear speedup.

After having established the distributed nature of our model, we proceeded to add the element of uncertainty and studied the AANTS problem. In Section 4.1.2 we introduced a lower bound, and in Chapter 4 we presented multiple algorithms with different assets and drawbacks. Our main result is an algorithm with a runtime that is $o(\log(n))$-competitive w.h.p.

In Chapter 5 we analyzed the effect of slightly enhancing the computational power of an ant, by allowing them to use an infinite stack. We proved that a single pushdown automaton ant is not more powerful than a finite state machine ant; this result does not only hold for our model, but can be applied to all problems where a robot must solve a task in an infinitely large environment without any sensory input.

7.2 Concluding Remarks

As our model is the only one presented so far that includes both the need for collaboration and an element of uncertainty, it can be deduced that our model captures the most significant problems that ants face during foraging. We showed that even in a very restrictive setting, i.e., without super-constant memory, with only a very simple form of communication and in the absence of any cues in the environment, ants can efficiently locate food. This confirms the conclusion of Emek et al. [1], that ants neither need vast amounts of memory, nor an approximation of n, to efficiently discover the treasure.

Yet, from the complexity of the presented algorithms, and from a comparison of the presented algorithms with observations on ant foraging behavior, we conclude that ants probably do not employ our searching strategies. Nevertheless, it must be noted that much of the complexity of our algorithms arise from only two aspects of our model: First, we omit any bounds on the distance between hive and treasure D, and second, we use a very strong adversarial model. Even though these assumptions make sense form a theoretical computer science point of view, it is self-evident that ants in nature do not face such tough problems; for example, it can safely be assumed that ants do not need to walk hundreds of kilometers away from the hive, or that an ant colony that suffers a loss of a large fraction of their ants may rather become extinct, than continuing to efficiently forage.

Constraining only one of those parameters leads to a lot simpler algorithms; for example, by restricting D, the algorithm Uniform Initialization alone would already suffice to discover the treasure within asymptotically optimal runtime. Therefore, we conclude that the chosen model in this thesis serves as nice upper bound on how bad things can get for ants. The model investigated by Feinerman et al., however, can be seen as a lower bound on how bad things can get; e.g., it will certainly not any get better than having unbounded memory. Combining these bounds, we conclude that the computational power lies somewhere in between those bounds, featuring simplicity as in the algorithm presented by Feinerman et al., as well as robustness and actual collaboration, as presented in this thesis.

7.3 Outlook on Interdisciplinary Research

The coalescence of biology and theoretical computer science is still in its very early stages. Even though the collaboration promises to be fruitful, there exists not much work so far. However, the fact that there is a tendency of computer scientists to rather poke around than to actually work together closely with biologists will eventually only lead to discredit instead of interesting results. The following anecdotal evidence clarifies this apprehension.

When we discussed the findings of the work from Feinerman et al., Emek et al., and the ones of this thesis, with some researchers from biology, they indeed agreed on the potential of the collaboration, yet they were not particularly excited about the individual findings. From the perspective of biological research, the question of intelligence is not so much a question of memory size, but more a question of adaptability, i.e., the ability to learn. It is absolutely crucial to elaborate on the term "learning". In classical theoretical computer science, a lot of things are learned during the execution of any algorithm. For example, throughout the execution of our algorithms for the AANTS problem, individual ants learn their role, they learn that a certain other ant died, they learn that a certain segment is discovered, and so on, until they eventually learn the position of the treasure.

However, there is a vast difference between this type of learning and to what biologists refer to as learning. When they speak of learning, they refer to a process of adapting to unforeseeable events; for example, foraging ants may be used to find only one type of food. Then, all of a sudden, they discover different types of food, creating a new problem: The problem of deciding between different food types. It is important to note, that when trying to transfer such a setting from biology to theoretical computer science, one would directly begin with modeling both problems and the transition from the foraging to the decision problem. Yet, such a transfer exactly misses to capture the essence of the question! The essence of the question can colloquially formulated as: How well can ants think out of the box?

Transferring the essential part of the question into theoretical computer science is not only a difficult task to accomplish, but it is already difficult to even correctly phrase it; still, we present a first attempt: To capture the essence of the question of learning, the problem specification and/or the used model must change in an unpredictable way throughout the process of solving the initially specified problem. An immediate thought of modeling such a problem suggests the use of a meta-model, containing the specific model that is subject to change. However, at present, it remains unclear if the introduction of such a meta-model is actually pointing into the right direction or not.

Concluding this section, we discussed that even though the collaboration between biology and theoretical computer science is promising, current approaches

fail to address the essential questions. In order to address these questions, it seems necessary to develop radically new approaches. Introducing a meta-model could be a first suggestion in which direction such a development could go, but it basically remains an open problem.

7.4 Future Work

Since the topic of this thesis lies at the intersection of various problem domains, there are many different directions for future work. In the following, we present some ideas illustrating a few of the directions.

One possibility is to pursue the current path of research, i.e., to alter the model used in the ANTS problem, trying to gain new insights. For example, one could alter the movement behavior such that ants could walk over multiple fields in one time unit. A different idea is to limit the number of ants per field, a constraint that is clearly justified by nature. Also, the AANTS problem could be extended, e.g., one could lift the assumption of synchronous time.

Another idea is to continue to investigate the obstacle ANTS problem introduced in Chapter D. A first step would be to precisely specify the computational power of the ants, e.g., their communication capabilities. Thinking about ants in nature, it probably makes sense to constrain the communication quite rigidly; for example, by allowing only messages of constant size. Then, one could try to extend the described algorithm for one turing machine ant to multiple ants.

However, the possibly most interesting path to follow is the one outlined in Section 7.3, as it seems to be the most promising approach to actually progress in the interdisciplinary research between biology and theoretical computer science.

Bibliography

[1] Emek, Y., Langner, T., Uitto, J., Wattenhofer, R.: Ants: Mobile finite state machines. arXiv preprint arXiv:1311.3062 (2013)

[2] Feinerman, O., Korman, A., Lotker, Z., Sereni, J.S.: Collaborative search on the plane without communication. In: PODC. (2012) 77–86

[3] Feinerman, O., Korman, A.: Memory lower bounds for randomized collaborative search and implications for biology. In: Distributed Computing. Springer (2012) 61–75

[4] Lenzen, C., Radeva, T.: The power of pheromones in ant foraging

[5] Albers, S., Henzinger, M.R.: Exploring unknown environments. SIAM Journal on Computing **29**(4) (2000) 1164–1188

[6] Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. In: Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on, IEEE (1990) 355–361

[7] Dessmark, A., Pelc, A.: Optimal graph exploration without good maps. Theoretical Computer Science **326**(1) (2004) 343–362

[8] Wernli, D.: Grid exploration. PhD thesis, Master thesis, ETH Zürich, Department of Computer Science (2012)

[9] Betke, M., Rivest, R.L., Singh, M.: Piecemeal learning of an unknown environment. Machine Learning **18**(2-3) (1995) 231–254

[10] Gasieniec, L., Pelc, A., Radzik, T., Zhang, X.: Tree exploration with logarithmic memory. In: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics (2007) 585–594

[11] Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph exploration by a finite automaton. Theoretical Computer Science **345**(2) (2005) 331–344

[12] Bender, M.A., Fernández, A., Ron, D., Sahai, A., Vadhan, S.: The power of a pebble: Exploring and mapping directed graphs. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing, ACM (1998) 269–278

[13] Bender, M.A., Slonim, D.K.: The power of team exploration: Two robots can learn unlabeled directed graphs. In: Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on, IEEE (1994) 75–85

[14] Arkin, R.C., Balch, T., Nitz, E.: Communication of behavorial state in multi-agent retrieval tasks. In: Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on, IEEE (1993) 588–594

[15] Dynia, M., ŁopuszaŃski, J., Snhindelhauer, C.: Why robots need maps. In: Structural Information and Communication Complexity. Springer (2007) 41–50

[16] Dynia, M., Korzeniowski, M., Schindelhauer, C.: Power-aware collective tree exploration. In: Architecture of Computing Systems-ARCS 2006. Springer (2006) 341–351

[17] Dynia, M., Kutyłowski, J., auf der Heide, F.M., Schindelhauer, C.: Smart robot teams exploring sparse trees. In: Mathematical Foundations of Computer Science 2006. Springer (2006) 327–338

[18] Fraigniaud, P., Gasieniec, L., Kowalski, D.R., Pelc, A.: Collective tree exploration. Networks **48**(3) (2006) 166–177

[19] Alon, N., Avin, C., Koucky, M., Kozma, G., Lotker, Z., Tuttle, M.R.: Many random walks are faster than one. In: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, ACM (2008) 119–128

[20] Das, S., Flocchini, P., Kutten, S., Nayak, A., Santoro, N.: Distributed exploration of anonymous graphs by multiple agents

[21] Dereniowski, D., Disser, Y., Kosowski, A., Pajak, D., Uznanski, P., et al.: Fast collaborative graph exploration. (2013)

[22] Ortolf, C., Schindelhauer, C.: Online multi-robot exploration of grid graphs with rectangular obstacles. In: Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures, ACM (2012) 27–36

[23] Dudek, G., Jenkin, M., Milios, E., Wilkes, D.: A taxonomy for swarm robots. In: Intelligent Robots and Systems' 93, IROS'93. Proceedings of the 1993 IEEE/RSJ International Conference on. Volume 1., IEEE (1993) 441–447

[24] Choset, H.: Coverage for robotics–a survey of recent results. Annals of mathematics and artificial intelligence **31**(1-4) (2001) 113–126

[25] Hsiang, T.R., Arkin, E.M., Bender, M.A., Fekete, S.P., Mitchell, J.S.: Algorithms for rapidly dispersing robot swarms in unknown environments. In: Algorithmic Foundations of Robotics V. Springer (2004) 77–94

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

[27] Parker, L.E.: Alliance: An architecture for fault tolerant multirobot cooperation. Robotics and Automation, IEEE Transactions on **14**(2) (1998) 220–240

[28] Agmon, N., Peleg, D.: Fault-tolerant gathering algorithms for autonomous mobile robots. SIAM Journal on Computing **36**(1) (2006) 56–82

[29] Deneubourg, J.L., Goss, S., Franks, N., Sendova-Franks, A., Detrain, C., Chrétien, L.: The dynamics of collective sorting robot-like ants and ant-like robots. In: Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats. (1991) 356–363

[30] Aspnes, J., Ruppert, E.: An introduction to population protocols. Bulletin of the EATCS **93** (2007) 98–117

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

[32] Feinerman, O., Korman, A.: Theoretical distributed computing meets biology: A review. In: Distributed Computing and Internet Technology. Springer (2013) 1–18

[33] Chazelle, B.: Natural algorithms. In: Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics (2009) 422–431

[34] Bruckstein, A., Mallows, C., Wagner, I.: Probabilistic pursuits on the grid. The American mathematical monthly **104**(4) (1997) 323–343

[35] Hirsh, A.E., Gordon, D.M.: Distributed problem solving in social insects. Annals of Mathematics and Artificial Intelligence **31**(1-4) (2001) 199–221

[36] Harkness, R., Maroudas, N.: Central place foraging by an ant (cataglyphis bicolor fab.): a model of searching. Animal Behaviour **33**(3) (1985) 916–928

[37] Wehner, R., Srinivasan, M.V.: Searching behaviour of desert ants, genus cataglyphis (formicidae, hymenoptera). Journal of comparative Physiology **142**(3) (1981) 315–338

[38] Müller, M., Wehner, R.: Path integration in desert ants, cataglyphis fortis. Proceedings of the National Academy of Sciences **85**(14) (1988) 5287–5290

[39] Wehner, R.: Desert ant navigation: how miniature brains solve complex tasks. Journal of Comparative Physiology A **189**(8) (2003) 579–588

[40] Wehner, R.: The desert ant's navigational toolkit: procedural rather than positional knowledge. Navigation **55**(2) (2008) 101–114

[41] Grah, G., Wehner, R., Ronacher, B.: Path integration in a three-dimensional maze: ground distance estimation keeps desert ants cataglyphis fortis on course. Journal of experimental biology **208**(21) (2005) 4005–4011

[42] Knaden, M., Wehner, R.: Nest mark orientation in desert ants cataglyphis: what does it do to the path integrator? Animal behaviour **70**(6) (2005) 1349–1354

[43] Cheng, K., Shettleworth, S.J., Huttenlocher, J., Rieser, J.J.: Bayesian integration of spatial information. Psychological bulletin **133**(4) (2007) 625

[44] Graham, P., Philippides, A., Baddeley, B.: Animal cognition: Multi-modal interactions in ant learning. Current Biology **20**(15) (2010) R639–R640

[45] Müller, M., Wehner, R.: Path integration provides a scaffold for landmark learning in desert ants. Current Biology **20**(15) (2010) 1368–1371

[46] Steck, K., Knaden, M., Hansson, B.S.: Do desert ants smell the scenery in stereo? Animal Behaviour **79**(4) (2010) 939–945

[47] Buehlmann, C., Hansson, B.S., Knaden, M.: Path integration controls nest-plume following in desert ants. Current Biology **22**(7) (2012) 645–649

[48] Buehlmann, C., Hansson, B.S., Knaden, M.: Desert ants learn vibration and magnetic landmarks. Plos one **7**(3) (2012) e33117

[49] Collett, T.S., Graham, P.: Animal navigation: path integration, visual landmarks and cognitive maps. Current Biology **14**(12) (2004) R475–R477

[50] Emek, Y., Smula, J., Wattenhofer, R.: Stone age distributed computing. arXiv preprint arXiv:1202.1186 (2012)

[51] Markou, E., Paquette, M.: Black hole search and exploration in unoriented tori with synchronous scattered finite automata. In: Principles of Distributed Systems. Springer (2012) 239–253

[52] Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Mobile search for a black hole in an anonymous ring. In: Distributed Computing. Springer (2001) 166–179

[53] Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Searching for a black hole in arbitrary networks: optimal mobile agent protocols. In: Proceedings of the twenty-first annual symposium on Principles of distributed computing, ACM (2002) 153–162

[54] Darwin, C.: Origin of certain instincts. Nature **7**(179) (April 1873) 417–418

[55] Habermann, A.N.: Parallel neighbor-sort (or the glory of the induction principle). Technical report, Carnegie-Mellon University, Computer Science Department (1972)

[56] Bar-Eli, E., Berman, P., Fiat, A., Yan, P.: On-line navigation in a room. In: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics (1992) 237–249

Appendix A

Well-Known Formulas and Theorems

Theorem A.1 (Markov's inequality) *For any random variable X and $a > 0$, it holds*

$$P[|X| \geq a] \leq \frac{E[X]}{a}.$$

Theorem A.2 (Chernoff bound) *Let $Y_1, \ldots, Y_n$ be n independent Bernoulli random variables and let $Y := \sum_{i=1}^{n} Y_i$. For any $0 \leq \delta \leq 1$ it holds*

$$P\Big[Y < (1-\delta) \cdot E[Y]\Big] \leq e^{-\frac{\delta^2}{2} \cdot E[Y]}$$

and for $\delta > 0$

$$P\Big[Y \geq (1+\delta) \cdot E[Y]\Big] \leq e^{-\frac{\min\{\delta, \delta^2\}}{3} \cdot E[Y]}.$$

Theorem A.3 (union bound) *Boole's inequality or union bound: For a countable set of events $E_1, E_2, E_3, \ldots$, we have*

$$P[\bigcup_i E_i] \leq \sum_i P[E_i].$$

$\diamond$

Appendix B

Derived Formulas and Theorems

Theorem B.1 (number of fields up to a certain distance) *Let u be a field on the 2-dimensional grid. By $F(d)$ we denote the number of fields on the grid up to distance d from u, i.e., $F(d) := |\{v \mid d(u,v) \leq d\}|$. It holds*

$$F(d) = 2d^2 + 2d + 1.$$

Proof Let $F^*(d)$ denote the number of fields that are in exact distance d to the u.

$d = 0$ The only field is u, i.e., $F^*(0) = 1$.

$d \geq 1$ If we look at the fields in distance d to u, we see that those fields form a square with side length $d + 1$, which is rotated by 45°. See Figure B.1 for an example. Thus we can simply sum the number of fields on each side of the square, subtracting the number of corners (since they are counted twice), which yields $F^*(d) = 4 \cdot (d + 1) - 4 = 4d$.

Since $F(d) = \sum_{i=0}^{d} F^*(i)$, we can now derive an explicit formula for $F(d)$,

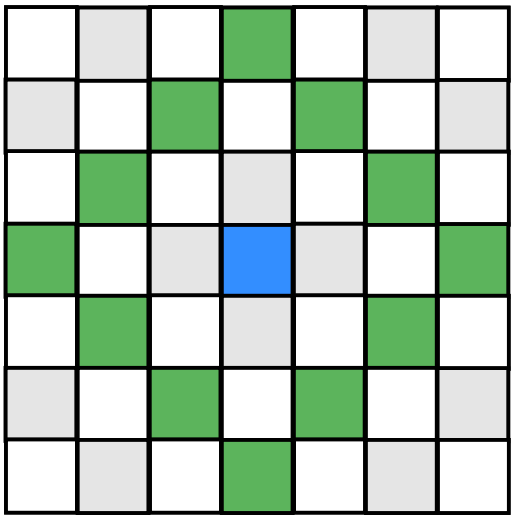


Figure B.1: All fields with distance 3 to the center (blue) are highlighted green.

$$
\begin{aligned}
F(d) &= \sum_{i=0}^{d} F^*(i) && \mid F^*(0) = 1 \\
&= 1 + \sum_{i=1}^{d} F^*(i) && \mid F^*(d) = 4d, d \geq 1 \\
&= 1 + \sum_{i=1}^{d} 4i \\
&= 1 + 4 \sum_{i=1}^{d} i \\
&= 1 + 4 \cdot \left(\frac{d^2 + d}{2} \right) \\
&= 2d^2 + 2d + 1.
\end{aligned}
$$

■

Theorem B.2 (bound on the runtime based on the expected runtime) *Let T be the random variable denoting the runtime of any algorithm $\mathcal{A}$. It holds that*

$$P\Big[T < 2 \cdot E[T]\Big] \geq 1/2 \ .$$

Proof Using Markov's inequality (A.1) with $a = 2 \cdot E[T]$ we get

$$P\Big[T \geq 2 \cdot E[T]\Big] \leq \frac{E[T]}{2 \cdot E[T]} = 1/2 \ .$$

Using the fact that $P[T < t] = 1 - P[T \geq t]$ the claim directly follows that

$$P\Big[T < 2 \cdot E[T]\Big] = 1 - \underbrace{P\Big[T \geq 2 \cdot E[T]\Big]}_{\leq 1/2} \geq 1/2\,. \qquad \blacksquare$$

Theorem B.3 (dividing $c \cdot n$ ants into two teams) *Let $k = c \cdot n$, for a constant $c > 0$. k ants can split into two separate teams in a single round, containing k_1 and k_2 many ants. Both $k_1, k_2 \geq k/3$ w.h.p.*

Proof Each of the k ants tosses a random coin, joining either one of the teams, by switching into the respective state. As this obviously takes only one round, it only remains to show that enough ants join each team w.h.p. Observe that $E[k_1] = E[k_2] = k/2$. Using a chernoff bound (See Theorem A.2) with $\delta = 1/3$ we can derive

$$\begin{aligned} P\left[k_1 < \frac{k}{3}\right] &= P\left[k_1 < \left(1 - \frac{1}{3}\right) \cdot \frac{k}{2}\right] \\ &\leq e^{-\frac{(1/3)^2}{2} \cdot \frac{k}{2}} \\ &= e^{-\frac{k}{36}}. \end{aligned}$$

Since the coin tosses are independent Bernoulli random variables, we know that $P\left[k_1 < \frac{k}{3}\right] = P\left[k_1 > \frac{2k}{3}\right]$. Combined with the fact that $k_2 < \frac{k}{3}$ holds if and only if $k_1 > \frac{2k}{3}$, we get

$$\begin{aligned} P\left[k_1 \geq \frac{k}{3} \wedge k_2 \geq \frac{k}{3}\right] &= P\left[k_1 \in \left[\frac{k}{3}, \frac{2k}{3}\right]\right] \\ &\geq 1 - 2 \cdot e^{-\frac{k}{36}} \\ &\geq 1 - k^{-c}. \end{aligned}$$

$\blacksquare$

Appendix C

Detailed Explanations

C.1 Segment Sweep

This section contains a detailed description of the procedure Segment Sweep. Since Segment Sweep only searches a quarter plane and not the entire grid at once, we use the illustration method for the quarter plane as introduced in Section 4.2.4. First we give a detailed description of the algorithm; since whether or not failures occur during the execution of the algorithm leads to a different execution, we look at both cases individually. Afterwards, in Section C.1.3, we discuss the failure tolerance of Segment Sweep.

C.1.1 Algorithm (Success Case)

A successful segment-attempt consists of four phases: successful segment-attempt, position fix after success, sweep-back and move to next segment.

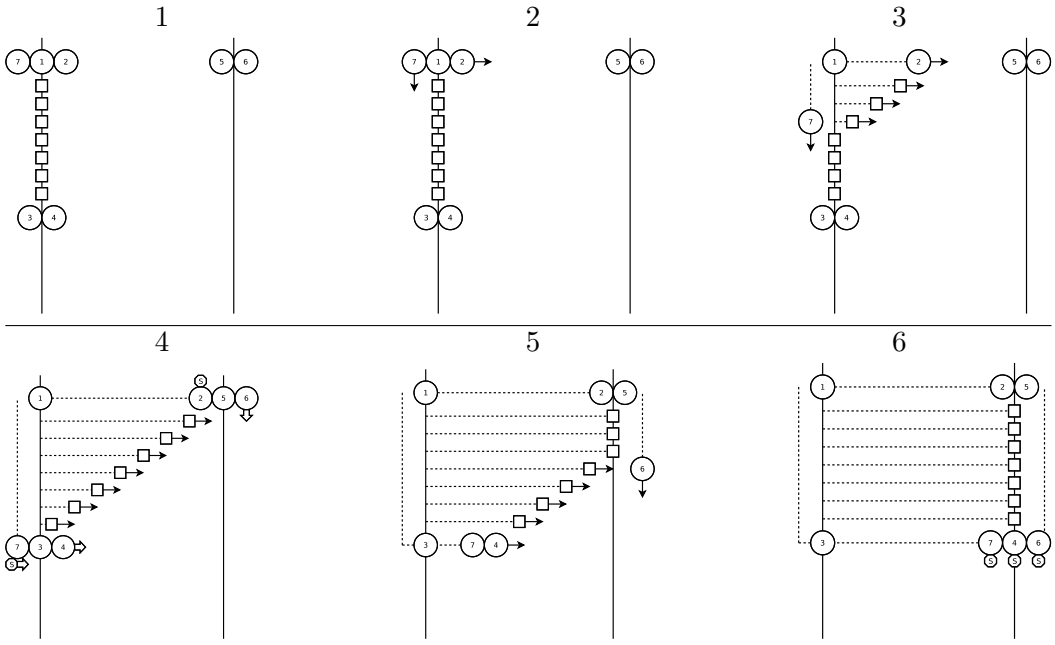
Phase 1: Successful Segment-attempt

Starting from the initial configuration with seven super ants, the super ant number ⑦ starts to walk to super ants ③,④ notifying all the worker ants on the way to start walking their quarter circle; at the same time, super ant ② does its quarter circle, so that ⑥ knows that it must walk on the right ray to catch all the arriving worker ants. Once ⑦ reaches ③,④, super ants ④,⑦ do their quarter circle as well. Both ⑥ and ③,④ stop once they meet each other.

Observe that worker ants start with a time delay of 1, and since each quarter circle is 2 fields longer than the previous one, worker ants will arrive with a time delta of 3. This knowledge can be used by ⑥ to catch the arriving worker ants, namely by walking one step and then waiting two rounds; the worker ant must either arrive exactly at this timestep, or super ant ⑥ knows that the worker ant has failed.

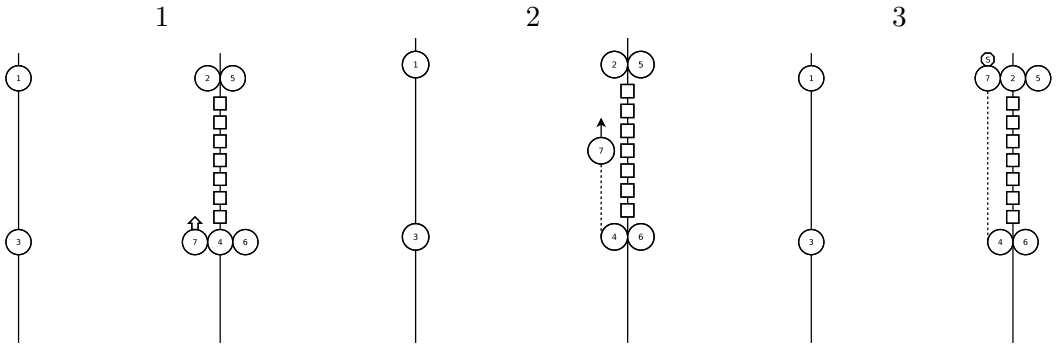
As every stop or turn is guided by one or more super ants (e.g., ⑦ knows when to do the quarter circle when it meets ③,④), all movements can be performed without worrying about failures. Note that ⑥ stores one bit, either all workers have arrived, or at least one worker did not arrive. At the end of the segment-attempt this bit is shared with ④,⑦, meaning that ④,⑥,⑦ know whether or not the segment-attempt was successful.

The following six images illustrate the described procedure. Super ants are represented by circles with the number of the super ant, worker ants are squares. We added arrows and stop-signs to facilitate interpreting of the images.



Phase 2: Position Fix After Success

At the end of phase 1, ④,⑥,⑦ know that the segment was discovered successfully, and super ant ⑦ walks to ②,⑤ in order to prepare for the sweep-back. Upon arrival, ⑦ informs ②,⑤ that the segment-attempt was successful. The following three images illustrate this movement.

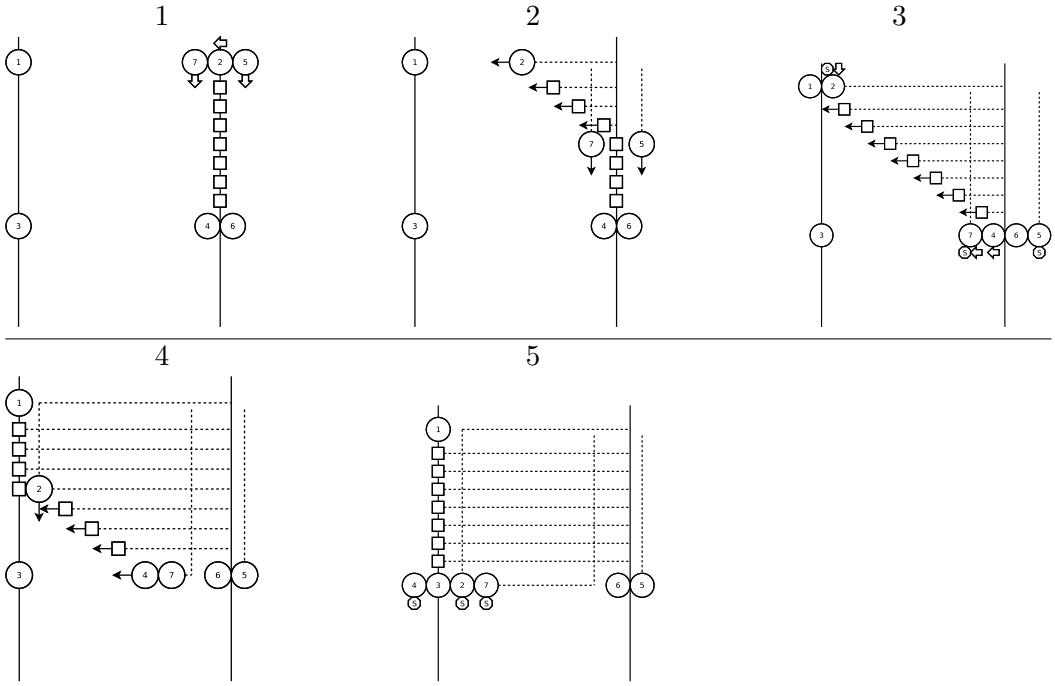


Phase 3: Sweep-back After Success

As now all the super ants on the right ray know that the segment-attempt was successful, some super ants walk back to the left ray, together with all the worker ants. The sweep-back works in the exact same way as the segment-attempt of phase 1, with the only change that some super ants have different roles now.

When arriving on the left ray, the remaining super ants ①,③ get informed about the success. Observe that from this point on all super ants know that the segment was discovered successfully.

In addition to the sweep-back, ⑤ moves to ⑥, in order to prepare for the next segment. The following 5 images illustrate this procedure.



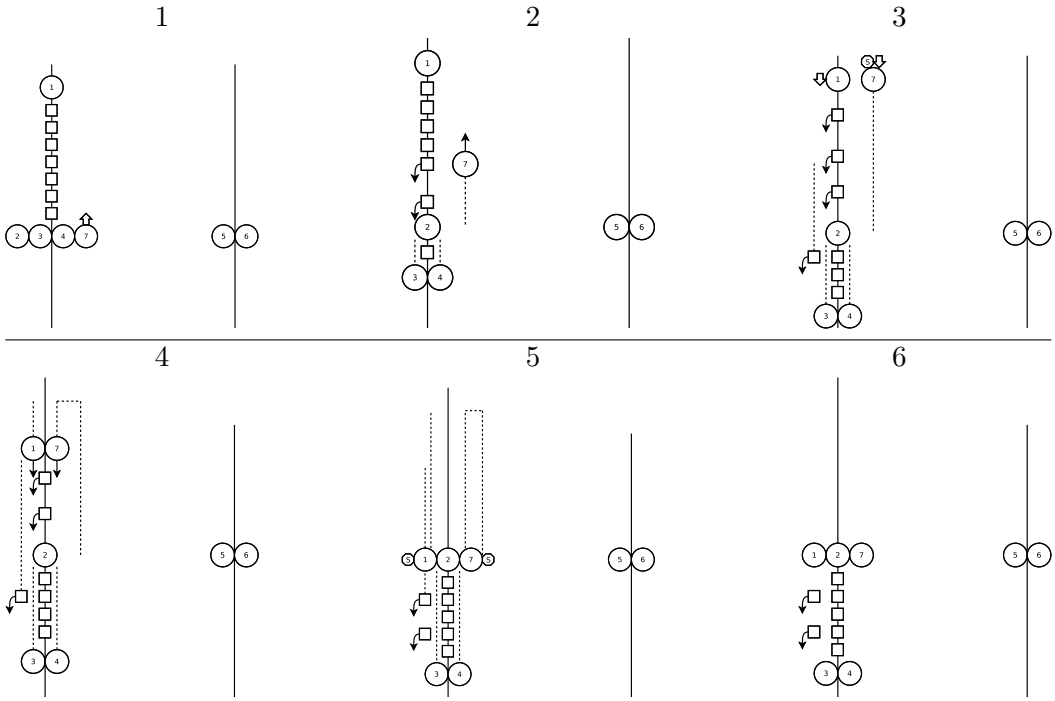
Phase 4: Move to Next Segment

Since the super ants on the right ray are already in position for the next segment, the only thing that remains to be done is to move the super ants and the worker ants on the left ray outwards. This is achieved in the following way: ⑦ starts to walk inwards until it meets ①. On the way it tells every worker ant to start moving outwards until they meet ③,④. When they meet ③,④ they stop, and ③,④ walk one field outwards themselves. When ⑦ meets ①, they both start to walk outwards themselves until they meet ② again.

Note that no two worker ants will be on the same field at the same time, as they start to walk with a time difference of 1 (due to ⑦ walking outwards and notifying one worker ant at a time), and since they walk outwards while being notified in reverse order.

At the time when ①,⑦ meet ②, the first half of worker ants is already in position, and the second half is more outwards than ②, but still moving to ③,④. Note that even though this is not exactly equal to the configuration at the beginning of phase 1 (where all worker ants are waiting in their position), the next segment can be started. This is due to the fact that all moving worker ants will arrive at ③,④ before super ant ⑦, and thus at the time when ⑦ arrives the situation on the field will always be identical to the waiting configuration we assumed at the beginning of phase 1.

The following 6 images illustrate this procedure.



For clarification on how the different worker ants move outwards, we added an illustration; Figure C.1 illustrates the positions where the worker ants will stop walking when they meet ③,④. Note that half of the worker ants must walk at most a distance of x, and the maximum distance a worker ant must walk is $2x$, where x is the number of worker ants.

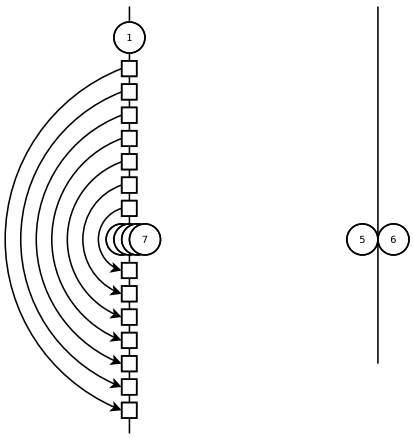


Figure C.1: Overview of the position changes for the different worker ants during the phase *move to next segment*.

C.1.2 Algorithm (Failure Case)

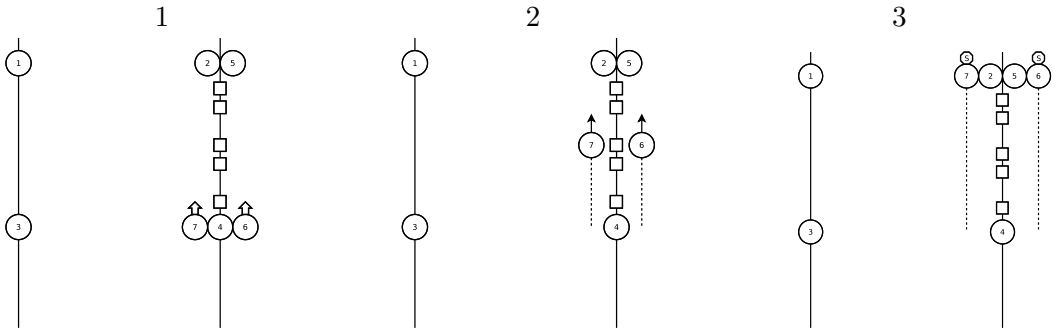
When failures occur during the segment-attempt, the algorithm performs a total of four phases as well: failed segment-attempt, position fix after failure, sweep-back after failure and segment fix odd-even.

Phase 1: Failed Segment-attempt

The segment-attempt is performed identically as in the successful case, as at this time the algorithm obviously cannot distinguish between the successful and the failure case. Hence we omit a more detailed explanation, but note that the difference to the successful segment-attempt is, that **at least** one worker ant must fail **before** it arrives on the right ray; and hence at the end of phase 1, super ants ④,⑥,⑦ know that the segment-attempt failed.

Phase 2: Position Fix After Failure

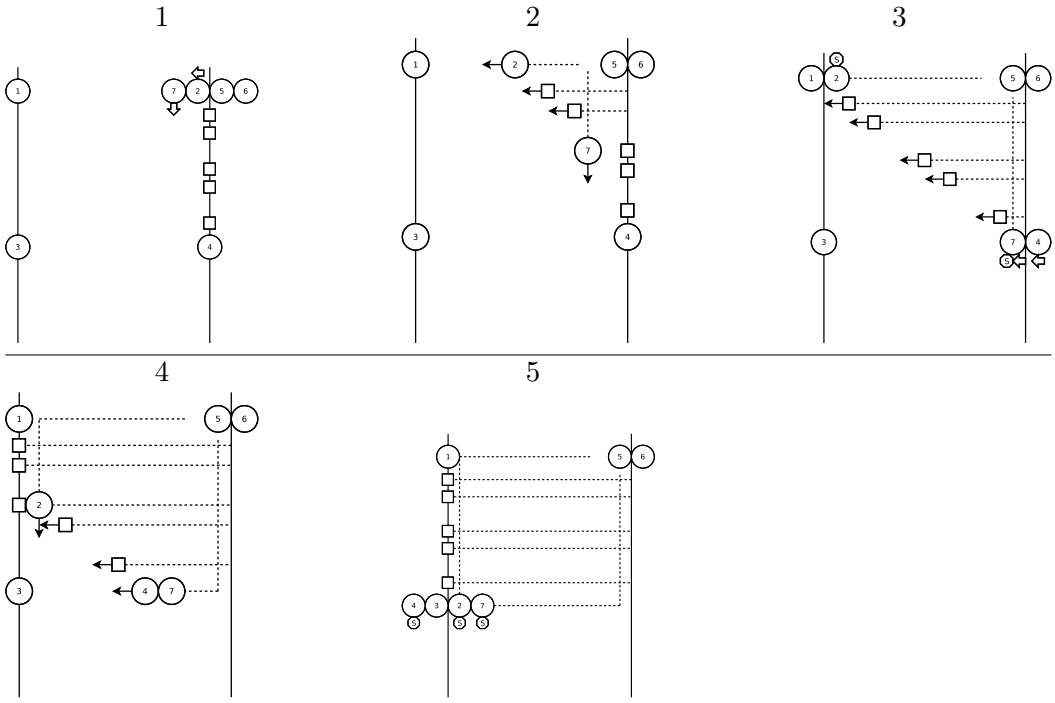
The purpose of this phase is again preparing the super ants for the sweep back, as well as moving super ants on the right ray to prepare for the same segment to be repeated. The only difference to the position fix after success is, that not only ⑦ walks towards ②,⑤, but so does ⑥ as well. The following three images illustrate this procedure.



Phase 3: Sweep-back After Failure

Phase 3 is also nearly identical to phase 3 of the success case, there is only one difference: ⑤ does not move, but rather stays at the current position. This change is necessary as we do want to repeat the segment later on, and hence do not want to move the super ants outwards.

Observe that there is at least one worker ants missing, leading to gaps in the segment of worker ants. The following five images illustrate this procedure.



Phase 4: Segment Fix Odd-even

The last phase is entirely different than phase 4 of the success case, as it tries to a achieve a different goal: Instead of moving the worker ants outwards, we want to condense them, i.e., we want to create a shorter segment of worker ants, that contains no gaps anymore. To achieve this goal, ②,⑦ walk inwards to ①, and on their way they tell all worker ants (and super ant ④) to start repeating *condensing steps*. Once ②,⑦ reach ①, every worker ant is performing condensing steps. Super ant ⑦ turns around and walks slowly (i.e., performs only one step every four timesteps) outwards back to ③. Note that this takes $4x$ time, where x is the number of worker ants that existed at the beginning of the Segment Sweep, i.e., the number of worker ants is now smaller than x. Once ⑦ meets ③, they both start to walk inwards with normal speed, until they meet super ant ④. Super ant ③ stops at this field, and ⑦ keeps walking until it meets ①,②. On its way, ⑦ tells every worker ant (and also super ant ④) to stop performing condensing steps, so that when ⑦ reaches ①,②, the worker ants are all waiting for the next segment.

Let us now look at how the condensing works. Each condensing step takes four timesteps. The first two steps are used by ants that are standing on a field with an **odd** distance to the hive, steps three and four are used by ants on a field with an **even** distance to the hive. In the first (third) timestep, the ant

moves to the inward neighboring field. If there is no other ant on this field, the stays on this field and does nothing on the second (fourth) timestep. If there is already an ant on the field, it steps back onto its old field in step two (four). It therefore follows that in each condensing step an ant walks (at least[1]) one step closer to the hive, if it has no inner neighbor. Note that no ant walks past the innermost super ant, which serves as a boundary.

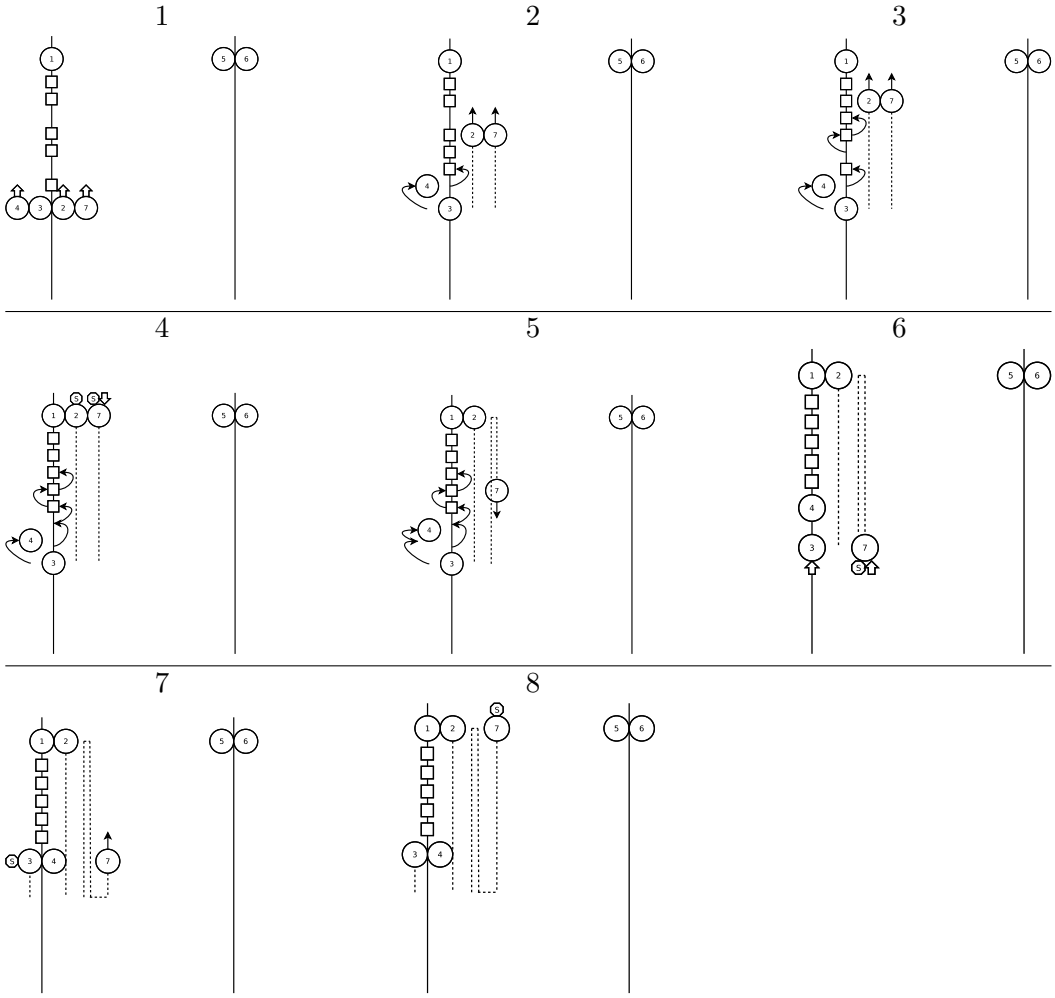
It is clear that all the ants are in one continuous segment, which is adjacent to the innermost super ant, if we repeat this process long enough (and there are no new failures anymore). Observe that once there is a continuous segment, executing more condensing steps will not affect the positions of the ants.

We now show the correctness of the procedure. Assume that there are no failures during the condensing. It quickly becomes evident that the condensing procedure is actually nothing else than sorting an array containing only zeros and ones. The ones are represented by the ants, and the zeros are missing ants. This task can obviously also be solved by an algorithm that sorts an array with arbitrary values. Habermann showed in [55] that any array can be sorted in parallel within at most n steps, where n is the size of the array. Since it takes 4 timesteps to perform an exchange of ants in our model, we can therefore conclude that after at most $4x$ timesteps all ants will be lined up in a continuous segment adjacent to the innermost super ant.

Let us now omit the assumption that there are no failures. The first thing to notice is that a worker ant that fails does not alter the behavior of other ants. Thus, we can look at what happens when an ant fails individually; an ant that fails simply creates a new gap in the segment. If the ant fails early during the procedure, the gap will be fixed as if it existed from the beginning of the condensing procedure. Whereas a failure during the later stages of the procedure might not get fixed, i.e., a gap will remain to exist after the condensing. Observe that this has same effect as if the ant would not have failed **during** the condensing, but rather at the start of the next Segment Sweep. And since such failures are handled by the following iteration, we do not need to assume that there are no failures during the condensing, since we simply defer taking care of the failed ants to the next iteration of Segment Sweep.

The following eight images illustrate the described procedure.

[1] Observe that an ant starting on an odd field can condense twice in one condensing step, if the next two fields are both empty.



C.1.3 Fault Tolerance

In this section we show that failures that occur during the execution of Segment Sweep can only affect the current or the following iteration. Starting in the initial configuration, it is straightforward to see that failures that occur before the respective worker ant reaches the right ray are dealt with by switching to the failure case of the algorithm. Hence it only remains to show that a failure that occurs during phases 2-4 can be dealt with.

Failures in Phase 2

Since – in both the successful and the failed case – phase 2 is only an intermediate step to reposition super ants, failures that occur during such a phase are equivalent to failures that occur at the end of the phase. Therefore, they are

equal to failures that occur at the beginning of phase 3, and do not need to be treated already in phase 2.

Failures in Phase 3

Phase 3 is responsible for bringing the worker ants back to the left ray. Since this procedure simply sends all the worker ants back to the left ray, without being dependent on anything else than super ants, all alive ants will arrive properly. Note that at the end of phase 3 there might be quite a number of gaps in the segment of worker ants: All the gaps that existed already at the start of phase 3, and all the ants that failed during phase 3. Since phase 3 is oblivious to those gaps (in both success and failure case), failure handling can be deferred to phase 4.

Failures in Phase 4: Success Case

In phase 4 of the success case, all worker ants move outwards. Observe that due to the reordering nature of this procedure, all gaps that exist due to failures that occurred up to this point will automatically be fixed. Only if ants fail after they have stopped walking outwards, a new gap will get created that remains until the end of phase 4. Since such a gap is equivalent to an ant failing at the beginning of phase 1 of the following iteration of Segment Sweep, it will be handled in the following iteration.

Failures in Phase 4: Failure Case

Recall that, in phase 4 of the failure case, we use a condensing procedure to fix all gaps that exist in the segment of worker ants. If a worker ant fails early enough in this procedure, the gap will be fixed by the condensing procedure as well. Only if an ant fails during the later stage of the procedure, or at the end of the phase, a gap will remain. Note that this is again equivalent to a failure occurring at the beginning of phase 1 of the following iteration, and thus can be deferred to the following iteration.

Concluding Remarks

We have shown that all failures that occur are either dealt with right away (during the execution of the current iteration of Segment Sweep itself), or they are passed on to the next iteration. Since the next iteration is designed to handle failures that occur within phase 1, Segment Sweep can be repeated infinitely many times without ants performing incorrect behavior.

Appendix D

ANTS and Obstacles

Ants dying is certainly one way to add an element of uncertainty to the ANTS problem. Another idea, that is also motivated by nature, is to add obstacles to the environment; an idea that we investigate in this chapter.

D.1 Obstacles

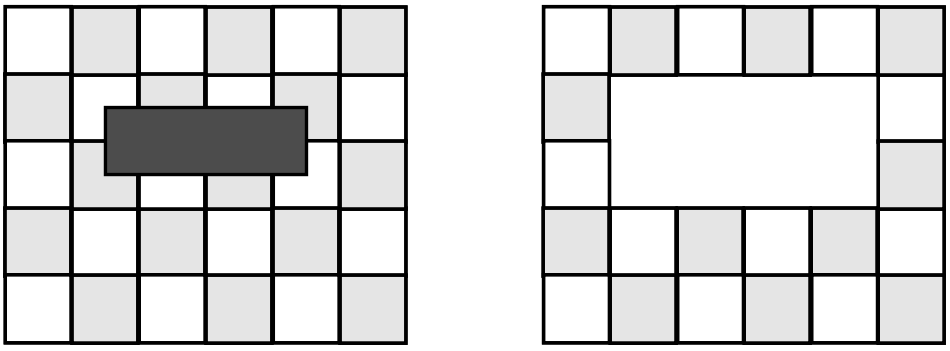
In this section, we investigate what it means to have obstacles on the grid. We discuss constraints that the obstacles must satisfy in order to make the problem solvable, and we put such obstacles into relation with nature. We proceed to give a formal definition of obstacles, and we specify the Obstacle ANTS problem. Additionally, we investigate the computational power that an ant needs, in order to be capable of coping "reasonably" with obstacles.

D.1.1 Modeling Obstacles

Modeling obstacles in a grid can be done be removing those fields from the grid, that are covered by an obstacle. Figure D.1 illustrates this modeling process: On the left, a part of the grid as a human observer would think of it. On the right, the effect of the obstacle on the graph; i.e., the presence of the obstacle has lead to a deletion of fields.

The resulting graph is obviously not a grid anymore. Yet, one must bear in mind that the resulting graph is also not completely arbitrary. Several assumptions about the resulting graph still hold; in particular, every field has between one and four adjacent fields (assuming that the graph is still connected). The knowledge that the underlying graph is a grid can also be used to discover the graph without traveling over all the edges. It follows already from these insights, that, even though the resulting graph is not a grid anymore, the problem does not correspond to the standard setting of graph exploration with arbitrary graphs.

The advantage of constructing a graph in such a way is its natural resemblance to actual environments. Other work on graph exploration often uses arbitrary graphs – or graphs with arbitrary high degree – as a basis (see, e.g., [15, 16]). It is clear, that having so much flexibility when it comes to specifying the degree of the graph, or even the type of the graph, can be very handy to construct algorithms or bounds; nevertheless, one must doubt the relevancy of such work. Bearing those considerations in mind, we decided to model an environment with obstacles in the described way. This modeling process corresponds to the one used in Wernli [8] and to the one from Ortolf and Schindelhauer [22].



(a) The obstacle, as it is perceived by a human observer.

(b) The same part of the grid, as it is perceived by an ant.

Figure D.1: Modeling an obstacle in a grid.

D.1.2 Rectangular Obstacles

There exists at least one property that the obstacles must satisfy: The obstacles should not disconnect the graph. When thinking about other constraints, it quickly comes to mind that, if the obstacles are not constrained in any further way, an adversary could easily construct a complex concave obstacle configuration that would make it extremely complicated for ants to discover the treasure. Since we do not expect ants to fully explore such a complex concave environment anyway, but rather to search food on the relatively open plane – with potentially a handful of obstacles are in their way – we concluded that we need to further restrict the obstacles.

A first approach to constrain obstacles, is to limit their size. This can easily be motivated by nature, where obstacles can be divided into two groups: One group of contains obstacles that are rather small, for example stones, pieces of wood, small puddles of water. It is reasonable to assume that ants will surround those obstacles and search for food behind the obstacles. Then, there exists a group of very large obstacles, e.g., rivers and lakes. It is straightforward to argue

that ants will not try to surround those obstacles and that they will rather ignore the possibility that there is food on the other side on the river. Therefore, it is safe to assume that the type of obstacles that interesting for our problem are relatively small.

However, constraining the size of the obstacles has also a major drawback: The obstacles simply disappear in an asymptotic analysis. Upper bounding the obstacle size with a constant allows even finite state machine ants to trivially surround the obstacles, and from a high-level perspective, the problem would essentially not differ from the ANTS problem at all. Hence, we decided not to constrain the size of the obstacles.

Since we want prevent an adversary from constructing complex concave mazes, we must think about what it means for an obstacle to be concave. It quickly becomes clear, that the only shape of obstacle (on a grid) that is not concave, is a rectangle. As two adjacent rectangular obstacles are modeled as one larger obstacle, it follows that we cannot allow obstacles to be directly adjacent (without breaking the assumption that obstacles are non concave). Thus, we constrain the adversary to only place obstacles that are rectangular and non adjacent. Note that a **finite** grid containing such obstacles has already been studied by Bar-Eli et al. in [56]. In [22] Ortolf and Schindelhauer studied the graph exploration problem with multiple robots in a finite grid with such obstacles.

These considerations are captured by the following definitions.

Definition D.1 (rectangular obstacle) *A* rectangular obstacle $\mathcal{R}$ *on the grid is defined by two fields* $(x,y),(x',y') \in \mathbb{Z}^2$*, where* $x \leq x'$ *and* $y \leq y'$*. A field* $(\alpha,\beta) \in \mathbb{Z}^2$ *is* covered *by* $\mathcal{R}$*, if and only if* $x \leq \alpha \leq x'$ *and* $y \leq \beta \leq y'$ *holds.* ◇

Definition D.2 (surroundable obstacle) *A rectangular obstacle* $\mathcal{R}$ *defined by the fields* $(x,y),(x',y') \in \mathbb{Z}^2$ *is called* surroundable*, if and only if none of the fields inside the rectangle defined by* $(x-1,y-1),(x'+1,y'+1) \in \mathbb{Z}^2$ *is covered by a rectangular obstacle* $\mathcal{R}' \neq \mathcal{R}$ *(see Figure D.2).* ◇

Definition D.3 (disjoint rectangular obstacles) *A set of rectangular obstacles* $\mathcal{S}$ *is called* disjoint*, if and only if all of the obstacles* $\mathcal{R} \in \mathcal{S}$ *are surroundable.* ◇

D.1.3 Problem Statement: Obstacle ANTS

A team of n mobile agents, initially located at the origin of the infinite grid G^∞, collaboratively search for a treasure placed by an adversary at distance D from the hive. The adversary is allowed to provide a (potentially infinite) set of disjoint rectangular obstacles to hinder the search process, with the restriction that the adversary is not allowed to place the treasure on an obstacle.

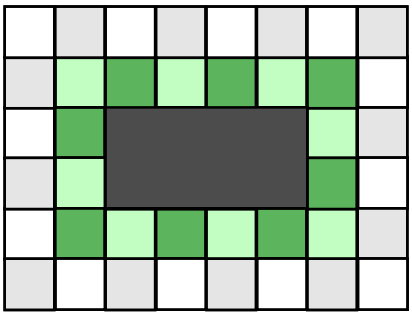


Figure D.2: The black obstacle is *surroundable*, since all of its surrounding fields (green) are not covered by any obstacle.

There are two variants that can be studied: The oblivious adversary, who must provide the set of obstacles at the beginning of the search process, and the adaptive adversary, who is allowed to define the obstacles throughout the search process. Note that the adaptive adversary may not change obstacles that have already been discovered during the search process; i.e., she may defer to specify the location and the size of obstacles for areas that have not yet been discovered by ants. Yet, for every field that has been visited by an ant, and for any field that is blocked by an obstacle and ant has visited an adjacent field of it, the adversary must decide for the obstacle definition, and may not change the obstacle placement anymore. Note that this does not mean that the adversary must fully specify the size of an obstacle as soon as the first ant bumps into it – it is sufficient to declare that there is an obstacle that blocks this field. The adversary is allowed to specify the size of the obstacle iteratively based on the decisions of the ants.

D.1.4 Ants: Computational Power?

It quickly becomes clear that finite state machine ants will not only perform poorly in the Obstacle ANTS problem, but that algorithms that manage to surround obstacles with finite state machines will deviate vastly from actual ant behavior. In order to resemble actual ant behavior at least to some degree, we claim that a single ant should be capable of walking past an obstacle by itself. By "walking past an obstacle", we refer to the following task: Once an ant hits an obstacle, it walks around the obstacle, to the field that lies exactly on the other side of the obstacle. Figure D.3 illustrates this process.

Lemma D.4 *A single pushdown automaton ant can walk past an obstacle.*

Proof When a pushdown automaton ant hits an obstacle, it starts walking along the side of the obstacle in one direction. For each field traveled, it pushes

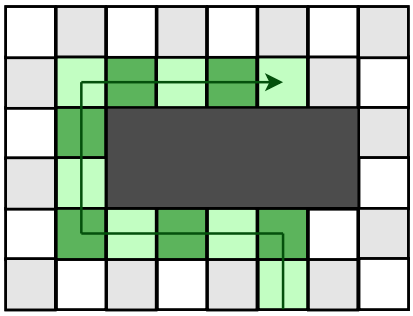


Figure D.3: An ant walks (green path) past an obstacle (black); i.e., it walks exactly to the field to which it would walk, if there was no obstacle.

one symbol onto the stack. Once it arrives at the end of the obstacle, it turns by 90° and continues to walk along the side of the obstacle. When it meets the second corner of the obstacle, it turns again by 90°. It walks back, removing one symbol from the stack after each movement step. When the stack is empty, the pushdown automaton knows that it arrived on the right field. ■

Based on Lemma D.4, we decided that we want to investigate the Obstacle ANTS problem with ants that have at least the computational power of a pushdown automaton.

D.2 A Hard Obstacle Set

In this section, we present an obstacle set, that makes it difficult for ants to discover the treasure efficiently. For this set, we assume that the adversary is allowed to construct obstacles of infinite size. Note that, as throughout this thesis, we again make use of the fact that we can design algorithms for a quarter plane only[1], which simplifies the presentation of the obstacle configuration.

In Figure D.4 we present a small part of the obstacle set, illustrating the idea. The obstacle set contains infinitely long obstacles of width 1, forcing the ants to explore the *passages* between the obstacles sequentially. In a typical graph exploration problem, this would actually not be a problem, since, e.g., depth-first search would efficiently explore this graph. But in the ANTS setting, trying to explore one passage in depth is way too costly, as the treasure could be in the next passage, very close to the hive. As a consequence, any algorithm that discovers the distances from the hive in ascending order must walk back to the diagonal and into the passages all the time.

Observation D.5 *The i-th passage starts at distance $2 + 4i$ from the hive.*

[1] Observe that algorithms must deal with obstacles that lie on the axes carefully.

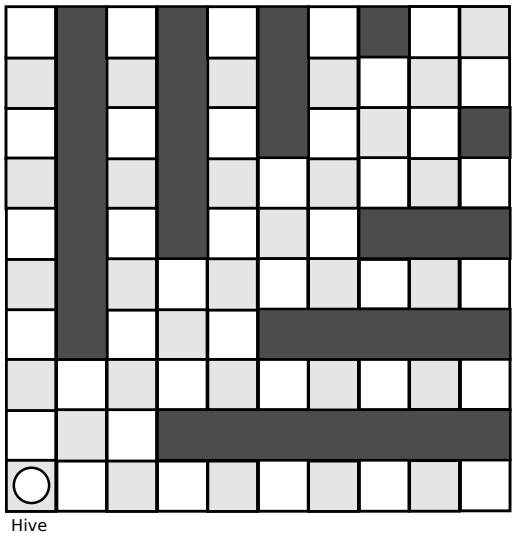


Figure D.4: Part of the hard obstacle set. The obstacles (black) are infinitely long.

Corollary D.6 *If all fields in distance d to the hive should be discovered, an ant must enter the first $\lfloor (d-2)/4 \rfloor$ passages.*

Corollary D.7 *There are at least $2 \cdot \lfloor (d-2)/4 \rfloor \in \Theta(d)$ many fields in distance d, one in every passage.*

Lemma D.8 *Let $\delta_p(d)$ denote the total distance that an ant travels inside passages to discover all fields in distance d to the hive (i.e., without the movement on the diagonal to switch between the different passages). It holds $\delta_p(d) \in \Omega(d^2)$.*

PROOF It follows from Observation D.5 that an ant must travel a distance of $d-(2+4i)$ in the i-th passage to reach distance the field in distance d from the hive. Combining this observation with Corollary D.6 and the fact there are two i-th passages yields

$$
\begin{aligned}
\delta_p(d) &\geq 2 \cdot \sum_{i=0}^{\lfloor \frac{d-2}{4} \rfloor} (d - (2+4i)) \\
&= 2 \cdot \left(\left(\left\lfloor \frac{d-2}{4} \right\rfloor + 1 \right) \cdot (d-2) - 4 \sum_{i=0}^{\lfloor \frac{d-2}{4} \rfloor} i \right) \\
&= 2 \cdot \left(\left(\left\lfloor \frac{d-2}{4} \right\rfloor + 1 \right) \cdot (d-2) - 2 \left(\left\lfloor \frac{d-2}{4} \right\rfloor \cdot \left(\left\lfloor \frac{d-2}{4} \right\rfloor + 1 \right) \right) \right) \\
&= 2 \cdot \left(\left(\left\lfloor \frac{d-2}{4} \right\rfloor + 1 \right) \cdot \left(d - \left(2 + 2 \cdot \left\lfloor \frac{d-2}{4} \right\rfloor \right) \right) \right) \\
&\geq 2 \cdot \left(\left(\left\lfloor \frac{d-2}{4} \right\rfloor + 1 \right) \cdot \left(d - \left(2 + 2 \cdot \frac{d-2}{4} \right) \right) \right) \\
&= 2 \cdot \left(\left(\left\lfloor \frac{d-2}{4} \right\rfloor + 1 \right) \cdot \left(\frac{d-1}{2} \right) \right) \\
&\geq 2 \cdot \left(\left(\frac{d-2}{4} \right) \cdot \left(\frac{d-1}{2} \right) \right) \\
&= \frac{1}{8} \cdot \left(d^2 - 3d + 2 \right).
\end{aligned}
$$

■

Lemma D.9 *Any algorithm that searches the plane by iteratively discovering the distances from the hive in an ascending order*[2]*, is at best* $\Omega(D)$ *competitive.*

PROOF Note that all algorithms presented in this thesis, and the algorithm presented by Emek et al. [1], work in such a way. This is no coincidence, as any efficient algorithm must discover fields that are closer to the hive before it discovers the fields farther away.

We assume that the adversary places the obstacles as described. We know that the ant that discovers all fields in distance d must discover $\Theta(d)$ many fields (see Corollary D.7). From Lemma D.8 we know that it requires $\Omega(d^2)$ time to discover those fields. It therefore follows that every ant discovers a new field at most every $\Omega(d)$ time units.

Observe that there are at least $\Omega(D^2)$ many fields in distance $\Theta(D)$ from the hive, namely those fields that have a distance $d \in \left[\frac{D}{2}, D\right]$ to the hive. Hence, there are $\Omega(D^2)$ many fields, of which every ant discovers at most one every $\Omega(D)$ time units. As there are n ants, we know that the discovery rate is at most

[2] I.e., any algorithm that first discovers all fields in distance 1, then all fields in distance 2, and so on.

$O(n/D)$. Combining the discovery rate with the number of fields that must be discovered yields

$$\begin{aligned}\sum_{t=0}^{T} O(n/D) &= \Omega(D^2)\\ T \cdot O(n/D) &= \Omega(D^2)\\ T &= \Omega(D^3/n).\end{aligned}$$

Recalling the lower bound of $\Omega(D + D^2/n)$ established in Feinerman et al. [2], we see that an algorithm requiring $\Omega(D^3/n)$ time is at best $\Omega(D)$ competitive.■

Based on this lemma, we conclude that the algorithms developed so far cannot easily be converted to work well in the Obstacle ANTS problem. Since both the finite state machine model and the pushdown automaton model[3] are quite limiting, it seems hard (or even impossible) to develop an algorithm that does not discover the distances one after another without sacrificing performance. Thus, we think that it makes sense to consider turing machines in order to solve the Obstacle ANTS problem efficiently.

D.3 A Single Turing Machine Ant

Reasoning about how a single turing machine could efficiently discover the treasure, it makes sense to recall the work on graph exploration. It is well-known that depth first search discovers **any** finite graph in asymptotically optimal runtime. When we try to transfer this idea into our setting, we immediately see that it does not quite work: A turing machine ant would get stuck, e.g., in the first passage, trying to reach its end, and it would never return. Nevertheless, one can derive an algorithm that discovers the treasure in **any** graph, within asymptotically optimal runtime, based on a very similar idea.

In the following we use a routine called "depth limited search", which visits fields in a depth first search fashion, but only visits fields up to a certain distance. Once all fields within this bound are discovered, the ant walks back to the hive. Since every edge that is within the bound is visited at most twice, the runtime of one execution of depth limited search corresponds to the number of edges within the distance bound. Observe that every field has at most 4 edges. Since there are $\Theta(d^2)$ many fields up to distance d (see Theorem B.1), it follows that the runtime of an execution of depth limited search with distance limit d is in $\Theta(d^2)$.

[3]Obviously, two pushdown automaton ants could move together and simulate a turing machine, using the two stacks. We do not consider algorithms that make use of this trick, as it circumvents the purpose of modeling ants as pushdown automata in the first place.

Algorithm 1 Optimal Single Agent Search

```
1: i := 1
2: while true do
3:     Perform Depth-Limited-Search up to distance i
4:     i := 2 · i
5: end while
```

Lemma D.10 *Algorithm 1 discovers the treasure in asymptotically optimal runtime.*

PROOF Using the fact that the runtime of depth limited search up to distance i requires $\Theta(i^2)$ time yields a total runtime T, for which holds that

$$\begin{aligned}
T &= \sum_{i=1}^{\lceil \log_2(D) \rceil} \Theta((2^i)^2) \\
&= \Theta\left(\sum_{i=1}^{\lceil \log_2(D) \rceil} (2^i)^2 \right) \\
&= \Theta\left(\sum_{i=1}^{\log_2(D)+1} (2^i)^2 \right) \\
&= \Theta\left(\sum_{i=1}^{\log_2(D)+1} 4^i \right) \\
&= \Theta\left(\frac{4^{\log_2(D)+2} - 1}{4 - 1} \right) \\
&= \Theta\left(4^{\log_2(D)} \right) \\
&= \Theta\left(4^{\frac{\log_4(D)}{\log_4(2)}} \right) \\
&= \Theta\left(D^{\frac{1}{\log_4(2)}} \right) \\
&= \Theta(D^2).
\end{aligned}$$

Since Algorithm 1 discovers $\Theta(D^2)$ fields within $\Theta(D^2)$ time, the lemma follows. ■

D.4 Multiple Turing Machine Ants

Note that an algorithm that follows the idea of Algorithm 1 does not suffer from the problem that arises from discovering distances one after another. Trying to extend this concept to multiple ants seems therefore promising.

Due to time restrictions, we could not develop such an algorithm yet; thus it is deferred to future work.