Fault-Tolerant ANTS

Tobias Langner, Jara Uitto, David Stolz, and Roger Wattenhofer

ETH Zürich, Switzerland

Abstract. In this paper, we study a variant of the *Ants Nearby Treasure Search* problem, where n mobile agents, controlled by finite automata, search collaboratively for a treasure hidden by an adversary. In our version of the model, the agents may fail at any time during the execution. We provide a distributed protocol that enables the agents to detect failures and recover from them, thereby providing robustness to the protocol. More precisely, we provide a protocol that allows the agents to locate the treasure in time $\mathcal{O}(D + D^2/n + Df)$ where D is the distance to the treasure and $f \in \mathcal{O}(n)$ is the maximum number of failures.

1 Introduction

Ant colonies are a prime example of biological systems that are fault-tolerant. Removing some or even a large fraction of ants should not prevent the colony from functioning properly. In this paper we study the so-called *Ants Nearby Treasure Search (ANTS)* problem, a natural benchmark for ant-based distributed algorithms where n mobile agents try to efficiently find a food source at distance D from the nest. We present a novel distributed algorithm that can tolerate (up to) a constant fraction of ants being killed in the process.

In distributed computing, most algorithms can survive f crash faults by replication. Following this path, each ant can be made fault-tolerant by using $f + 1$ ants with identical behavior, making sure that at least one ant survives an orchestrated attack. However, since we allow $f \in \mathcal{O}(n)$ crash failures, we would be left with merely a constant number of fault-tolerant "super-ants", and a constant number of ants cannot find the food efficiently. As such we have to explore a smarter replication technique, where faulty ants have to be discovered and replaced in a coordinated manner.

In more detail, we study a variation of the ANTS problem, where the n agents are controlled by randomized finite state machines and are allowed to communicate by constant-sized messages with agents that share the same cell. The goal is to locate an adversarially hidden treasure. There is a simple lower bound of $\Omega(D + D^2/n)$ to locate the treasure [10]. This bound is based on the observation that at least one agent has to move to distance D, which takes time $\Omega(D)$, and that there are $\Omega(D^2)$ cells with distance at most D while a single agent can visit at most one new cell per round, which yields the $\Omega(D^2/n)$ term. In previous work, it was shown that the treasure can be located with randomized finite-state machines in optimal time in an asynchronous environment [8]. That approach, however, is rather fragile and requires the agents to be absolutely

F. Kuhn (Ed.): DISC 2014, LNCS 8784, pp. 31–45, 2014.

reliable. The failure of just a single agent can already result in not finding the treasure.

In the model of this paper, $f \in \mathcal{O}(n)$ of the agents can fail at any point in time. However, despite the presence of failures, we show that the treasure can be located efficiently, i.e., we find the treasure in time $\mathcal{O}(D + D^2/n + Df)$. In essence, we implement an error checking mechanism that detects if an agent died. As we keep track of the progress of the search by "remembering" which cells have been searched so far, we can then restart the search while avoiding to search cells that have already been searched.

1.1 Related Work

Searching the plane with n agents was introduced by Feinerman et al. In the original ANTS problem, the agents only communicate in the origin and thus search independently for a treasure [9, 10]. Moreover, the agents are controlled by randomized Turing machines and assuming knowledge of a constant approximation of n, the agents are able to locate the treasure in time $\mathcal{O}(D + D^2/n)$. This model was studied further by Lenzen et al., who investigated the effects of bounding the memory as well as the range of available probabilities of the agents [13]. Protocols in their models are robust by definition as the agents do not communicate outside of origin and thus the failure of an agent cannot affect any other agent.

The main differences between our model and theirs lie in the communication and computation capabilities of the agents. First, we use a significantly weaker computation model: our agents only use a constant amount of memory and are governed by finite automata. Second, our agents are allowed to communicate with each other during the execution. However, the communication is limited to constant sized-messages and only allowed between agents that share the same cell at the same time. The communication and computation model was originally introduced in a graph setting by Emek et al. [8].

Searching the plane is a special case of graph exploration. In the general version of the problem, the task is to visit all the nodes/edges by moving along the edges [1, 6, 7, 12, 16, 17]. In the finite case, it is known that a random walk visits all nodes in expected polynomial time [2]. In the infinite case, a random walk can take infinite time in expectation to reach a designated node.

Another closely related problem is the classic *cow-path* problem, where the task is to find food on a line. It is known that there is a deterministic algorithm with a constant competitive ratio. Furthermore, the spiral search is an optimal algorithm in the 2-dimensional variant [5]. The problem has also been studied in a multi-agent setting [14].

Searching graphs with finite state machines was studied earlier by Fraigniaud et al. [11]. Other work considering distributed computing by finite automata includes for example *population protocols* [3, 4].

1.2 Model

We investigate a variation of the *Ants Nearby Treasure Search (ANTS)* problem, where a set of mobile *agents* explore the infinite integer grid in order to locate a treasure positioned by an adversary. All agents are operated by randomized finite automata with a constant number of states and can communicate with each other through constant-size messages when they are located in the same cell. In contrast to [8] where the agents do not have to deal with robustness issues, our agents can fail at any time during the execution, thus making it much harder to develop correct algorithms for the ANTS problem. In all other aspects, our model is identical to the one of [8].

Consider a set $\mathcal{A}$ of n mobile agents that explore $\mathbb{Z}^2$. All agents start the execution in a dedicated grid cell – the *origin* (say, the cell with coordinates $(0,0) \in \mathbb{Z}^2$). The agents are able to determine whether they are located at the origin or not. The grid cells with either x or y-coordinate being 0 are denoted as *north/east/south/west-axis*, depending on the respective location.

We measure the *distance* $\mathrm{dist}(c, c')$ between two grid cells $c = (x, y)$ and $c' = (x', y')$ in $\mathbb{Z}^2$ with respect to the ℓ_1 norm (a.k.a. Manhattan distance), i.e., $|x - x'| + |y - y'|$. Two cells are called *neighbors* or *adjacent* if the distance between them is 1. In each execution step, an agent located in cell $(x, y) \in \mathbb{Z}^2$ can move to one of the four neighboring cells $(x, y+1), (x, y-1), (x+1, y), (x-1, y)$, or stay still. The four *position transitions* are denoted by the respective cardinal directions N, E, S, W, and the latter (stationary) position transition is denoted by P ("stay put"). We point out that the agents have a common sense of orientation, i.e., the cardinal directions are aligned with the corresponding grid axes for every agent in every cell.

The agents operate in a *synchronous environment*, meaning that the execution of all agents progresses in discrete rounds indexed by the non-negative integers. The runtime of a protocol is measured in the number of rounds that it takes the protocol to achieve its goal/terminate. We fix the duration of one round to be one time unit and thus can take the liberty to use the terms round and time interchangeably.

In comparison to the original ANTS problem, the communication and computational capabilities of our agents are more limited. An agent can only communicate with agents that are positioned in the same cell at the same time. This communication is restricted though: agent a positioned in cell c only senses for each state q whether there exists at least one agent $a' \neq a$ in cell c whose current state is q.

All agents are controlled by the same finite automaton. Formally, the agent's protocol $\mathcal{P}$ is specified by the 3-tuple $\mathcal{P} = \langle Q, s_0, \delta \rangle$, where Q is the finite set of *states*, $s_0 \in Q$ is the *initial state*, and $\delta : Q \times 2^Q \to 2^{Q \times \{N,S,E,W,P\}}$ is the *transition function*. At the beginning of the execution, each agent starts at the origin in the initial state s_0. Suppose that in round i, agent a is in state $q \in Q$ and positioned in cell $c \in \mathbb{Z}^2$. Then, the state $q' \in Q$ of agent a in round $i+1$ and the corresponding movement $\tau \in \{N, S, E, W, P\}$ are dictated based on the transition function δ by picking the tuple (q', τ) uniformly at random from

$\delta(q, Q_a)$, where $Q_a \subseteq Q$ contains state $p \in Q$ if and only if there exists at least one agent $a' \neq a$ such that a' is in state p and positioned in cell c in round i. We assume that the application of the transition function and the corresponding movement occur instantaneously and simultaneously for all agents at the end of the round i.

Adversarial Failures. In contrast to previous work, the agents in our model are not immune to foreign influences and thus can fail at any time during the execution of their protocol. We consider an *adaptive off-line adversary* (sometimes also called omniscient adversary) that has access to all the parameters of the agents' protocol as well as to their random bits. Formally, the adversary specifies for each agent a the *failure time* $t^f(a)$ as the round at the end of which agent a fails. If the adversary does not fail a certain agent a at all, we set $t^f(a) = \infty$. If an agent a fails in round $r = t^f(a)$, then it is removed from the grid as well as the set $\mathcal{A}$; the agent cannot be observed anymore by other agents in any round $r' > r$ (failed agents do *not* leave a corpse behind).

Problem Statement. The goal of ANTS problem is to locate an adversarially hidden *treasure*, i.e., to bring at least one agent to the cell in which the treasure is positioned. The distance of the treasure from the origin is denoted by D while the maximum number of failures that the adversary may cause is denoted by f. We say that a protocol is $g(n)$-*robust* if it locates the treasure w.h.p. for some $f \in \Theta(g(n))$. A protocol that finds the treasure if (up to) a constant fraction of the agents fail is hence n-robust. The goal of this paper is to show that such an n-robust protocol indeed does exist. Therefore, we consider a scenario where $f = \alpha \cdot n$ for a constant α that will be determined later. The performance of a protocol is measured in terms of its runtime, which corresponds to the index of the round in which the treasure is found. Although we express the runtime complexity in terms of the parameters D, n, and f, we point out that neither of these parameters are known to the agents (who in general could not even store them in their constant memory).

2 An n-Robust Protocol

The goal of this section is to develop an n-robust protocol that solves the ANTS problem. In other words, we want to find a protocol that finds the treasure even if a constant fraction of the agents fails. During the remainder of the paper, we use a set of definitions which we shall introduce here. We refer to all cells in distance ℓ from the origin as *level* ℓ. We say that a cell c is *explored* in round r if it is visited by any agent in round r for the first time. Furthermore, a *configuration* of the agents is a function $C : \mathcal{A} \to \mathbb{Z}^2$ that maps each agent $a \in \mathcal{A}$ to a certain cell $c \in \mathbb{Z}^2$.

Giants. A key concept that will be used throughout this paper is the *giant*. A giant is a cluster of k agents that all perform exactly the same operations and

always stay together during the execution of a protocol. If $k > f$, where we recall that f is the maximum number of agents that can fail, we can consider the cluster as a single (giant) agent that cannot be failed by the adversary.

As we design an n-robust protocol, all our giants will consist of $\alpha \cdot n$ agents for a constant $0 < \alpha < 1$. Observe that there can only be a constant number of giants. Since our protocol only requires a constant amount of giants, we proceed to explain how a protocol can create constantly many giants. Consider a protocol that requires g giants, each of size $\Theta(n)$, plus $\Theta(n)$ normal agents. At the beginning of the execution, each agent uniformly at random transitions to one of $g+2$ distinct states, one state for each of the g giants and two additional states for the normal agents. By a simple Chernoff bound argument, it follows that the number of agents per giant is at least $n/(g+3)$ and the number of normal agents is at least $2n/(g+3)$ w.h.p.[1] Hence, a protocol that relies on the survival of its g giants can tolerate $n/(g+3) - 1$ failures and still operate correctly.

2.1 Overview

We describe a protocol that uses 10 giants, which can therefore tolerate up to $f = n/13 - 1$ failures by the above argument. The remaining agents (w.h.p. at least $2n/13$) will be called Explorers as their job is to explore cells in bulk. At any time during the execution, we are guaranteed to have at least $n/13$ surviving Explorers and we will denote this number by n_e.

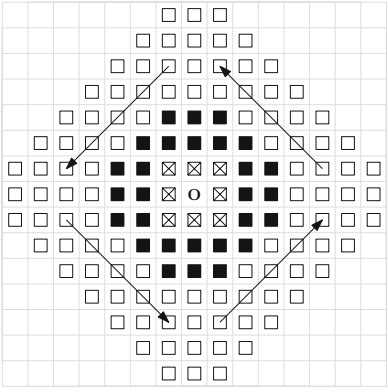

Fig. 1. This figures shows the ring of cells that is supposed to be explored by the ExpoSweep protocol in iteration 1 (crossed boxes), 2 (filled boxes) and 3 (empty boxes). The width of the ring increases by factor of (roughly) two in each iteration and the agents move further outwards.

Our protocol works iteratively and in each iteration, the Explorers explore all cells in a ring around the origin: The Explorers line up along the north axis on a

[1] We say that an event occurs *with high probability*, abbreviated by w.h.p., if the event occurs with probability at least $1 - n^{-\beta}$, where β is an arbitrarily large constant.

segment with a length that depends on the iteration. Then, all Explorers, together with the giants, perform a *sweep* around the origin by moving along the sides of a rectangle. If the exploration of a ring was not successful, meaning that at least one cell in the ring was not explored, the agents regroup and re-explore the ring. If the exploration was successful, the agents move further outwards to prepare for the exploration of the next ring. Then they approximately double the length of the segment (as long as possible) and start a new iteration. Figure 1 gives an illustration of the execution.

2.2 Basis Configuration

All four procedures presented in the following require that at their beginning, the agents form a special configuration, called a *basis*. All procedures also ensure that at their end, the agents are again in a basis. A basis consists of ten giants while all other (non-giant) agents serve as Explorers. An InnerGiant and a CollectGiant are positioned on the east, south, and west axis in the cell with distance d_1 from the origin. On the north axis, an InnerGiant, a StartGiant and a TriggerGiant reside in cell $(0, d_1)$ while an OuterGiant is in cell $(0, d_2)$ with $d_2 > d_1$. All Explorers are located somewhere along the cells between d_1 and d_2 on the north axis. If the parameters are relevant in the context, we write (d_1, d_2)-basis or (d_1)-basis if the second parameter is not relevant (or not known explicitly). See Figure 2 for an illustration.

2.3 Compacting a Segment

The goal of the Compact procedure is to ensure that the Explorers occupy a contiguous segment of cells on the north-axis between InnerGiant and OuterGiant (unless failures occur). If this is not the case, they are compacted towards the origin to form a contiguous, yet shorter, segment.

Let the agents be in a (d_1, d_2)-basis. The procedure Compact is started by the StartGiant, which moves with speed 1/2 (it stays put every second round) towards the OuterGiant and instructs each group of Explorers that it meets to start repeated *compacting steps*. A compacting step consists of two rounds. First, the Explorer moves one cell closer to the origin. If that cell is empty, it stays there and does nothing in the second round, otherwise it moves back to its previous cell in the second round. When an Explorer moves onto the cell containing the InnerGiant, it moves back and *stops* compacting. The same happens if an Explorer moves onto a cell with at least one stopped Explorer.

When the StartGiant has reached the OuterGiant, it instructs the OuterGiant to perform compacting steps as well. Then, the StartGiant waits two rounds and then moves back towards the InnerGiant with speed 1/2 until it arrives there (without further instructing Explorers on the way).

Analysis. The duration of a Compact execution is defined as the time between the StartGiant moving away from the InnerGiant and returning to the InnerGiant again. Observe that if the agents start Compact from a (d_1, d_2)-basis, they

form a (d_1, d_2')-basis at the end for some $d_2' \leq d_2$. Let $E_b = (n_d)_{d_1<d<d_2}$ and $E_e = (n_d)_{d_1<d<d_2'}$ be the sequences of the counts of Explorers on the cells $(0, d)$ at the beginning and the end of the execution of COMPACT, respectively. Further, we denote by $S|_0$ the sub-sequence of the sequence S where each 0-element is removed. Then the following lemma establishes the correctness of COMPACT.

Lemma 1. *If no failures occur during a* COMPACT *execution, then* $E_e = E_b|_0$.

Proof. Let us call the set of Explorers that occupy the same cell at the beginning of a COMPACT execution a *team* and let us index the teams by $1, 2, \ldots, k$ according to increasing distances from the origin. Observe that during COMPACT, the Explorers of a fixed team behave (and move) identically and thus it suffices to examine the individual teams.

By design, team i never overtakes team $i-1$ and moreover only meets team $i-1$ if the latter has already stopped. Team i only stops in a cell that does not contain another stopped team and therefore no two teams will end up at the same cell at the end of the execution. As a team only stops in the cell directly next to the cell that contains either a stopped team or the InnerGiant, the teams will occupy a contiguous segment of cells outwards from the InnerGiant. As the OuterGiant also performs compacting steps, it will end up directly adjacent to the outermost team. Thus, all cells between cell $(0, d_1)$ and $(0, d_2')$ are occupied by the teams 1 to k in that order and the claim follows. □

As an agent moves one step towards the origin every two rounds unless it has reached the cell in which it will stop, all agents have stopped in their target position when the StartGiant arrives back at the InnerGiant.

2.4 Searching a Ring

In this section we introduce the procedure SEGSWEEP (segment sweep) which aims to search all cells in a ring, i.e., a set of consecutive levels. As all our procedures, SEGSWEEP requires the agents to be in a basis. Let the agents be in a (d_1, d_2)-basis.

A SEGSWEEP consists of four QSWEEPs (quarter sweep), one for each quarter-plane, that are executed subsequently. Figure 2 gives an illustration of the different steps of a single QSWEEP. The first QSWEEP (of the north-east quarter-plane) is initiated by the StartGiant which starts moving north towards the OuterGiant along the north axis and while passing the Explorers tells them to diagonally move towards the east-axis by alternatingly moving east and south. As soon as the StartGiant starts moving north, the TriggerGiant moves diagonally towards the east-axis and will meet the east-InnerGiant and east-CollectGiant in cell $(d_1, 0)$. When the TriggerGiant arrives at cell $(d_1, 0)$ in round r, it stops there and instructs the CollectGiant to move outwards (east).

The CollectGiant moves to cell $(d_1 + 1, 0)$ to receive the Explorers that are exploring distance $d_1 + 1$ and thus should arrive in cell $(d_1 + 1, 0)$ soon. Now we have to distinguish two cases. Either at least one Explorer arrives in round $r + 3$ (the Explorer in distance $d + 1$ starts moving towards the east-axis one

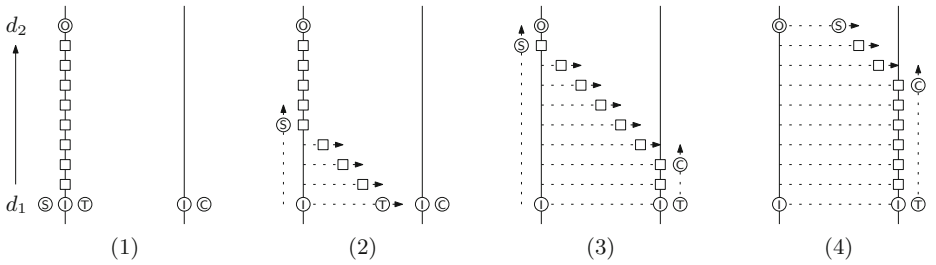

Fig. 2. This figure illustrates different stages of a QSweep. The two (perpendicular) axes between which the QSweep is performed are aligned parallel to each other for the sake of clarity. (1) shows the (d_1, d_2)-basis while in (2) the StartGiant (S) has already sent on their way several Explorers (□) and the TriggerGiant (T). In (3), the TriggerGiant has reached the CollectGiant (C) on the second axis which is now on the way to collect the incoming Explorers and in (4) the StartGiant has reached the OuterGiant (O) on the first axis and is now en route towards meeting the CollectGiant on the second axis.

round later than the Explorer in distance d and has to visit two more cells before arriving there) which means that the search of the north-east quarter-plane in distance $d_1 + 1$ was successful or no Explorer arrives in round $r + 3$, which means that the search was not successful because the team of Explorers was failed. In both cases, the CollectGiant moves one cell outwards in round $r + 4$ to receive the Explorers of level $d_1 + 2$ which are bound to arrive there in round $r + 6$. The CollectGiant continues to move a cell outwards every three rounds and whenever a group of Explorers meet the CollectGiant, they stop in the respective cell.

When the StartGiant arrives at the OuterGiant on the north-axis, the OuterGiant moves inwards (south) and when it arrives at the InnerGiant, it becomes a CollectGiant and stays put. The StartGiant then moves diagonally towards the east-axis and will meet the (moving) CollectGiant in cell $(d_2, 0)$ to notify it that the QSweep is complete upon which the CollectGiant becomes an OuterGiant and stays put. Now the StartGiant moves inwards (west) until it meets the east-InnerGiant and the TriggerGiant. The configuration of the agents is now identical (apart from a 90°-rotation) to the configuration before the first QSweep and thus QSweeps of the south-east, south-west, and north-west quarter-plane can be performed in an analogous fashion.

When the StartGiant arrives at the north-axis for the second time, the last of the four QSweeps is finished. On its way back towards the InnerGiant, the StartGiant now observes whether each cell between the OuterGiant and the InnerGiant contains at least one Explorer. If this is the case, the StartGiant enters a special *complete* state, which, as we will later show, implies that all levels ℓ with $d_1 \leq \ell \leq d_2$ have been explored. Otherwise, the StartGiant enters a special *incomplete* state, meaning that at least one level might not have been explored completely.

Analysis. We say that a SegSweep *begins* in the round in which the StartGiant starts moving towards the OuterGiant from the cell containing the InnerGiant

and TriggerGiant. The SegSweep *ends* when the StartGiant arrives back at the InnerGiant on the north-axis after the fourth QSweep.

Our agents operate in an adversarial environment and thus we need to show that the SegSweep procedure works correctly independent of failures of the agents. Here, that means that all (surviving) agents end up in a (d_1)-basis after a SegSweep and that if the StartGiant enters the complete state, a ring was completely explored. To see the former, note that the design of the procedure ensures that, regardless of potential failures, each Explorer is stopped by a CollectGiant when crossing an axis and the StartGiant and CollectGiant will meet in the cell in distance d_2' on every axis. All other giants are in their original position and thus, after four QSweeps, the agents are again in a (d_1)-basis. The following two lemmas are essential for the correctness of the procedure.

Consider a single execution of SegSweep that starts from a (d_1, d_2)-basis. We call the execution *successful* if at the end, all levels ℓ with $d_1 \leq \ell \leq d_2$ have been explored.

Lemma 2. *If the StartGiant is in the complete state at the end of a* SegSweep*, then the* SegSweep *was successful.*

Proof. Observe that the StartGiant can only enter the complete state if, at the end of a SegSweep, each cell between InnerGiant and OuterGiant contains at least one Explorer. The design of the procedure ensures that an Explorer can only end up in cell $(0, d)$ for $d_1 < d < d_2'$ at the end of a SegSweep if it has started the SegSweep in cell $(0, d)$ and in between explored all cells of level d (and in passing almost all cells of level $d + 1$). As level d_1 and d_2' are explored by TriggerGiant and StartGiant, the claim follows. □

Lemma 3. *If no agent failed during a* Compact *execution and the subsequent* SegSweep*, then the* SegSweep *was successful.*

Proof. The Compact execution ensures that before the first QSweep all cells between InnerGiant and OuterGiant contain at least one Explorer. If no agent fails, all these Explorers will end up in the same cell at the end of the fourth QSweep by design of the procedure. Hence, the StartGiant will observe at least one Explorer in each cell between InnerGiant and OuterGiant and thus enter the complete state. The claim then follows from Lemma 2. □

2.5 Shifting the Segment

In this section, we introduce the procedure Shift, an additional building block that allows the agents to move further outwards from the origin. Its concept is similar to the giant movement during a SegSweep. Shift assumes that all agents form a (d_1, d_2)-basis for some $d_1 < d_2$ and transforms it into a (d_2+1, d_3)-basis for some $d_3 > d_2 + 1$.

The StartGiant moves north towards the OuterGiant and sends the TriggerGiant away to move diagonally to the cell $(d_1, 0)$ on the east-axis, where an InnerGiant/CollectGiant reside. When the TriggerGiant arrives at there, it stays

put and sends the two other giants to move outwards (east) with speed 1/3. When the StartGiant arrives at the OuterGiant, it moves one cell further outwards (to cell $(0, d_2 + 1)$) and then also moves diagonally towards the east-axis. As the speed of the two giants moving outwards on the east-axis is 1/3, they will meet the diagonally moving StartGiant in cell $(0, d_2 + 1)$ and stop there. The StartGiant moves inwards (west) until meeting the TriggerGiant in cell $(0, d_1)$. This process is repeated three times to move the InnerGiant/CollectGiant on the other axis outwards to the cell in distance $d_2 + 1$ from the origin.

When the StartGiant has returned to the north-axis and meets the TriggerGiant in cell $(0, d_1)$, it first sends the TriggerGiant and InnerGiant north in order to stop in cell $(0, d_2 + 1)$, which is one cell outwards of the cell currently occupied by the OuterGiant. Then it moves north with speed 1/2 and whenever it meets a group of Explorers, it instructs them to move north until they find an empty cell. Whenever the OuterGiant observes an Explorer in its cell, it moves one cell north to make sure that it always marks the outermost cell. When the StartGiant arrives at the cell containing the TriggerGiant/InnerGiant, it stops. Now the agents form a $(d_2 + 1, d_3)$-basis for some $d_3 > d_2 + 1$.

2.6 Uniform Splitting

In this section, we introduce the procedure UniSplit (uniform splitting) to line up the agents properly for the SegSweep procedure. Before we go into the implementation details of UniSplit, we briefly explain a few important aspects we have to take into account with the design. First, we do not want the *size* of any segment, i.e., the distance between d_1 and d_2 in a (d_1, d_2)-basis to be much larger than the distance to the treasure D. Since it takes at least time linear in the size of the segment to line the agents up, we might end up using a lot of time lining up unnecessarily many Explorers.

Second, we want to explore the grid as fast as possible. Therefore, we want to line up the Explorers as quickly as possible while maintaining the first property mentioned above. Since we are interested in the asymptotic runtime and the memory bounds are constant, we choose an exponential approach. In other words, we double the segment size after every sweep, as long as there are enough agents available.

Third, we observe that if some level in the SegSweep is explored with a single Explorer, it only takes the adversary one failure to force our protocol to repeat the whole segment. Therefore, as long as we are using segment sizes that are sub-linear to the number of agents, it makes sense to use many agents per level. Thus, the aim of UniSplit is to split the agents along the segment uniformly.

Doubling the Segment Size. Assume that the agents form a (d_1, d_2)-basis. As before, we call all Explorers residing in the same cell a team. To double the segment size, the agents perform the following. The TriggerGiant moves north with speed 1/2 instructing all the cells containing Explorers to perform a *split*. Each Explorer a tosses a fair coin and if the coin shows head, a moves north with speed 1 until it finds the first cell without an Explorer (if the coin shows tail,

they stay put). To ensure that the OuterGiant marks the end of the segment, it always moves north whenever it sees an Explorer. When the TriggerGiant reaches the OuterGiant, it turns around and moves back to the InnerGiant. Once the TriggerGiant reaches the InnerGiant, the agents again form a (d_1)-basis.

We refer to the process of doubling the segment size to as a *pass* of UniSplit. Notice that the segment size k does not necessarily double, i.e., it might be that the new size is $k' \leq 2k$, if some cells contained less than two Explorers. In addition, there might be empty cells along the segment due to unfortunate coin tosses or failures. As the next step, we show that the size of the segment grows by a constant factor in every pass with high probability as long as the team size distribution is "good enough". The key to prove this property is to treat the splitting process as a balls-into-bins experiment.

Consider the situation after the j^{th} pass of UniSplit. The coin tosses performed by the agents so far assign to each agent a bit-sequence of length j. As there are 2^j different possible bit-sequences, one can model our setting as follows: Each of the n agents throws a single ball into the bin corresponding to its bit-sequence while there are 2^j bins altogether. The following lemma establishes that only a constant fraction of the bins is empty w.h.p.

Lemma 4. *Consider a balls-into-bins experiment where $m \geq 4$ balls are thrown uniformly at random into 2^j bins for an integer j with $0 < j < \log m$. Let Z^j be the number of empty bins at the end of the experiment. We have $Z^j < 2/e \cdot 2^j$ w.h.p.*

Proof. Let us first consider the case where $j \leq \kappa \log\log m$ for some $\kappa \geq 2$ to be determined later. Then the number of bins is $\mathcal{O}(\log^\kappa m)$ and the expected number of balls per bin is $\Omega(m/\log^\kappa m)$. Observe that the probability that a fixed bin is empty is $(1-1/2^j)^m \leq e^{-m/2^j}$. By the union bound, the probability that there exists an empty bin is at most $\sum_{i=1}^{2^j} e^{-m/2^j} \in e^{-\Omega(m/\log^\kappa m)}$. Thus we get $\Pr[Z^j \geq 2/e \cdot 2^j] \leq \Pr[Z^j \geq 0] \in e^{-\Omega(m/\log^\kappa m)} \subset o(m^{-\beta})$ for any $\beta > 0$.

Now consider the case where $j > \kappa \log\log m$. Let Z_i^j be the indicator random variable for the event that bin i of 2^j is empty and we have $Z^j = \sum_{i=1}^{2^j} Z_i^j$. A well-known result from balls-into-bins is that instead of dissecting the dependencies between the loads of different bins, one can approximate the scenario well by modeling the load of each bin by an *independent* Poisson random variable [15]. We will denote all random variables derived from this approximation with a tilde and the ones corresponding to the exact scenario without.

Let $\tilde{B}_i^j$ be the random variable indicating the number of balls in bin i and observe that $\Pr[\tilde{B}_i^j = r] = e^{-\mu}\mu^r/(r!)$ for $\mu = m/2^j$ as $\tilde{B}_i^j$ has a Poisson distribution with parameter μ where we observe that $\mu > 1$. Let $\tilde{Z}_i^j$ be the indicator random variable for the event that $\tilde{B}_i^j = 0$ and observe that $\mathbb{E}[\tilde{Z}_i^j] = \Pr[\tilde{B}_i^j = 0] = e^{-\mu} < 1/e$. Let $\tilde{Z}^j = \sum_{i=1}^{2^j} \tilde{Z}_i^j$ be the random variable for the total number of empty bins and by linearity of expectation we get $\mathbb{E}[\tilde{Z}^j] < 2^j/e$. As the $\tilde{Z}_i^j$ are independent by assumption, we can use a Chernoff bound to get $\Pr[\tilde{Z}^j \geq 2/e \cdot 2^j] \leq \Pr[\tilde{Z}^j \geq 2\mathbb{E}[\tilde{Z}^j]] \leq e^{-2^j/(3e)}$. Observe that since $m \geq 4$,

$\kappa \geq 2$, and $j > \kappa \log\log m$, it holds that $\kappa \log m \leq \log^{\kappa} m$ and we get

$$\Pr[\tilde{Z}^j \geq 2/e \cdot 2^j] = e^{-\log^{\kappa} m/(3e)} \leq e^{-\kappa \log m/(3e)} < m^{-\kappa/(3e)} \ .$$

We can now use a result from [15] stating that any event that takes place with probability p in the Poisson approximation takes place with probability at most $pe\sqrt{m}$ in the exact case where m is the number of balls thrown. Hence, we get for the exact case $\Pr[Z^j \geq 2/e \cdot 2^j] < \sqrt{m}e \cdot m^{-\kappa/(3e)} \leq m^{-\beta}$ for any $\beta > 0$ and a large enough value of κ. □

Lemma 5. *Let E be any subset of (surviving) Explorers of size n_e. After the j^{th} iteration of* UniSplit *for $0 < j < \log(n_e)$, the Explorers in E are members of $\Omega(2^j)$ different teams w.h.p.*

Proof. Lemma 4 states that after the j^{th} iteration there are at most $2/e \cdot 2^j$ empty bins w.h.p. Thus there are at least $(e-2)/e \cdot 2^j \in \Omega(2^j)$ many non-empty bins w.h.p., which the Explorers in E must occupy. The claim follows. □

Recall that n_e, the minimum number of surviving Explorers, is guaranteed to be $\Theta(n)$. Thus, Lemma 5 implies that no matter which subset of Explorers the adversary lets survive, these Explorers will be members of $\Omega(2^j)$ different teams after the j^{th} pass of UniSplit for $0 < j < \log(n_e)$ w.h.p.

Corollary 6. *The number of teams after the j^{th} pass of* UniSplit *is $\Omega(2^j)$ for $0 < j < \log(n_e)$ w.h.p.*

2.7 Putting Everything Together

In this section we explain how we can connect the procedures presented in the previous section in order to obtain the n-robust protocol ExpoSweep (exponential sweep) for the ANTS-problem.

The protocol starts with all agents located at the origin. Then the agents create the 10 giants required by SegSweep as described earlier. Now, the agents ensure that the StartGiant, InnerGiant, and TriggerGiant, are located in cell $(0, 1)$, the OuterGiant in cell $(0, 3)$, and an InnerGiant/CollectGiant-pair on the east-, south-, west-axis in the cell with distance 1 to the origin. Observe that this configuration is a (1,3)-basis. Then the agents iteratively perform the protocol described in Algorithm 1.

It is easy to verify that all the aforementioned subroutines of our protocol only require a constant amount of states and therefore, the total number of states required by our protocol is also a constant.

3 Runtime

We begin the runtime analysis by bounding the time needed for any SegSweep in terms of distance to the treasure.

Algorithm 1. ExpoSweep

1. The StartGiant triggers the execution of Compact as described in Section 2.3.
2. The StartGiant triggers the execution of SegSweep as described in Section 2.4. When the SegSweep is finished, there are two cases: If the StartGiant enters the incomplete state, go to step 2. Otherwise, proceed to step 3.
3. The StartGiant triggers the execution of Shift as described in Section 2.5.
4. The StartGiant triggers the execution of UniSplit as described in Section 2.6.

Lemma 7. *If the treasure has not been found at the start of iteration i of* SegSweep *and the agents form a (d_1, d_2)-basis, then $d_1 < D$ and $d_2 \leq 2D$.*

Proof. Observe that the agents only move to a (d_1)-basis after SegSweep has explored all levels $\ell < d_1$, and hence, $d_1 < D$. Assume for contradiction that $d_2 > 2D$. Since d_2 can at most double in UniSplit, there must have been a pass of UniSplit that started from a (d'_1, d'_2) basis, where $d'_1 \leq D \leq d'_2$. Since UniSplit is only performed after a successful execution of SegSweep, the treasure must have already been found. □

Lemma 8. *Any iteration i of* ExpoSweep *before the treasure was found lasts at most $\mathcal{O}(D)$ rounds.*

Proof. By Lemma 7, $d_2 \leq 2D$ for any (d_1, d_2)-basis at the start of iteration i. By looking at the details of the ExpoSweep protocol, we first observe that the time complexity of Compact is clearly $\mathcal{O}(D)$ since the time needed is bounded simply by the time it takes the StartGiant to move from InnerGiant to OuterGiant and back. Second, it is easy to see that each QSweep takes at most $\mathcal{O}(D)$ rounds to finish. Since searching a ring consists of four QSweeps, the second step of our protocol takes $\mathcal{O}(D)$ rounds. A similar argument holds for the Shift procedure. The time complexity of step 4 is again bounded by the time that it takes the TriggerGiant to move back and forth a distance of at most $d_2 \leq 2D$ and thus, the claim follows. □

Now we can combine the previous results to establish the total runtime of the ExpoSweep protocol.

Theorem 9. *The runtime of the* ExpoSweep *protocol is $\mathcal{O}(D + D^2/n + Df)$ for $f = n/13$ w.h.p.*

Proof. By Lemma 8 we know that the furthest level that is searched by the ExpoSweep protocol is $\mathcal{O}(D)$. As the failure of a single agent can cause at most one repetition of a ExpoSweep iteration, the maximum time that it takes the ExpoSweep protocol to recover from the failure of an agent is $\mathcal{O}(D)$. Thus, we can account for all failure-induced runtime costs by an additional term of $\mathcal{O}(Df)$. In the remainder of the proof, we will therefore only bound the runtime of ExpoSweep iterations without any failures.

Let us first examine the case when $D \in o(n)$, which means that the Explorers are still performing splits when the treasure is in range. Consider the i^{th} iteration of ExpoSweep. Using Corollary 6, we can bound the maximum distance

explored by the preceding iterations from below by $d(i) = \sum_{j=0}^{i-1} \Omega(2^j) = \Omega(2^i)$. The treasure will be explored in the smallest iteration i' such that $d(i') \geq D$. Observe that $i' \in c \log D$ for some constant $c > 0$. As iteration i explores at most level $d(i) + 2^i \in \mathcal{O}(2^i)$, we can bound the time required to complete iterations 1 to i' by

$$\sum_{i=0}^{c \log D} \mathcal{O}(2^i) \in \mathcal{O}(D) \ .$$

Now let us consider the case when $D \in \Omega(n)$. By Corollary 6, we know that after $\mathcal{O}(\log n)$ iterations of ExpoSweep, there are $\Omega(n)$ teams of Explorers. Hence, the treasure will be discovered after $\mathcal{O}(D/n)$ additional iterations. By Lemma 8, any iteration takes at most $\mathcal{O}(D)$ rounds. The total runtime is therefore

$$\sum_{i=0}^{c \log D} \mathcal{O}(2^i) + \sum_{i=c \log D+1}^{\mathcal{O}(D/n)} \mathcal{O}(D) = \mathcal{O}(D^2/n) \ .$$

Including the $\mathcal{O}(Df)$ term for the runtime costs caused by agent failures yields the theorem. □

4 Conclusion

In this work we presented an algorithm that solves the ANTS problem in time $\mathcal{O}(D+D^2/n+Df)$ while tolerating $f \in \mathcal{O}(n)$ failures during the execution w.h.p. Our algorithm uses a combination of a constant number of fault-tolerant giants and $\Theta(n)$ Explorer agents, working together. The few "expensive" giants are used to manage the algorithm such that it is fault-tolerant, and the many "cheap" Explorers are responsible for solving the problem efficiently. It is an interesting open question whether one can solve the ANTS problem in a fault-tolerant way without making use of classic replication, and we conjecture that this is not the case, i.e., that some structure like giants is necessary to solve the ANTS problem in the presence of failures.

References

1. Albers, S., Henzinger, M.: Exploring Unknown Environments. SIAM Journal on Computing 29, 1164–1188 (2000)
2. Aleliunas, R., Karp, R.M., Lipton, R.J., Lovasz, L., Rackoff, C.: Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems. In: Proceedings of the 20th Annual Symposium on Foundations of Computer Science (SFCS), pp. 218–223 (1979)
3. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in Networks of Passively Mobile Finite-State Sensors. Distributed Computing, 235–253 (March 2006)
4. Aspnes, J., Ruppert, E.: An Introduction to Population Protocols. In: Garbinato, B., Miranda, H., Rodrigues, L. (eds.) Middleware for Network Eccentric and Mobile Applications, pp. 97–120. Springer (2009)

5. Baeza-Yates, R.A., Culberson, J.C., Rawlins, G.J.E.: Searching in the Plane. Information and Computation 106, 234–252 (1993)
6. Deng, X., Papadimitriou, C.: Exploring an Unknown Graph. Journal of Graph Theory 32, 265–297 (1999)
7. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree Exploration with Little Memory. Journal of Algorithms 51, 38–63 (2004)
8. Emek, Y., Langner, T., Uitto, J., Wattenhofer, R.: Solving the ANTS Problem with Asynchronous Finite State Machines. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014, Part II. LNCS, vol. 8573, pp. 471–482. Springer, Heidelberg (2014)
9. Feinerman, O., Korman, A.: Memory Lower Bounds for Randomized Collaborative Search and Implications for Biology. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 61–75. Springer, Heidelberg (2012)
10. Feinerman, O., Korman, A., Lotker, Z., Sereni, J.S.: Collaborative Search on the Plane Without Communication. In: Proceedings of the 31st ACM Symposium on Principles of Distributed Computing (PODC), pp. 77–86 (2012)
11. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph Exploration by a Finite Automaton. Theoretical Computer Science 345(2-3), 331–344 (2005)
12. Förster, K.-T., Wattenhofer, R.: Directed Graph Exploration. In: Baldoni, R., Flocchini, P., Binoy, R. (eds.) OPODIS 2012. LNCS, vol. 7702, pp. 151–165. Springer, Heidelberg (2012)
13. Lenzen, C., Lynch, N., Newport, C., Radeva, T.: Trade-offs between Selection Complexity and Performance when Searching the Plane without Communication. In: Proceedings of the 33rd Symposium on Principles of Distributed Computing, PODC (2014)
14. López-Ortiz, A., Sweet, G.: Parallel Searching on a Lattice. In: Proceedings of the 13th Canadian Conference on Computational Geometry (CCCG), pp. 125–128 (2001)
15. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)
16. Panaite, P., Pelc, A.: Exploring Unknown Undirected Graphs. In: Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 316–322 (1998)
17. Reingold, O.: Undirected Connectivity in Log-Space. Journal of the ACM (JACM) 55, 17:1–17:24 (2008)