DISS. ETH NO. 22905

Collaboration in Multi-Agent Systems: Adaptivity and Active Learning

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

JARA UITTO

MSc, University of Helsinki, Finland

born on 4.11.1987

citizen of
Finland

accepted on the recommendation of

Prof. Dr. Roger Wattenhofer, examiner
Prof. Dr. Yuval Emek, co-examiner
Dr. Amos Korman, co-examiner

2015

Abstract

Communication is a powerful tool that enables the operation of a variety of old and new systems, such as mobile networks, trade, and human relationships to name a few. Such systems consist of many actors that, in one way or another, communicate with each other to reach a common goal. In this dissertation, we study two rather different examples of such *multi-agent* systems, namely, we explore the realm of recommendation algorithms and networks of primitive models of computation.

In the case of recommendation algorithms, the agents are able to share their preferences on some items via a centralized entity that keeps track of the sharing history. The goal is to utilize the sharing history so that the agents can learn which items they could potentially like. In the case that the agents have many preferences in common, learning the preferences can be helpful in finding good items for the agents. However, if the agents do not have any preferences in common, it is hard to utilize the sharing information. With this in mind, we propose a new scheme for competitive analysis of recommendation algorithms, where we compare our online algorithms to offline algorithms, that have limited information about the input. In addition, we propose an online algorithm that finds a good item for every agent in the system that has, up to a polylogarithmic factor, an optimal competitive ratio in terms of our new scheme.

In the second part of the dissertation, we focus on the Stone Age model of computation, where the agents are modeled by finite state machines. First, we study the computation of a maximal independent set (MIS) motived by observations from biology, where cells are known to solve this problem. We extend the study to arbitrary networks, where the nodes are subject to crash failures. We propose a distributed algorithm that is, on top of solving the MIS problem in the presence of failures, able to contain the effects of the failures from the nodes of the network that are not directly connected to the failed nodes.

Second, we study the Ants Nearby Treasure Search (ANTS) problem in a similar model, where the agents are mobile. Our problem setting is motivated by the foraging behavior of real world ants and the goal of our agents is to locate a treasure hidden in an infinite grid. The agents are faced by two different challenges in the environment. One of the challenges

is unexpected deaths (failures) of the agents, which require the agents to replace the dead agents to ensure that the treasure is eventually discovered. The other challenge considers obstructions, that is, the agents have to adapt to arbitrary obstacles in the environment. We show that by means of cooperation, the agents are able to deal with both challenges and only require a small overhead in the time complexity or in the number of agents required to solve the task.

Astratto

La communicazione è uno strumento molto potente che permette la creazione di numerosi sistemi sia nuovi che vecchi, come per esempio reti di communicazioni mobili, commercio e relazioni umane. Questi sistemi sono composti da un numero di attori che, in un modo o l'altro, comunicano tra di loro per compiere un'obiettivo comune. In questa dissertazione, studiamo due esempi molto diversi di questi *sistemi multi-agenti*, cioè esploriamo algoritmi di raccomandazione e reti di modelli primitivi di compiutazione.

Nel caso degli algoritmi di raccomandazione, gli agenti hanno la possibilità di scambiare le loro preferenze a proposito di oggetti tramite un'entità centralizzata che registra le condivisioni. Lo scopo è di utilizzare le preferenze condivise in modo tale che gli agenti imparano le proprie preferenze basate su quelle condivise. Nel caso due entità hanno molte preferenze in comune, imparare le preferenze può aiutare per trovare nuovi oggetti. Se gli agenti non hanno alcuna preferenza in comune è difficile dedurre le possibili preferenze da quelle condivise. Tenendo conto di questo, proponiamo un nuovo schema per l'analisi competitiva per algoritmi di raccomandazione, nel quale confrontiamo algoritmi online con algoritmi offline, che hanno una vista incompleta delle preferenze nel sistema. Inoltre proponiamo un algoritmo online che trova raccomandazioni per ogni entità nel sistema e che sono ottimi, aparte un fattore polilogaritmico, secondo l'analisi competitiva presentata.

Nella seconda parte della dissertazione ci concentriamo sul modello di computazione neolitico nel quale agenti sono modellati come automa a stati finiti. Inizialmente studiamo la computazione di insiemid'indipendenza massimali (IIM) motivati da osservazioni dalla biologia che indicano che cellule sono capaci di trovare una soluzione. Estendiamo lo studio a reti generiche nel quale agenti possono fallire. Proponiamo un algoritmo distribuito che, oltre a trovare soluzioni del problema IIM, riesce anche a isolare effetti provocati da fallimenti di agenti se non direttamente connessi con agenti falliti.

Infine studiamo il problema FORMICHE in un modello simile nel quale gli agenti sono mobili. Il problema è ispirato dal comportamento foraggero di formiche reali, e lo scopo è di trovare una fonte di cibo su una grig-

lia infinita. Gli agenti incontrano due problemi nell'ambiente. Il primo problema è quello della morte improvvisa (fallimento) degli agenti, che necessita rimpiazzare l'agente fallito per assicurare che alla fine il cibo viene trovato. L'altro problema si pone se ci sono ostacoli nell'ambiente che bisogna circumnavigare, richiedendo un'adattamento della strategia all'ambiente. Proviamo che grazie alla cooperazione gli agenti riescono a risolvere entrambi i problemi, necessitando solo di un piccolo incremento nella complessita temporale o nel numero degli agenti per trovare una soluzione.

Acknowledgements

In the following, I would like to express my gratitude to the people that helped me along my PhD studies. First, I would like to thank my supervisor Roger Wattenhofer for granting me the opportunity to work for his research group. In particular, I appreciate the enormous impact that you had on my presentation skills and all the fruitful discussions we had when I was stuck on my research problems. Let us also not forget to mention that I am happy that you (and all the others in DISCO group) were able to tolerate my weird sense of humor for four years.

Then, I would like to thank my co-referees Yuval Emek and Amos Korman for all the effort they put into reviewing my thesis and attending my defense. During my studies, Yuval also guided me through nasty formal proofs and taught me plenty of things that should not be said in the security check when entering the plane to/from Israel.

During my time at the Distributed Computing Group I met many wonderful people that made my experience immemorial. I want to thank Pascal Bissig for all the pleasant images, Philipp Brandes for helping me to get to the Diamond league in Starcraft 2, Sebastian Brandt for teaching the proper way to sit down on a sofa, Christian Decker for sharing my forgetfulness, Raphael Eidenbenz for his turtle shot, Beat Futterknecht for being truly super-efficient, Klaus-Tycho Förster for coming whenever needed, Benny Gächter for keeping up the motivation to go to the Crossfit-box, Michael König for showing how to make essentially anything into a funny(?) joke, Tobias Langner for his big, black, and sturdy bottle-opener, Laura(a) Peer for being a cat-expert, Jochen Seidel for letting me use his bathroom rug, Jasmin Smula for her +1 intelligence hat, David Stolz for his insights combined with the grin, Samuel Welten for involving me in his weather balloon project and teaching me how to boulder around tables, and Stephan Holzer for sharing an office with me during the start of my PhD.

I am in debt to Panda Metaiel and Pandatar Matkustus for solving, more or less, all the tough questions regarding my research topics and for letting me participate in the talks they gave around the world. I would also like to thank my previous supervisors Jukka Suomela and Petteri Nurmi for showing me the way into research and giving me plenty of

advice on how to survive in the academic world.

Furthermore, I would like to thank my mother Pirjo Pernu for her indomitable effort to keep me motivated during all my studies and my siblings Miranna and Jyry for their support. I would also like to thank my good friends Pekka Hiltunen and Niko Ahonen for their unwavering love and making sure that I do not forget everything about Finnish culture by inviting me and my colleagues to lake excursions and metal concerts.

Finally, I am very grateful to my girlfriend (and colleague) Barbara Keller for discovering the connection between my name and the Indian goddess of household and kitchen and letting me express my thereby assumed "spiritual" side. I am looking forward to the transmission of Epsilon.

Contents

1 Collaboration in Multi-Agent Systems **1**

I Competitive Recommendations **5**

2 Introduction **7**
2.1 Related Work 11

3 The Online Algorithm **15**
3.1 Model 15
3.2 The Quasi-offline Algorithm 17
3.3 Online Algorithms 21

4 The Anonymous Algorithm **31**
4.1 Model 32
4.2 Anonymous Recommendations 33
4.3 Learning the Preferences 35
4.4 The Greedy **mssc** Algorithm 44

5 Conclusion **49**

II Adaptivity in the Stone Age Model **51**

6 Fault Tolerance **53**
6.1 Related Work 55
6.2 Model 58
6.3 Maximal Independent Set 61
6.4 The Fixing Component 71
6.5 Lower Bound 75
6.6 Pseudo-Locality 76

7 Mobile Agents **79**
7.1 Related Work 81
7.2 Model 82
7.3 An **n**-Robust Protocol 84
7.4 Runtime 96

8 Labyrinth Search **99**
8.1 Model 100
8.2 Basic Idea 102
8.3 Basic Capabilities 104
8.4 Advanced Procedures 109
8.5 Searching the Plane 116

9 Conclusion **121**

1

Collaboration in Multi-Agent Systems

Collective effort is the main building block of many fundamental systems that exist in the world today and, for example, the modern society as such could not exist without people working together. The need for collaboration between the entities of a system becomes even more evident when looking at examples like sports teams, traffic, ant colonies, or biological cells. The common feature among all of the aforementioned examples is that the system consists of many *entities* or *agents* that wish to reach a certain global or selfish goal. These entities can *communicate* with each other by various means, such as observing each other and sending messages. The communication can be *local*, like in the traffic example, where the entities can only observe the actions of the other entities in the visibility range or *global*, like in the sports team example, where it may be

possible to observe the whole configuration of the game. Furthermore, the goal can only be reached if the entities work together, or at least the entities not cooperating can be prevented from ruining the task for the rest.

One of the intriguing questions related to such multi-agent systems is how much cooperation is required between the agents. On one end of the spectrum is the case where each agent acts alone and independent from other agents, whereas on the other end, every agent reports every step they take to every other agent before proceeding to the next step. In the first part of this thesis, we study this question in the context of recommendation algorithms. Our goal is to find an item for every agent that the agent likes while minimizing the communication needed between the agents.

Intuitively, one can guess which items the agents like by knowing the preferences of other agents. Given that the preferences of people are not independent of each other, which is quite commonly the case, agents can indeed improve the quality of the recommendation algorithm by providing the algorithm with accurate knowledge of their preferences. In the first part of this thesis, we study the amount of knowledge each agent has to contribute to the system until every user has found an item that he or she likes. We show that the similarity between the preferences of the agents directly affects the amount of information we have to learn from the agents. Furthermore, we analyze our algorithms in a competitive manner and show bounds on the runtime, i.e., the number of preferences that have to be queried during the execution, that are tight up to logarithmic factors.

Another interesting question is related to the capabilities of the agents, i.e., what kind of model of computation/communication are the agents employing. For example, it is clear that the communication model that a colony of ants employ to find a new location for their nest is completely different to the one that a sports team uses to communicate their strategy.

In the second part of the thesis, we focus on the individual capabilities of the agents. Our models are based on the so called *Stone Age model* introduced by Emek et al. [35], where the agents are only allowed to communicate with messages that are of constant size with respect to the number of agents and the agents are controlled by finite state automata.

The motivation behind this model is to bring the capabilities of the agents closer to biological systems such as cellular networks, where the agents, i.e., the cells, are likely not to be Turing complete.

This thesis brings this attempt further by extending the Stone Age model to enable the agents to adapt to unpredictable environments and to changes in their environment. First, we study a variant of the stone age model which incorporates crash failures of nodes. We show that even in the presence of crash failures, the agents are able to solve fundamental problems efficiently.

Second, we study a variant that enables the agents to be mobile, which brings the model closer to settings such as the foraging of ants. Since ants are subject to various hazards in the environment, it is crucial for the survival of the nest as a community to be able to tolerate the deaths of many of their members. Our results show that even if a constant factor of the foraging ants die in our model, the ants are still able to locate the food. Furthermore, the runtime is asymptotically optimal in the case that all the ants survive.

Finally, we include obstacles into our foraging model which brings us closer to the field of graph exploration. We restrict our search domain to *labyrinths*, i.e., an infinite grid where any subset of nodes/cells along with their adjacent edges are removed. We show that it is possible for even a small group of ants to locate the food within an arbitrary labyrinth.

Part I

Competitive Analysis of Recommendation Algorithms

2 Introduction

Algorithmic research studies a variety of models that are beyond the traditional input-output paradigm, where the whole input is given to the algorithm. Examples of such unconventional models are distributed, streaming, and multiparty algorithms (where the input is distributed in space), or regret, stopping, and online algorithms (where the input is distributed in time). Not having all the input initially is a drawback, and one will generally not be able to produce the optimal result. Instead, the algorithm designer often compares the result of the restricted online/distributed algorithm with the result of the best offline/centralized algorithm by means of competitive analysis.

However, it turns out, this is sometimes not possible. In particular, if the hidden input contains more information than we can learn within the execution of the algorithm, we might be in trouble. This is generally an issue in the domain of recommendation and active learning algorithms.

In this chapter, we study a purely algorithmic learning process that starts out with zero knowledge. Given an unknown arbitrary binary $n \times m$ matrix, how many entries do we have to query (probe) until we find a 1-entry in each row? Clearly the answer to this question depends on the matrix. If all the entries of the matrix are 1, the task is trivial. On the other hand, if there is only one 1-entry in each row at a random position, the task is hard.

The unknown binary matrix can be seen as a *preference matrix*, which represents the preferences of n users on m items. In particular, a 1-entry at position (i, j) of the matrix indicates that user i likes item j, whereas a 0-entry indicates that user i does not like item j. Thus, row i in the matrix can be seen as the *taste* vector of user i. The goal is to *satisfy* each user by finding a suitable item for her, i.e., to discover at least one 1-entry in the taste vector of each user with as few queries as possible. We call this problem the *ignorant* recommendation problem, since initially, the algorithm is completely ignorant about the taste matrix, and only over time (hopefully) learns about the taste of the users.

Naturally, satisfying a user who does not like any item is impossible and therefore, we assume that the taste vector of any user u, i.e., the row in the preference matrix that corresponds to the preferences of u, contains at least one 1-entry. Furthermore, we observe that there are instances of the ignorant recommendation problem, where any algorithm performs badly. An example of such an input is a matrix with one 1-entry in each row at a random position. In this example, the rows share no mutual information and therefore, the best any algorithm can do is to query random entries of unsatisfied users until all users are satisfied. This indicates that the cost for any algorithm is $\Omega(n \cdot m)$ in the worst case. Therefore, instead of looking only at the worst case input, we analyze our algorithm in a competitive manner.

In a competitive analysis, an offline algorithm that can see the whole input is compared against an online algorithm that has no knowledge of the input. In other words, the offline algorithm knows the taste of each user in advance, while the taste vectors are hidden from the online algorithm. Thus, the offline algorithm can simply recommend each user an item she likes, resulting in a cost of n for any input. On the other hand, if each user likes only one item chosen independently at random, any

(randomized) online algorithm will have to query a single user $\Omega(m)$ times in expectation before finding the item she likes. Therefore, the competitive ratio of any online algorithm against the optimal offline algorithm is $\Omega(m)$. Note that an algorithm that simply reveals every entry of the input matrix achieves this competitive ratio and in this sense, this competitive ratio is trivial.

Since we do not want to change the ignorant recommendation problem, our only hope is to make the non-ignorant competition weaker. What is the strongest model for the offline algorithm that allows reasonable (or non-trivial) results? We study two different models for the offline algorithm, namely the *static* model and the *anonymous* model. To emphasize that we are not comparing our algorithm to the optimal offline algorithm algorithm that sees the whole input matrix, we refer to the offline algorithm in the weaker static and anonymized models as the *quasi-offline* algorithm. Furthermore, we refer to the analysis against the quasi-offline algorithm as *quasi-competitive*.

Definition 2.1 (Quasi-Competitiveness)**.** Let A be an online algorithm. Algorithm A is α-quasi-competitive if for all inputs I

$$c(A(I)) \leq \alpha \cdot c(\mathrm{OPT_q}(I)) + \mathcal{O}(1) \ .$$

where $\mathrm{OPT_q}$ is the optimal quasi-offline algorithm and $c(\cdot)$ is the cost function of A and $\mathrm{OPT_q}$, respectively.

In the static model, we hide the input matrix from the adversary and instead, only show the quasi-offline algorithm a probability distribution D over possible preference vectors and the number of users n. We refer to this variant of the recommendation problem as the static recommendation problem. For every execution of the quasi-offline algorithm, the preference vectors of the n users are chosen independently at random from D. We show that from the perspective of the static quasi-offline algorithm, solving our problem is equivalent to solving the Min Sum Set Cover (**mssc**) problem.

The input of **mssc** is, similarly to the well-known Set Cover problem, a collection of sets, but the output is an ordered list of the sets. This order induces a cost for each element, where the cost is the ordinal of the first set that covers the element. The optimal solution to **mssc** minimizes the

expected cost for a randomly chosen element. To connect our problem to **mssc**, we identify each user with an element and each item with the set of users that like it. Therefore, the problem becomes static in the sense that it does not help the quasi-offline algorithm to adapt the strategy according to the execution history.

In the anonymous model, the quasi-offline algorithm knows the whole taste matrix, but the users are anonymous, i.e., the rows of the taste matrix have been permuted arbitrarily. We call this variant the anonymous recommendation problem. The anonymous setting allows the quasi-offline algorithm to make use of an *adaptive* strategy, in which the ordering of the items, according to which they are recommended to users, can be different to different users and change after each recommendation. The anonymous model is at least as strong as the static model in the sense that any quasi-offline algorithm running in the anonymous model has at most the same runtime as the same algorithm running in the static model with the same input matrix.

Our aim is to first show in Chapter 3 that there exists an online ignorant recommendation algorithm that achieves a quasi-competitive ratio of $\mathcal{O}(\sqrt{n}\log^2 n)$ when compared to an quasi-offline algorithm in the static model. In addition, we show that the corresponding quasi-competitive ratio for any algorithm is $\Omega(\sqrt{n})$.

Then, in Chapter 4, we show that the anonymous and the static models are asymptotically equally strong, that is, given the same input, the runtime of the optimal quasi-offline algorithm in the anonymous model is asymptotically the same as the runtime of the optimal quasi-offline algorithm in the static model.

Furthermore, for the **mssc** problem, it is known that the greedy algorithm is a 4-approximation [17]. We make use of this result when analyzing our online algorithm in the static model by considering the greedy algorithm instead of the optimal quasi-offline algorithm. The relations between the aforementioned problems are illustrated in Figure 2.1.

Finally, consider a matrix where everyone likes item b and there are $m-1$ items that no one likes. The optimal static quasi-offline algorithm simply proposes item b to every user, resulting in a cost of n in total. On the other hand, an online algorithm that does not know which item is the popular one can be forced to do $\Omega(m)$ queries to find a single 1-entry in the

Ignorant Opt — Static Greedy — Static Opt — Anonymous Opt

$\mathcal{O}(\sqrt{n}\log^2 n)$ — $\mathcal{O}(1)$ [17] — $\mathcal{O}(1)$

Figure 2.1: We show that, from an asymptotic perspective, the static model and the anonymous model for the offline algorithm are equally strong. Then, we give an online algorithm that attains a quasi-competitive ratio of $\mathcal{O}(\sqrt{n}\log^2 n)$ against the greedy algorithm for **mssc**, and thus also for the optimal static quasi-offline algorithm as well as the optimal anonymous quasi-offline algorithm.

matrix. This indicates that the number of items dominates the running time of any online algorithm while not affecting the cost of the quasi-offline algorithm. To make the cost of the online algorithm comparable to the quasi-offline algorithm, we assume that the number of items is not much larger than the number of users, i.e., $m \in \mathcal{O}(n)$. We emphasize that besides this restriction and assuming that the input is feasible, i.e., every user likes at least one item, we do not assume anything from the input.

2.1 Related Work

Our problems are variants of online recommendation problems. Learning recommendation systems, where the goal is to get a good estimation of the whole preference matrix, has been studied by Alon et al. [6]. They showed that with high probability[1], one can learn the matrix with little error in polylogarithmic time in a distributed setting given that there are similarities between the preferences of different users. The main connection between their work and ours is the model, i.e., the task is to learn properties of the input matrix by probing the entries of the matrix and to minimize the number of probes. The main difference is in the objective and in the complexity measure. They are learning the matrix to distinguish between different users and to thereby provide good individual rec-

[1] We say that an event occurs *with high probability*, if the event occurs with probability at least $1 - n^{-c}$, where c is an arbitrarily large constant.

ommendations, whereas we are not interested in all the items a user likes and only aim to find some intersecting preferences to find a single item of interest for everyone. Furthermore, we benchmark our algorithms against **mssc**, which overcomes the trivial worst-case lower bound of $\Omega(n^2)$ in the case where there are no mutual information between the rows.

Our recommendation problems are special cases of learning a binary relation. Goldman et al. studied a more general version of the problem, where the task of the learner is to predict given entries in the matrix. They showed that with an arbitrary input, the learner can be forced to make $\Omega(n^2)$ mistakes [48]. They studied the learning task with four different kinds of input sequences: a helpful teacher, random, an adversary, and a case where the learner can choose which entry to look at. Furthermore, Kaplan et al. [56] gave more general bounds for similar learning tasks by converting the input of **mssc** into a set of DNF clauses, where an element belonging to a set corresponds to a term being true in the clause that corresponds to the set.

Awerbuch, Patt-Shamir, Peleg, and Tuttle gave a centralized algorithm that outputs a set of recommendations that satisfies all users with high probability [13]. The idea is to select a committee and learn the preference vectors of the committee members completely. The favorite product of each committee member is then suggested to all remaining users. They note that in the presence of malicious users, the committee based approach has disadvantages. Thus, they also present a distributed algorithm for the recommendation problem that does not use a committee, and show that it is resilient to Byzantine behavior. The connection to our work is the model they use with the basic idea of suggesting the most preferred product to the rest of the users. The main contrast is in the complexity measures. They use the similarities of preferences as a basis of the complexity of their algorithms, whereas we compare the cost of our algorithms to the cost of an (quasi-)offline algorithm. We also study worst-case performances and show approximation guarantees.

In addition, Awerbuch et al. studied reputation systems, which are closely related to recommendation systems [12]. They considered a model where items are liked either by everyone or by no one. The goal is to find for all users an item they like by querying random objects or by asking other users for their preferences. They measure the cost of their

algorithm by the number of entries revealed during the execution. They also considered Byzantine users and restricted access to items. The main difference to our work is in the worst case input. They assume that there is always a possibility of cooperation between users, whereas our analysis always considers an arbitrary feasible input.

The classic result related to the Min Sum Set Cover problem is that the greedy algorithm provides a constant approximation. It was shown by Bar-Noy, Bellare, Halldórsson, Shachnai, and Tamir [17] that the greedy solution is a 4-approximation. Feige, Lovász, and Tetali gave a simpler proof for this result and showed that getting an approximation ratio of $4 - \epsilon$ for any $\epsilon > 0$ is NP-hard [37]. Our work extends their results from the static and offline setting into an adaptive and online setting, where the algorithm is allowed to change its strategy during the execution but is not given full information in the beginning. Online variants of the **mssc** problem have been studied before for example by Munagala et al. [74], who showed that even if the elements contained in the sets are hidden from the algorithm, one can achieve an $\mathcal{O}(\log n)$-approximation.

Also other variations of **mssc** have been considered. As an example, Azar and Gamzu studied ranking problems, where the goal is to maintain an adaptive ranking while learning in an active manner [15]. They provided an $\mathcal{O}(\log(1/\epsilon))$-approximation algorithm for ranking problems, where the cost functions have submodular valuations. Golovin and Krause [49] studied problems with submodular cost functions further and in particular, they considered them in an adaptive environment. Furthermore, **mssc** is not the only classic optimization problem studied in an active or an adaptive environment. There exists work on adaptive and active versions of, for example, the well-known Set Cover [47, 69], Knapsack [28], and Traveling Salesman [51] problems.

The anonymous task we are considering can also be seen as a relaxed version of learning the identities of the users, that is, we wish to classify the unknown users into groups according to their preferences. Since the users are determined by their preferences, this can further be seen as finding a matching between the users and the preferences. The matching has to be perfect, i.e., in the end every user has to be matched to a unique preference. A similar setting was studied in economics, where the basic idea is that each buyer and seller has a hidden valuation on the goods they

are buying or selling and the valuations are learned during the execution. Then the goal is to find a perfect matching between a set of buyers and a set of sellers, where an edge in the matching indicates a purchase between the corresponding agents [20, 60].

In general, the problem of offline algorithms being too powerful is not new. However, the usual approach is to provide the online algorithm with additional power. For example, online algorithms with lookahead into the future have been studied for the list update [3] and bin packing [50] problems. In our case however, the cost of an offline solution is always n regardless of the input and therefore, a competitive analysis does not make sense even if the online algorithm was granted more power. The term competitive in our context was introduced earlier by Drineas et al. [32]. In contrast to us, they measure their competitiveness against the number of rows that the algorithm has to learn to be able to predict the rest.

3

The Online Algorithm

In this chapter, we design an online recommendation algorithm for the ignorant recommendation problem. Given that we are considering only the ignorant recommendation problem in this chapter, we take the liberty to call the ignorant recommendation problem simply the recommendation problem. To keep our analysis simple, we consider the static model for the quasi-offline algorithm, i.e., we compare our solution to the optimal algorithm for the **mssc** problem. In Chapter 4, we show that the same asymptotic bound holds against the anonymous quasi-offline algorithm.

3.1 Model

We begin defining our model by giving a formal description of the recommendation problem. The input for the recommendation problem consists of a set of users $\{u_1, \ldots, u_n\} = U$, a set of items B, and a hidden proba-

bility distribution D, where initially $|U| = n$ and $|B| = m \in \mathcal{O}(n)$. The users are *labeled* by their indices, i.e., a recommendation algorithm can distinguish users from each other. A preference vector of a user is a binary vector of length m with at least one 1-entry. Each user is assigned a hidden preference vector chosen independently at random from D, where D is a probability distribution over all possible preference vectors. For simplicity, we assume that the probabilities of D are rational.

A recommendation algorithm works in rounds. In the beginning of each round, the algorithm is given a user u uniformly at random from the set U. Then the algorithm has to recommend an item $b \in B$ to u, which is equivalent to checking whether u likes b or not. The execution of a round of a recommendation algorithm is divided into three steps:

1. Receive a user $u \in U$ chosen uniformly at random.
2. Recommend an item b to the user u.
3. If u likes b, choose whether or not to remove u from U.

If the algorithm decides to remove user u from U in step 3, then u is labeled as *satisfied*.

From here on, the set U is referred to as the set of *unsatisfied* users. The goal of a recommendation algorithm A is to satisfy all users, i.e., the execution terminates when the set of unsatisfied users U becomes empty. The *cost* of A is the number of queries (rounds) A has to perform in expectation until all users are satisfied. The algorithm is allowed to make computations during all the steps, and all computations are considered free.

The probability distribution D is chosen by an adversary and the choice of the adversary is allowed to depend on the online algorithm. The choice of D directly induces a cost to the optimal quasi-offline algorithm. We emphasize that prior to the execution, an online recommendation algorithm A has no knowledge of either D or the random assignment of the preference vectors and A can only gain information about these parameters by querying the users. We measure the quality of A with respect to its quasi-competitive ratio, i.e., maximum ratio between the cost of the online recommendation algorithm A and the cost of the optimal (static) quasi-offline algorithm.

An important concept throughout the paper is the *popularity* of an item. The popularity of an item is the number of unsatisfied users that like it.

Definition 3.1. Let b be an item. The popularity $|b|$ of item b is the number of unsatisfied users that like this item, i.e.,

$$|b| = |\{u \in U \mid u \text{ likes } b\}| \ .$$

We note that as the input can be interpreted as a $n \times m$ binary matrix, user u can be identified with the index of the corresponding row and item b with the index of the corresponding column in the matrix. From here on $u \in U$ indicates both the element and the index, and similarly for $b \in B$.

3.2 The Quasi-offline Algorithm

As we mentioned before, it is possible to build an example where it will take $\Omega(n \cdot m)$ queries in expectation to satisfy all users for any online algorithm (diagonal matrix for example). We tackle this issue by performing a quasi-competitive analysis on our algorithms, i.e., we compare our algorithms to a quasi-offline algorithm that is provided with the probability distribution D from where the preference vectors of the n users were chosen. We begin this section by showing a connection between the optimal quasi-offline algorithm and the optimal algorithm for the **mssc** problem that allows us to benchmark our online algorithm directly against the optimal algorithm for **mssc**.

We observe that since any preference vector v of user u in the input is picked at random and independently from previous picks, gaining information from other users than u does not help the quasi-offline algorithm to identify u. Therefore, the quasi-offline algorithm does not gain anything from using different recommendation strategies on different users or from recommending more items to u after finding an item u likes. In other words, the recommendation strategy for any user u is an ordered set of items that are successively recommended to u. Therefore, to minimize the total expected cost, it is enough for the quasi-offline algorithm to find an ordered set of items that minimizes the expected cost for a single user.

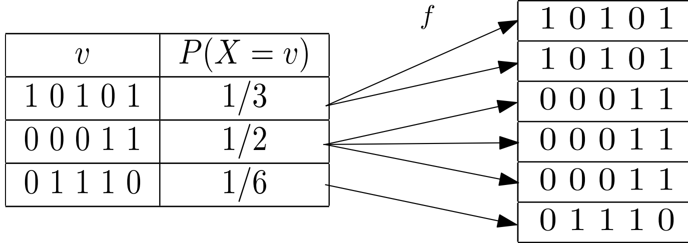


Figure 3.1: An input $I = (D, 6)$ for the quasi-offline algorithm (on the left) is transformed into an instance of **mssc**. The preference vectors that are assigned with non-zero probability are illustrated in the column denoted by v. The column denoted by $P(X = v)$ describes the probability of assigning the corresponding vector to a user, where X is a random variable that obeys distribution D.

To identify an input $I = (D, n)$ of the quasi-offline algorithm with an input of **mssc**, let X be a random variable that obeys distribution D. The idea is to construct a function f that maps I to an input $f(I)$ for **mssc**. Let N be the smallest integer such that $P[X = v] \cdot N$ is an integer for every preference vector v and $N \geq n$. Recall that we assumed the probabilities in D to be rational and thus, N always exists. Now to construct an instance for **mssc**, let us again consider the items as sets and the users as elements. For every preference vector v, let there be $P[X = v] \cdot N$ users that have preference vector v. The construction is illustrated in Figure 3.1.

We denote the ordered set of items chosen by the optimal quasi-offline algorithm by $\mathcal{C}(I) = b_1, b_2, \ldots, b_k$. Consider now the set of users U given by $f(I)$ and let

$$S_i(f(I)) = \{u \in U \mid u \text{ likes } b_i \wedge u \text{ does not like } b_j \text{ for any } j < i\} .$$

We refer to the size of $S_i(f(I))$ as the *disjoint popularity* of b_i.

Recall that the number of occurrences of any preference vector in $f(I)$ is simply scaled up by a factor of N/n compared to the expected number of occurrences of the same vector given by I. Thus, we get that, by using

$\mathcal{C}(I)$, the average number of queries

$$\frac{1}{N}\sum_{i=1}^{k} i \cdot |S_i(f(I))|$$

required to satisfy a user in $f(I)$ equals to the expected number of queries $\mathcal{E}(I)$ required by the optimal quasi-offline algorithm to satisfy a user with a preference vector chosen randomly according to D. To simplify notation, we write $\mathcal{E} = \mathcal{E}(I)$ whenever the exact input is either irrelevant or clear from the context.

Observation 3.2. *Let $I = (D, n)$ be an input of the quasi-offline algorithm. Then $\mathcal{E}(I) = c(f(I))/N$, where $c(\cdot)$ is the cost function of the optimal* **mssc** *algorithm.*

It has been shown that a greedy algorithm, that successively selects sets that cover as many uncovered elements as possible, yields a 4-approximation to the optimal solution for **mssc** [17]. Therefore, we lose only an additional constant factor overhead by comparing our solution to the greedy one instead of the optimal. For the rest of the chapter, we refer to the greedy quasi-offline algorithm for **mssc** as the quasi-offline algorithm.

We use the rest of the section to present general bounds on the introduced concepts, which we will later use in the analysis of our recommendation algorithm. Given Observation 3.2, instead of considering a random outcome of the choices of preference vectors by distribution D given by input I, we will only consider a fixed set of users $f(I)$. Furthermore, since the number of users in $f(I)$ does not affect the average cost, we simply assume that $n = N$. Thus, we omit the input from the notation. First, we give an upper bound for the number of items of a certain disjoint popularity in $\mathcal{C}$.

Lemma 3.3. *Let $r \in \mathbb{R}$. The maximum number of items with disjoint popularity of at least n^r in $\mathcal{C}$ is at most $n^{(1-r)/2}\sqrt{2\mathcal{E}}$.*

Proof. Let ℓ be the number of items with disjoint popularity n^r or larger. Now $\ell \leq k$ and therefore,

$$\mathcal{E} \geq \frac{1}{n}\sum_{i=1}^{\ell} n^r \cdot i = \frac{1}{n} n^r \sum_{i=1}^{\ell} i = n^{r-1} \cdot \frac{\ell^2 + \ell}{2}$$

and thus, $\ell \leq \sqrt{\ell^2 + \ell} \leq \sqrt{2\mathcal{E}} \cdot n^{1/2-r/2}$. □

Next, we give an upper bound for the size of U when given the popularity of the most popular item. The most popular item b^* in round i is the item with maximum popularity, i.e., $|b^*| \geq |b|$ for all $b \in B$. Note that since the popularities of the items can be reduced during the execution, another item might be the most popular in round $i+1$.

Lemma 3.4. *Let $r \in \mathbb{R}$ be such that n^r is the popularity of the most popular item. Then the size of U is smaller than $4\sqrt{\mathcal{E}} \cdot n^{1/2+r/2}$ in expectation and with high probability.*

Proof. To count the total number of unsatisfied users, we count the unsatisfied users that like the items in $\mathcal{C}$. As the popularity of the most popular item is at most n^r, we know that there are at most n^r unsatisfied users that like any single item in $\mathcal{C}$. By Lemma 3.3, initially there are at most $\sqrt{2\mathcal{E}}n^{1/2-r/2}$ items of disjoint popularity n^r or greater in $\mathcal{C}$. Therefore, the total number of unsatisfied users liking these items is at most $\sqrt{2\mathcal{E}}n^{1/2+r/2}$.

To bound the number of users that do not like any of the items with disjoin popularity of at least n^r, we define a random variable X that denotes the number of items we have to suggest to a randomly chosen user. We observe that $\mathbb{E}[X] = \mathcal{E}$ and by Markov's inequality

$$p = \mathbb{P}(X > \sqrt{2\mathcal{E}}n^{1/2-r/2}) \leq \frac{\mathcal{E}}{\sqrt{2\mathcal{E}}n^{1/2-r/2}} \leq \sqrt{\mathcal{E}} \cdot n^{r/2-1/2} \ .$$

Let $X^n = \sum_{i=1}^{n} X_i$, where X_i obeys the same distribution as X and all X_i are independent, denote the random variable that counts the number of users that are not satisfied with the items of popularity n^r or larger. We have $\mathbb{E}[X^n] = pn \leq \sqrt{\mathcal{E}} \cdot n^{r/2+1/2}$. By applying a Chernoff bound, we get that with high probability

$$X^n \leq 2\mathbb{E}[X^n] = 2 \cdot \sqrt{\mathcal{E}} \cdot n^{r/2+1/2} \ .$$

Finally, the total number of unsatisfied users is at most

$$|U| \leq \sqrt{2\mathcal{E}} \cdot n^{1/2+r/2} + 2\sqrt{\mathcal{E}} \cdot n^{1/2+r/2} < 4\sqrt{\mathcal{E}} \cdot n^{1/2+r/2} \ ,$$

in expectation and with high probability. □

3.3 Online Algorithms

In this section, we introduce two online algorithms for the recommendation problem. First, we present an algorithm that achieves an optimal quasi-competitive ratio when restricting ourselves to a case where every user likes exactly one item.

3.3.1 Users Like Exactly One Item

Let us assume that the number of items any user u likes is exactly 1. We observe that the probability of a randomly picked user liking a random item is at least $1/m$. Therefore, by suggesting random items to random users, we get a positive feedback after $\mathcal{O}(m)$ queries in expectation regardless of the number of unsatisfied users. We refer to picking both a user and an item to query at random as *sampling*.

Our algorithm for the easier environment with one item per user is the following. Initially, we start with an empty set of *good* items G. After a positive feedback on item b, we add it to G. Each user is recommended all the items in the set G (once) before they are recommended more random items. The cost of the algorithm is summarized in the following theorem.

Theorem 3.1. *If each user likes exactly one item, there exists a recommendation algorithm that satisfies all users with $\mathcal{O}(n\sqrt{n\mathcal{E}})$ queries in expectation.*

Proof. By Lemma 3.3 the number of items of disjoint popularity of at least one is $\mathcal{O}(\sqrt{n\mathcal{E}})$. Since every user likes at most one item and the items in $\mathcal{C}$ satisfy every user, the maximum number of items that need to be added to G is $\mathcal{O}(\sqrt{n\mathcal{E}})$. Furthermore, attempting to satisfy all the future users with the items from G takes $\mathcal{O}(n\sqrt{n\mathcal{E}})$ queries in total. As the expected number of queries to find a new item by randomly sampling users and items is $\mathcal{O}(m)$, it takes $\mathcal{O}(m\sqrt{n\mathcal{E}})$ queries with random items to discover the items that satisfy all users. Therefore, the cost of the algorithm to satisfy all users is

$$\mathcal{O}(n\sqrt{n\mathcal{E}}) + \mathcal{O}(m\sqrt{n\mathcal{E}}) \in \mathcal{O}(n\sqrt{n\mathcal{E}}) \ .$$

□

A matching lower bound can be found by constructing an example, where there are lots of users that like items of popularity one. These users have to be satisfied by searching the preferred item in a brute force fashion.

Theorem 3.2. *Any online recommendation algorithm needs* $\Omega(n\sqrt{n\mathcal{E}})$ *queries in expectation to satisfy all users in the worst case.*

Proof. Let $H = (D, n)$ be an input to the recommendation problem let there be $|B| = n$ items. In addition, let $1 \leq F \leq n$. The distribution D over the preference vectors is chosen in the following manner: There is one distinguished item $b_p \in B$ and $k = \sqrt{nF}$ items $b_i \neq b_p$, where $1 \leq i \leq k$. Item b_p is liked with probability $p = (n - \sqrt{nF})/n$ and item b_i is liked with probability $1/n$ for all i. Any other item is liked with probability 0. One possible outcome of the preferences of the users in input H is illustrated in Figure 3.2. By Observation 3.2, when $n \to \infty$, we can write average cost to satisfy a single user by the quasi-offline algorithm as

$$\mathcal{E} \leq \frac{1}{n}\left(n - \sqrt{nF} + \sum_{i=2}^{\sqrt{nF}+1} i\right) \in \mathcal{O}(F) \ .$$

Let $U' \subseteq U$ be the set of the users that do not like the popular item b_p. Consider now only the users in U' and let A be a deterministic online algorithm that solves the recommendation problem. Let $u \in U'$ be some user and let $\mathcal{H}$ be the set of all possible outcomes of the random choices of the preference vectors of users in U'. When averaging over $\mathcal{H}$, the average number of queries needed to satisfy u is at least

$$\frac{1}{|B|-1} \cdot \left(\frac{1}{2} \cdot (|B|-1) \cdot |B|\right) = \frac{|B|}{2} = \frac{n}{2} \in \Omega(n) \ ,$$

for any deterministic online recommendation algorithm.

Since the preference vector for u is chosen independently from other users, the expected number of queries needed to satisfy u is also independent from queries to other users. By Yao's principle [86] and since the expected size of $|U'|$ is in $\Omega(\sqrt{n})$, any randomized online recommendation algorithm thus requires $\Omega(n\sqrt{n\mathcal{E}})$ queries in expectation to satisfy all users in U'. □

$$
\begin{array}{r}
n-\sqrt{nF}\left\{\begin{array}{|cccccccccc}
1 & 0 & 0 & & & & & & & \\
1 & 0 & 0 & & & & & & & \\
1 & 0 & 0 & & & & & & & \\
1 & 0 & 0 & & & & & & & \\
1 & 0 & 0 & \dots & & & & & & \\
1 & 0 & 0 & & & & & & & \\
1 & 0 & 0 & & & & & & & \\
 & \vdots & & & & & & & & \\
1 & 0 & 0 & 0 & 0 & & & & &
\end{array}\right. \\
\sqrt{nF}\left\{\begin{array}{|cccccccccc}
0 & 1 & 0 & 0 & 0 & & & & & \\
0 & 0 & 1 & 0 & 0 & & & & & \\
0 & 0 & 0 & 1 & 0 & & & & & \\
 & & & & & \ddots & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}\right.
\end{array}
$$

Figure 3.2: An input matrix that is difficult for any algorithm that is initially oblivious to the input. The first $n - \sqrt{nF}$ rows have a single 1-entry in the first column. Let $r_1, \dots, r_{\sqrt{nF}}$ denote the remaining rows. Then row r_i contains a single 1-entry at position $i + 1$. For any online algorithm, it will take $\Omega(n)$ queries in expectation to find a 1-entry from rows $r_1, \dots, r_{\sqrt{nF}}$ since they have no mutual information with any other rows.

3.3.2 Users with Multiple Preferences

Now we lift the assumption that each user likes exactly one item. The basic idea behind our algorithm for the more general version of the problem is similar to the one preference case. However, by randomly sampling and fixing the first item with a positive feedback, we might end up recommending a bad item to many users that do not like it. As an example, consider an instance where everyone likes one item and there are $n/2$ unpopular items that are liked by $\log n$ users. In this example, it is likely that we find many 1-entries from the unpopular items before recommending the most popular item to everyone.

Therefore, we search for items that are almost as popular as the most popular item within the unsatisfied users. Specifically, we learn the preferences of the users until we get at least $c \log n$ positive reviews on a single item for some constant c. All the sampling information is stored in an integer matrix M, where an entry $M(u, b)$ corresponds to the number of positive feedbacks by user u on item b. The total number of positive feedbacks on a certain item b corresponds to the sum of positive feedbacks on column b. Note that while sampling, users are not removed and thus, the same user can give several positive feedbacks on the same item, i.e., even a single user will eventually give $c \log n$ positive feedbacks on some item.

In addition, we might have lots of items with almost equal popularity that are not liked by the same users, and doing the sampling for all of them successively might be costly. Therefore, after $c \log n$ positive feedbacks on a single item, we use the gained sampling information to select a set of equally popular items to the set of good items instead of just one. To avoid choosing items with overlap, i.e., items that are liked by the same users, we re-estimate the popularities after each choice. A pseudo-code representation of the algorithm is given in Algorithm 3.1. As the sampling and the greedy choices are done successively, we present their pseudo-code as subroutines of Algorithm 3.1. The pseudo-code for the sampling part is given in Algorithm 3.2 and for the greedy part in Algorithm 3.3.

We divide the execution of our algorithm into *phases*. One phase consists of two changes in the state, i.e., sampling until $c \log n$ positive feedbacks are given and the greedy choices have been made. We begin the analysis of the algorithm by showing that each greedy choice made

Algorithm 3.1 Phases(U, B)

```
M ← a zero matrix of size |U| × |B|.
STATE ← sample.
V ← a vector of length n such that V(u) = 1 for all u ∈ U
b* ← an arbitrary item b ∈ B.
while there are unsatisfied users do
    Receive a random user u ∈ U.
    if STATE = sample then
        Run Sampling(u, M, V, b*, U, B).
    else
        Run Greedy(u, M, V, b*, U, B).
    end if
end while
```

Algorithm 3.2 Sampling(u, M, V, b^*, U, B)

```
Choose a random item b ∈ B.
if u likes b, i.e., u(b) = 1 then
    M(u, b) ← M(u, b) + 1.
end if
if Σ_{i=0}^{n-1} M(i, b) ≥ ⌊c log n⌋ then
    b* ← b.
    Set V(u') = 0 for all u' ∈ U.
    STATE ← greedy.
end if
```

Algorithm 3.3 Greedy(u, M, V, b^*, U, B)

```
if V(u') = 1 for all u' ∈ U then
    Set V(u') = 0 for all u' ∈ U.
    Set b* ← b, where Σ_{i=0}^{n-1} M(i, b) is largest, ties broken arbitrarily.
end if
V(u) ← 1.
if u likes b*, i.e., u(b*) = 1 then
    Remove u from U.
    for 0 ≤ j < m do
        M(u, j) ← 0.
    end for
end if
if Σ_{i=0}^{n-1} M(i, j) < (c log n)/4 for all j then
    STATE ← sample.
    Reset M to a zero matrix.
end if
```

by Algorithm 3.3 during a single phase is either within a constant factor from the best one or all the reasonable choices are already made. To do this, we categorize the items by their popularities. An item b belongs to category cat_i if $2^{i-1} \leq |b| < 2^i$. We refer to the upper bound on the popularities of the items in cat_i as the *size* of the corresponding category.

Lemma 3.5. *Let $b \in cat_i$ be the most popular item in the beginning of phase j. Each item chosen greedily by Algorithm 3.3 during phase j is liked by at least 2^{i-4} unsatisfied users with high probability.*

Proof. Let X_b be a random variable that denotes the number of positive feedbacks given on item b during phase j. First, we want to show that the expected value $\mathbb{E}[X_b]$ is close to the amount of sampling required, i.e., $c \log n$, on one item before the greedy part of phase j begins. Let us assume for contradiction that $\mu = \mathbb{E}[X_b] > (3/2)c \log n$. The Chernoff bound states that with high probability, the actual value of X_b is within a constant multiplicative factor from its expected value after enough sampling. More precisely the bound states that

$$\Pr(X_b \leq c \log n) \leq \Pr(X_b < (1 - 1/3)\mu) < \left(\frac{e^{-1/3}}{(2/3)^{2/3}} \right)^{\mu}$$
$$< \left(e^{-1/3}\right)^{c \log n} \in \mathcal{O}\left(n^{-c/3}\right) .$$

This indicates that $X_b > c \log n$ with high probability, which is a contradiction since the sampling stops when any item has more than $c \log n$ positive feedbacks. Thus, $\mathbb{E}[X_b] \leq (3/2)c \log n$ with high probability.

The next step is to show that the number of positive feedbacks on any item $b' \in \mathrm{cat}_{i-4}$ is smaller than $(c \log n)/4$ with high probability. As the popularity of b' is less than $|b|/8$ by the definition of the categories, we have

$$\mu' = \mathbb{E}[X_{b'}] < \frac{\mathbb{E}[X_b]}{8} \leq \frac{3c \log n}{16} .$$

Again using the Chernoff bound, we get that

$$\Pr(X_{b'} > (c \log n)/4) < \Pr(X_{b'} > (1 + 1/3)\mu') \in \mathcal{O}\left(n^{-3c/32}\right) .$$

□

The next step of the analysis is to show that the size of U has decreased by a significant amount after each phase. We do this by showing that after each phase, the most popular item belongs to a smaller category than before running the phase.

Lemma 3.6. *Let i be the largest integer such that cat_i is non-empty. After running one phase of Algorithm 3.2, there are no items left in cat_i with high probability.*

Proof. Let b be the most popular item. Similarly to Lemma 3.5 we can use the Chernoff bound to show that with enough sampling, $c \log n$ is at most within factor 3/2 from $E[X_b]$, i.e., $(3/2)\mathbb{E}[X_b] \geq c \log n$. By the definition of categories we get that $\mathbb{E}[X_{b'}] > \mathbb{E}[X_b]/2$. Therefore, the Chernoff bound can also be used to show that with high probability, the number of positive feedbacks X_b' on any item $b' \in \mathrm{cat}_i$ is more than $\mathbb{E}[X_{b'}] \cdot (3/4)$.

Therefore, with high probability

$$X_b' > \frac{3E[X_b']}{4} > \frac{3\mathbb{E}[X_b]}{8} \geq \frac{c \log n}{4} \ .$$

Since items are chosen greedily until there are no more items with at least $(c \log n)/4$ positive feedbacks, all items from cat_i will eventually be chosen or their popularity will reduce to a lower category during the execution of one phase with high probability. □

3.3.3 The Cost

It follows from Lemma 3.6 that after $\mathcal{O}(\log n)$ phases, the popularity of the most popular item reduces to zero, i.e., all users are satisfied. Since we now have derived the number of phases needed to satisfy all users, it remains to analyze the cost of a single phase. We begin by tackling the sampling part where the dominating factors for the cost are the number of unsatisfied users and the popularity of the most popular item.

Lemma 3.7. *During each phase, Algorithm 3.2 is called $\mathcal{O}(n\sqrt{\mathcal{E}n} \log n)$ times in expectation and with high probability.*

Proof. Consider phase i and let b be the most popular item in the beginning of this phase. The probability of receiving a random user that

likes item b is $|b|/|U|$. By Lemma 3.4, the size of $|U|$ is at most $3\sqrt{\mathcal{E}n|b|}$ in expectation and with high probability. Furthermore, the probability of choosing any specific item randomly is $1/m$. Therefore, the expected amount of times item b is recommended to users that like it after $4m\sqrt{\mathcal{E}n} \cdot c \log n$ queries is at least

$$m4\sqrt{\mathcal{E}n} \cdot c \log n \cdot \frac{1}{m4\sqrt{\mathcal{E}n}} = c \log n \ .$$

Let X_k be the random variable that counts the number of times b is recommended to users that like b after k queries during phase i. By applying a Chernoff bound, we get that

$$\Pr\left(X_k < c \log n\right) < \mathcal{O}\left(n^{-c}\right) \ ,$$

for any $k > 2 \cdot (4m\sqrt{\mathcal{E}n} \cdot c \log n)$. Since

$$2 \cdot (4m\sqrt{\mathcal{E}n} \cdot c \log n) \in \mathcal{O}(n\sqrt{\mathcal{E}n} \log n) \ ,$$

the claim follows. □

The last item needed for the analysis is an upper bound for the cost of the greedy part of the algorithm.

Lemma 3.8. *Running one phase of Algorithm 3.1 costs $\mathcal{O}(n\sqrt{\mathcal{E}n} \log n)$ in expectation and with high probability.*

Proof. By Lemma 3.7 the cost of the sampling part of any phase is $\mathcal{O}(n\sqrt{\mathcal{E}n} \log n)$ in expectation and with high probability after which the greedy state is assumed.

By Lemma 3.5 all the greedily chosen items are liked by at least 2^{i-4} users with high probability, where i is the index of the largest category with non-empty set of items. By Lemma 3.4 there are at most $\mathcal{O}(\sqrt{\mathcal{E}n2^i})$ unsatisfied users left in the beginning of the phase. Therefore, after making $\mathcal{O}(\sqrt{\mathcal{E}n})$ greedy choices either all users have been satisfied or the algorithm has restarted the sampling part of the algorithm. Furthermore, it takes $\mathcal{O}(n \log n)$ rounds in expectation and with high probability for the

greedy part to recommend some item to every user, therefore the number of calls to the greedy subroutine is

$$\mathcal{O}(n \log n) \cdot \mathcal{O}(\sqrt{\mathcal{E}n}) \in \mathcal{O}(n\sqrt{\mathcal{E}n} \log n)$$

in expectation and with high probability. □

Theorem 3.3. *Algorithm 3.1 is $\mathcal{O}(\sqrt{n} \log^2 n)$-quasi competitive.*

Proof. In the beginning of the execution, the popularity of the most popular item is at most $n = 2^{\log n}$, i.e., the largest index of a non-empty category is at most $\log n$. By observing that the popularity of an item never increases and by applying Lemma 3.6, we get that the largest index of a non-empty category decreases by at least one in every phase with high probability. Therefore, after $\mathcal{O}(\log n)$ phases the largest index of a non-empty category is 0 with high probability and thus, there cannot be any more unsatisfied users.

As the cost of each phase is $\mathcal{O}(n\sqrt{\mathcal{E}n} \log n)$ by Lemma 3.8, in expectation the whole cost is

$$\mathcal{O}(\log n) \cdot \mathcal{O}(n\sqrt{n\mathcal{E}} \log n) \in \mathcal{O}(n\sqrt{n\mathcal{E}} \log^2 n) \ .$$

Since the cost of the quasi-offline algorithm is $n\mathcal{E}$, the quasi competitive ratio of Algorithm 3.1 is

$$\frac{\mathcal{O}(n\sqrt{n\mathcal{E}} \log^2 n)}{n\mathcal{E}} \in \mathcal{O}\left(\frac{\sqrt{n} \log^2 n}{\sqrt{\mathcal{E}}}\right) \ .$$

Specifically, when $\mathcal{E}$ is a constant, Algorithm 3.1 is $\mathcal{O}(\sqrt{n} \log^2 n)$-quasi competitive.

□

4

The Anonymous Quasi-Offline Algorithm

In the previous chapter, we established an online recommendation algorithm that solves the ignorant recommendation problem. We analyzed the performance of this algorithm against a quasi-offline algorithm that knows the distribution according to which the preferences of the users are chosen. We showed that the quasi-competitive ratio of the online algorithm is optimal up to some polylogarithmic factors.

However, the quasi-offline algorithm we considered was rather weak and in particular, it could not make any use of the fact that it is technically allowed to change its strategy during the execution. Our next goal is to study a considerably stronger model, where the quasi-offline algorithm is given more information about the input prior to its execution. In other words, we consider the anonymous model, where the quasi-offline

algorithm is shown the whole input, but the identities of the users are hidden. We strengthen the model for the quasi-offline algorithm further by allowing it to choose the input sequence according to which the users are selected. Despite these seemingly strong improvements, we show that asymptotically, the quasi-competitive ratio of our online algorithm stays the same even when compared to the stronger quasi-offline algorithm.

4.1 Model

Now, we introduce the anonymous recommendation problem. The model description is very similar to the ignorant case, but for the sake of completeness, we dedicate this section to formally and thoroughly describing the model. The main differences are in the (anonymized) input and in the complexity measure.

The input is a pair (U, V) consisting of a set of users $U = \{u_1, \ldots, u_n\}$ and a set of preference vectors $V = \{v_1, \ldots, v_n\}$ of length m, where each preference vector $v \in V$ corresponds to the (binary) preferences of some user $u \in U$ on m items. Each user u_i is assigned exactly one preference vector v_j according to a hidden bijective mapping $\pi : U \to V$. By identifying the users with the preference vectors, π is a permutation of the users. The permutation π is chosen uniformly at random from the set of all possible permutations.

The execution of an anonymous recommendation algorithm A can be divided into discrete rounds. In each round, a recommendation algorithm first picks a user $u \in U$ and then recommends some item b to this user. Recommending item b to user u is equivalent to checking whether user u likes b or not, i.e., the corresponding entry is revealed to the algorithm immediately after the recommendation. Algorithm A is allowed to pick the user and the item at random.

We say that user u is *satisfied* after she has been recommended an item that she likes. The goal is to satisfy all users which corresponds to finding at least one 1-entry from each preference vector. The algorithm terminates when all users are satisfied. The runtime of a recommendation algorithm is measured as the expected number of queries. In other words, the runtime corresponds to the number of rounds until all users are satisfied. Therefore, the trivial upper and lower bound for the runtime

are $n \cdot m$ and n, respectively, since it takes $n \cdot m$ queries to learn every element of every preference vector and n queries to learn one entry from each preference vector.

We assume that for any user u, there is always at least one item that she likes but we do not make any further assumptions on the input. Let OPT be the optimal recommendation algorithm for the anonymous recommendation problem. We measure the quality of a recommendation algorithm A by its approximation ratio, i.e., the maximum ratio between the expected number of queries by A and by OPT for any input I.

4.2 Anonymous Recommendations

The main goal of this chapter is to show that from an asymptotic perspective, the anonymous and the static recommendation problems are equally hard. Recall that in the static setting, the algorithm sees the probability distribution according to which the preferences are chosen and fixes an ordering O of the items before the first query. Then, each user is recommended items according to O until she is satisfied. Otherwise, the static model is similar to the anonymous model.

Note that to compare the static and anonymous model, we can consider I an instance of preferences picked from D, where D is the probability distribution given to the static algorithm. We first observe that solving the static recommendation problem takes at least as much time as solving the anonymous recommendation problem for any input $I = (U, V)$. Clearly, an anonymous algorithm that chooses the best fixed ordering of books is at least as fast as any static algorithm for this instance.

We show that the greedy algorithm for the static recommendation problem is a 24-approximation to the anonymous recommendation problem. We follow the general ideas of the analysis of the greedy algorithm for **mssc** by Feige et al. [37], where they show that the greedy algorithm provides a 4-approximation. The fundamental difference between our analysis and theirs comes from bounding from below the number of recommendations needed to satisfy a given set $U' \subseteq U$ of users. In the static setting, it is easy to get a lower bound on the number of recommendations needed per user. Given the most popular item b^* among users in U', $\Omega(|U'|^2/|b^*|)$ recommendations are needed, since one item can satisfy

at most $|b^*|$ users, and each item is recommended to all unsatisfied users. In the adaptive setting, this is not necessarily the case.

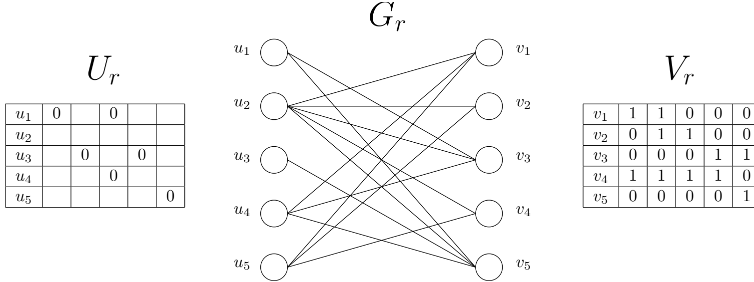
We first give a lower bound on the amount of queries that is needed to satisfy any group of users as a function of the most popular item within this group. In essence, we show that from an asymptotic perspective, any adaptive algorithm also needs $\Omega(|U'|^2/|b^*|)$ rounds to satisfy all users in U'. Then, in Section 4.4, we show how to utilize our lower bound and show that the greedy algorithm for **mssc** yields a constant approximation to the anonymous recommendation problem.

4.2.1 The Consistency Graph

We identify the users with their preference vectors, which indicates that each user $u \in U$ corresponds to an (initially) unknown binary preference vector of length m. Therefore, each user can be seen as a preference vector that denotes the information we have gained about user u. We also identify the items with their corresponding indices, i.e., for an item b that has been recommended to u, $u(b)$ denotes the entry in the preference vector of user u that corresponds to whether u likes b or not. We call $u \in U$ and $v \in V$ *consistent*, if $u(i) = v(i)$ for every revealed entry $u(i)$.

Let OPT be the optimal anonymous recommendation algorithm. We model the state of an execution of OPT as an undirected bipartite graph $G = (U \cup V, E)$, where $(u, v) \in E$ iff $u \in U$ and $v \in V$ are consistent. We refer to G as the *consistency* graph. The purpose of the consistency graph is to model the uncertainty that OPT has on the preferences of the users. To simplify our analysis, we provide OPT with the following advantage. Whenever OPT recommends user u an item b that u likes, we reveal the connection between $u \in U$ and $v \in V$ in permutation π, i.e., that $\pi(u) = v$. We note that this advantage can only improve the runtime of OPT, i.e., if we prove a lower bound for the performance of this “stronger” version of OPT, the same bound immediately holds for the optimal anonymous recommendation algorithm.

Now since the connection is revealed after finding a 1-entry and thus, the complete preference vector of u is learned, nothing further can be learned by recommending u more items. Therefore, we can simply ignore $u \in U$ and $\pi(u) \in V$ for the rest of the execution. Thus, upon recom-



u_1	0		0		
u_2					
u_3		0		0	
u_4			0		
u_5					0

v_1	1	1	0	0	0
v_2	0	1	1	0	0
v_3	0	0	0	1	1
v_4	1	1	1	1	0
v_5	0	0	0	0	1

Figure 4.1: A matrix representation of the unknown and the known entries of an input are given on the left and right, respectively. The consistency graph constructed based on V_r and the state of U_r is denoted by G_r. Nodes u_i and v_j are connected if and only if the corresponding rows are consistent.

mending user u an item that she likes, we simply remove u from U and the corresponding preference vector $\pi(u)$ from V. The modification also implies that the termination condition, i.e., all users being satisfied, is equivalent to the sets U and V becoming empty.

The construction of G is illustrated in Figure 4.1. We emphasize that the graph G changes over time and we denote the state of G in round $r \geq 0$ by $G_r = (U_r \cup V_r, E_r)$, where U_r and V_r are the users and their preference vectors remaining in round r, respectively, and E_r is the set of edges between consistent nodes in round r. Notice that $G_0 = (U_0 \cup V_0, E_0)$ is a complete bipartite graph. We omit the index from the consistency graph whenever the actual round number is irrelevant. In addition, we note that even if the identity of a certain user u is clear (see user u_3 in Figure 4.1 for an example), the edges connected to u and $\pi(u)$ are only removed from G when the corresponding users and preference vectors become inconsistent.

4.3 Learning the Preferences

The goal of this section is to quantify the amount of knowledge OPT can gain per round. The intuition behind modeling the anonymous recom-

mendation problem as a bipartite graph is that the number of remaining edges correlates with the amount of uncertainty OPT has. In other words, by querying the preferences of the users, OPT can exclude inconsistent edges in G. When a 0-entry is discovered by recommending item b to user u, at most $|b|$ (recall Definition 3.1) preference vectors can become inconsistent with u since there are $|b|$ vectors $v \in V$ such that $v(b) = 1$. On the other hand, when discovering a 1-entry, up to $2|U|$ edges might get removed due to removing u, $\pi(u)$ and all edges adjacent to them from G. Note that the consistency graph is simply a representation of the knowledge OPT has about the preference vectors, i.e., excluding the edges from the consistency graph only happens implicitly according to the revealed entries.

We employ an amortized scheme, where we pay in advance for edges that get removed by discovering 1-entries. Consider the case where a 0-entry is revealed from $u(b)$. Now all the edges $(u, v) \in E$, where $u(b) \neq v(b)$, are removed. For every edge (u, v) removed this way, we give both node u and node v two units of money that can be used later when their corresponding connections are revealed. Since at most $|b|$ edges are removed, we pay at most $2|b| + 2|b| = 4|b|$ units of money in total in a round where OPT discovers a 0-entry.

As an example, consider the graph illustrated in Figure 4.1 and assume that a 0 is revealed from $u_4(5)$. Now u_4 becomes inconsistent with v_3 and v_5, and the corresponding edges are removed. Upon removing these edges, we pay two units of money to v_3, two units of money to v_5 and four units of money to u_4.

4.3.1 Finding a 1-entry

Now we look at the case of discovering a 1-entry. In the following, we consider the consistency graph G_r for an arbitrary round r but omit the index when it is irrelevant for the proofs. Consider user $u \in U$ and let $\pi(u) = v$. Upon discovering the 1-entry from user u, we reveal that $\pi(u) = v$ and all the edges adjacent to u and v are removed. We divide the analysis into two cases. Consider first the case where the sum of degrees $d = |\Gamma(u)| + |\Gamma(v)|$ is at most $4|U|/3$, where $\Gamma(u)$ denotes the exclusive neighborhood of u, i.e., $\Gamma(u) = \{v \in V \mid (u, v) \in E\}$ and analogously for

$v \in V$.

Note that since satisfied users are removed from U, $G = (U \cup V, E)$ is a complete bipartite graph if there are no revealed 0-entries. Therefore, any edge $(u, v) \notin E$, where $u \in U = U_r, v \in V = V_r$, was removed by revealing a 0-entry. Given that $d \leq 4|U|/3$, we know that at least $2|U| - 4|U|/3$ of the edges adjacent to u or v have been removed by revealing 0-entries. We pay two units of money to either u and v for every edge removed from the set of edges adjacent to nodes in $\Gamma(u) \cup \Gamma(v)$ and therefore, the combined money that the nodes have is at least $2(2|U| - 4|U|/3) = 4|U|/3$. Therefore, the money "compensates" for all edges that are removed due to revealing the connection between u and v.

We use the rest of this section to study the second case, that considers the case where the sum of degrees of nodes u and v is high, i.e., larger than $4|U|/3$. The aim is to show that it is unlikely that v is the preference vector of u, since there are many consistent nodes with u and v and thus, there have to be considerably more valid permutations π', where $\pi'(u) \neq v$, than permutations, where $\pi'(u) = v$. This in turn implies that a randomly chosen permutation is likely not to have u connected to v.

We call a matching σ *compatible* with an edge $e = (u, v)$ if $(u, v) \in \sigma$ and *incompatible* with e otherwise. In the following lemma, we bound the number of perfect matchings that are compatible with a given edge e in G. Note that every perfect matching in G corresponds to some permutation of the users.

Lemma 4.1. *Assume that the sum of degrees of nodes $u \in U$ and $v \in V$ is larger than $4|U|/3$ in G. Let h be the total number of perfect matchings in G. Then there are at most $3h/|U|$ perfect matchings that are compatible with (u, v).*

Proof. Let σ be a perfect matching that is compatible with (u, v) in G. Let

$$U' = \{u' \in \Gamma(v) \mid \exists v' \in \Gamma(u) \text{ such that } (u', v') \in \sigma\} \setminus \{u\} ,$$

$k = |\Gamma(u)|$ and let

$$\Gamma_\sigma(\Gamma(u)) = \{u' \in U \mid \exists v' \in \Gamma(u) \text{ such that } (u', v') \in \sigma\}$$

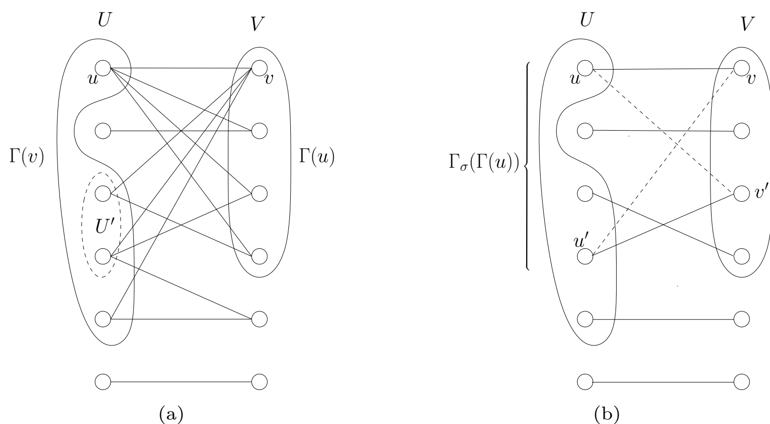


Figure 4.2: The consistency graph is illustrated on the left. On the right, we show a perfect matching compatible with (u, v) with the solid lines. For every node $u' \in U'$, we have a valid perfect matching that is incompatible with (u, v) if we use edges (u, v') and (u', v) instead of (u, v) and (u', v').

be the set of nodes matched to $\Gamma(u)$ by σ. See Figure 4.2 for an illustration. Since σ is a matching, we get that $k = |\Gamma(u)| = |\Gamma_\sigma(\Gamma(u))|$. Also, we know that $|\Gamma(v)| + |\Gamma(u)| > 4|U|/3$ and therefore $|\Gamma(v)| \geq 4|U|/3 - k + 1$.

By taking a closer look at the definition of U', we see that $U' = \Gamma(v) \cap \Gamma_\sigma(\Gamma(u)) \setminus \{u\}$ and by re-writing, we get that $U' = \Gamma(v) \setminus (U \setminus \Gamma_\sigma(\Gamma(u))) \setminus \{u\}$. From the equations above, it follows that

$$\begin{aligned}|U'| &= |\Gamma(v) \setminus (U \setminus \Gamma_\sigma(\Gamma(u)))| - 1 \\ &\geq |\Gamma(v)| - (|U| - k) - 1 \geq \frac{4|U|}{3} - k + 1 - (|U| - k) - 1 = \frac{|U|}{3} .\end{aligned}$$

For each node $u' \in U'$, we have a perfect matching $\sigma_{u'}$ that is incompatible with (u, v) in G, where (u, v) and $(u', v') \in \sigma$ are replaced by (u, v') and (u', v). In addition, the incompatible perfect matching $\sigma_{u'}$ is different for every $u' \in U'$, since $(u', v) \notin \sigma_z$ for any $u' \neq z \in U'$. Therefore, we have at least $|U|/3$ perfect matchings incompatible with (u, v) for every perfect matching that is compatible with (u, v). Note that no matchings are counted twice. The claim follows. □

In the beginning of the execution, the probability of user $u \in U$ to be matched to vector $v \in V$ is simply $1/n$. When revealing the unknown entries, these probabilities change. The next step is to bound the probability of user $u \in U$ to be matched to vector $v \in V$ given the state of the consistency graph G. We identify the randomly chosen permutation π with a perfect matching σ_π where $(u, v) \in \sigma_\pi$ iff $\pi(u) = v$. Since the permutation π was chosen uniformly at random, any valid permutation, i.e., a permutation that does not contradict the revealed entries, is equally likely to be σ_π. Therefore, the probability that an edge (u, v) is in matching σ_π corresponds to the ratio of perfect matchings in G that are compatible with (u, v) and the number of all perfect matchings in G. We denote the event that edge (u, v) is in σ_π by $A(u, v)$ and the probability of $A(u, v)$ given G by $\Pr[A(u, v) \mid G]$.

Lemma 4.2. *Let $G = (U \cup V, E)$ be the consistency graph. For any nodes $u \in U$ and $v \in \Gamma(u)$, such that $|\Gamma(u)| + |\Gamma(v)| > 4|U|/3$, $\Pr[A(u, v) \mid G] \leq \frac{6}{|\Gamma(u)| + |\Gamma(v)|}$.*

Proof. Since the permutation π is chosen uniformly at random, σ_π is equally likely to be any of the possible perfect matchings in G. Therefore, the likelihood of u being matched to $v \in V$ is the number of perfect matchings that are compatible with (u, v) divided by the number of all possible perfect matchings. By Lemma 4.1, the number of perfect matchings that are compatible with (u, v) is at most $3h/|U|$, where h is the total number of perfect matchings in G. Therefore,

$$\Pr[A(u,v) \mid G] \leq \frac{3h}{|U|} \cdot \frac{1}{h} = \frac{3}{|U|} \leq \frac{3}{\frac{1}{2}(|\Gamma(u)| + |\Gamma(v)|)} = \frac{6}{|\Gamma(u)| + |\Gamma(v)|} .$$

□

4.3.2 Progress

Now, we define the *progress* $c(u, b, r)$ for recommending item b to user u in round $r \geq 0$. Informally, the idea of the progress value is to employ the money paid during the execution to bound the expected number of edges removed per round.

Consider any round r and let $w_r(z)$ denote the *wealth* of node $z \in U_r \cup V_r$, where wealth refers to the amount of money z has in round r. In the case of revealing a 0-entry, the progress indicates the number of removed edges and the money that is paid to the nodes adjacent to the removed edges. When finding a 1-entry and revealing the connection between u and $\pi(u)$, the progress indicates the number of removed edges minus the money already paid to u and $\pi(u)$. Let

$$\Gamma_r(u) = \{v \in V_r \mid (u, v) \in E_r\}$$

and let $\Gamma_r^b(u)$ denote the neighbors of u that like item b, i.e.,

$$\Gamma_r^b(u) = \{v \in V_r \mid (u, v) \in E_r \wedge v(b) = 1\} .$$

Then, for entry $u(b)$ revealed in round r, the progress is given by

$$c(u,b,r) = \begin{cases} \sum_{v \in \Gamma_r^b(u)} 5 & \text{if } u(b) = 0 \\ |\Gamma_r(u)| + |\Gamma_r(\pi(u))| - (w_r(u) + w_r(\pi(u))) & \text{if } u(b) = 1 . \end{cases}$$

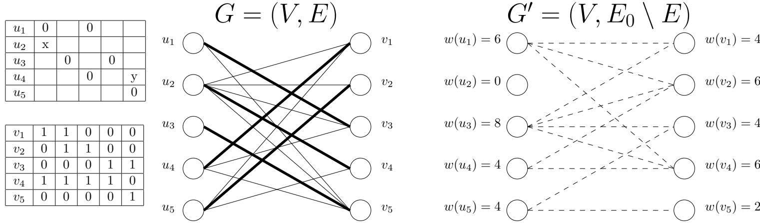


Figure 4.3: In graph G' the dashed lines indicate the edges removed from the consistency graph during the execution of an algorithm. The wealth of each node is illustrated next to the corresponding node. The bold lines denote the underlying permutation π that connects users in U with their preference vectors in V.

An illustration of the wealth and progress concepts is given in Figure 4.3. In the example given in Figure 4.3, revealing entry $u_2(1) = 1$, denoted by x, has a progress value of $|\Gamma(u_2)| + |\Gamma(v_4)| - (w(u_2) + w(v_4)) = 5 + 2 - 0 - 6 = 1$ and revealing entry $u_4(5) = 0$, denoted by y, yields a progress of $\sum_{\{v_3,v_5\}} 5 = 10$.

Next, we show that the total progress value counted from the first round up to any round r is never smaller than the number of edges removed from G within the first r rounds. We denote an execution of an algorithm until round r by $\mathcal{X}_r = (u^1, b^1), (u^2, b^2), \dots, (u^{r-1}, b^{r-1})$, where u^i corresponds to the user selected in round $i < r$ and similarly for item b^i.

Lemma 4.3. *For any round r and execution $\mathcal{X}_r$, it holds that*

$$\sum_{i=1}^{r-1} c(u^i, b^i, i) \geq |E_0| - |E_r| \ .$$

Proof. Let I_0 denote the set of indices $i < r$ such that $u^i(b^i) = 0$ and I_1 analogously for indices such that $u^i(b^i) = 1$. Let W_r denote the amount of money paid until round r, i.e.,

$$W_r = \sum_{i \in I_0} \sum_{v \in \Gamma_i^{b^i}(u^i)} 4 \ .$$

Now, we can write

$$\sum_{i \in I_0} c(u^i, b^i, i) = W_r + \sum_{i \in I_0} \sum_{v \in \Gamma_i^{b^i}(u^i)} 1 = W_r + \sum_{i \in I_0} |\Gamma_i^{b^i}(u^i)| \ ,$$

where $|\Gamma_i^{b^i}(u^i)|$ equals to the number of edges removed from the consistency graph in round i, i.e., $|\Gamma_i^{b^i}(u^i)| = |E_i| - |E_{i+1}|$.

Furthermore, we can bound the progress in rounds in I_1 by

$$\begin{aligned}\sum_{i \in I_1} c(u^i, b^i, i) &= \sum_{i \in I_1} |\Gamma_i(u^i)| + |\Gamma_i(\pi(u^i))| - (w_i(u^i) + w_i(\pi(u^i))) \\ &\geq \left(\sum_{i \in I_1} |\Gamma_i(u^i)| + |\Gamma_i(\pi(u^i)| \right) - W_r \ ,\end{aligned}$$

where $|\Gamma_i(u^i)| + |\Gamma_i(\pi(u^i))| = |E_i| - |E_{i+1}|$. Combining the sums from above we get that

$$\begin{aligned}\sum_{i \in I_0 \cup I_1} c(u^i, b^i, i) &\geq W_r + \sum_{i \in I_0} |\Gamma_i^{b^i}(u)| \\ &\quad + \left(\sum_{i \in I_1} |\Gamma_i \left(u^i\right)| + |\Gamma_i(\pi \left(u^i\right)|\right) - W_r \\ &= \sum_{i \in I_0} |\Gamma_i^{b^i}(u)| + \left(\sum_{i \in I_1} |\Gamma_i \left(u^i\right)| + |\Gamma_i(\pi \left(u^i\right)| \right) \\ &= \sum_{i=0}^{r-1} |E_i| - |E_{i+1}| = |E_0| - |E_r| \ .\end{aligned}$$

□

The last thing we need to show is an upper bound on the expected progress per round for any round r. For ease of notation, we omit the round index for the rest of the section. In addition, we identify $c(u, b)$ with a random variable that equals to the progress gained by revealing entry (u, b).

Lemma 4.4. *Let b^* be the most popular item among the set U of unsatisfied users. Then for any user $u \in U$ and item b, $\mathbb{E}[c(u,b)] \leq 6|b^*|$.*

Proof. Consider any user $u \in U$ and any item b. We partition the event space into three disjoint parts according to the outcome of querying user u for item b and show that for each part, $\mathbb{E}[c(u,b)] \leq 6|b^*|$. First, consider the case where a 0-entry is revealed. By definition, we get $\mathbb{E}[c(u,b) \mid u(b) = 0] \leq 5|b^*|$. Furthermore,

$$\mathbb{E}\big[c(u,b) \mid (u(b) = 1) \wedge \big(|\Gamma(u)| + |\Gamma(\pi(u))| \leq 4|U|/3\big)\big] \leq 0 ,$$

since $w(u) + w(\pi(u)) \geq 4|U|/3$ given that $|\Gamma(u)| + |\Gamma(\pi(u))| \leq 4|U|/3$.

Let us then consider the third part of the event space, where $u(b) = 1$ and $|\Gamma(u)| + |\Gamma(\pi(u))| > 4|U|/3$ and let us denote this event by B. Let

$$E' = \{(u,v) \in E \mid |\Gamma(u)| + |\Gamma(\pi(u))| > 4|U|/3)]\}$$

and $\hat{\Gamma}(u) = \{v \in \Gamma(u) \mid (u,v) \in E' \wedge v(b) = u(b) = 1\}$. Then, by Lemma 4.2,

$$\begin{aligned}\mathbb{E}[c(u,b) \mid B] &\leq \sum_{v \in \hat{\Gamma}(u)} (|\Gamma(u)| + |\Gamma(v)|) \cdot \Pr[A(u,v) \mid G] \\ &\leq \sum_{v \in \hat{\Gamma}(u)} (|\Gamma(u)| + |\Gamma(v)|) \frac{6}{|\Gamma(u)| + |\Gamma(v)|} = \sum_{v \in \hat{\Gamma}(u)} 6 \leq 6|b^*| ,\end{aligned}$$

where $|b^*| \geq |b| \geq |\hat{\Gamma}(u)|$, since b^* is the most popular item. Since the three aforementioned parts span the whole probability space, $\mathbb{E}[c(u,b)]$ is bounded by the maximum of the three cases and thus, the claim follows. □

By combining Lemma 4.4 and Lemma 4.3, we get the following theorem.

Theorem 4.1. *Let $R \subseteq U_0$ be a set of users and b^* the most popular item among these users. Any algorithm requires at least $|R|^2/(6|b^*|)$ queries to users in R in expectation to satisfy all users in R.*

Proof. Consider only users in R and let $v_1, \ldots, v_{|R|} = V_R \subseteq V$ be the corresponding preference vectors. Since each user $u \in R$ initially has $|R|$ edges connected to users in R, there are $|R|^2$ edges in total.

Recall that satisfied users are removed from the consistency graph. Thus, when all users in R are satisfied, the set of edges in the consistency graph is empty. By Lemma 4.4 and by linearity of expectation at least $|R|^2/6|b^*|$ queries are needed before the progress value is greater than $|R|^2$ in expectation. By Lemma 4.3, the progress value gives an upper bound on the number of edges removed. Therefore, the number of queries needed before all users are satisfied is at least $\frac{|R|^2}{6|b^*|}$. □

4.4 The Greedy mssc Algorithm

The goal of this section is to show that the greedy algorithm for the **mssc** problem provides an $\mathcal{O}(1)$-approximation for the anonymous recommendation problem. In particular, the goal is to establish the following theorem.

Theorem 4.2. *The greedy* **mssc** *algorithm provides a* 24*-approximation for the anonymous recommendation problem.*

```
while |U| > 0 do
  Choose the most popular item b among users in U.
  for all u ∈ U do
    Recommend b to u.
  end for
end while
```

Algorithm 4.1: Greedy **mssc** algorithm

Our proof follows the steps of the 4-approximation proof by Feige et al. [37] The crucial difference between their proof and ours is that in the static case of **mssc**, where each user is recommended items according to the same fixed order, it is clear that every algorithm requires $\Omega(|R|^2/|b|)$ rounds to to satisfy all users in $R \subseteq U_0$ given that b is the most popular item among users in U. In our case, we use Theorem 4.1 to provide a

similar observation in terms of expectation. For the sake of completeness, we dedicate this section to give a detailed proof that the existing tools used to prove the 4-approximation of the greedy **mssc** algorithm can be used for our purposes with a few modifications. The pseudo-code for the greedy algorithm is given in Algorithm 4.1.

We refer to the iterations of the while loop of the greedy algorithm as *steps* and label them by the positive integers. Note that each step of Algorithm 4.1 consists of $|U|$ many rounds according to our model, where U is the set of unsatisfied users in the beginning of the step. Let X_i be the set of users satisfied in step i and let R_i be the set of unsatisfied users prior to step i. The cost of the greedy algorithm is given by $\sum_i i|X_i| = \sum_i |R_i|$.

We define the price of user $u \in X_i$ to be $p_u = |R_i|/|X_i|$. Then we set

$$\textbf{price} = \sum_{u \in U} p_u = \sum_i \sum_{u \in X_i} p_u = \sum_i |X_i| \frac{|R_i|}{|X_i|} = \sum_i |R_i| \ ,$$

which shows that **price** is equal to the cost of the greedy algorithm.

We model the solutions for both greedy algorithm and an optimal algorithm OPT for the anonymous recommendation problem by the following diagrams. Consider first the greedy algorithm. There are $|U_0|$ columns, one for each user in the input. The users are ordered from left to right by the order in which they are satisfied by the greedy algorithm. The height of each column is the price p_u for the corresponding user u. The area under the histogram equals therefore to **price**.

Similarly, we have a diagram for OPT. Again, there is a column for every user. In the case of OPT however, the height of each column corresponds to the expected number of queries made to the corresponding user. Again, the area of the diagram of the optimal algorithm is equal to the expected cost of OPT. The columns are ordered in an ascending order by the height of each column. The diagrams are illustrated in Figure 4.4.

As the next step, we show that the area of the greedy diagram is at most 24 times the size of the optimal diagram. To show this, we shrink the diagram of the greedy algorithm by shrinking the height of each column by a factor of 12 and the width by 2. Then, we align the shrunk version of the diagram to the right of the other diagram corresponding to the solution of OPT. In other words, the diagram of the greedy algorithm

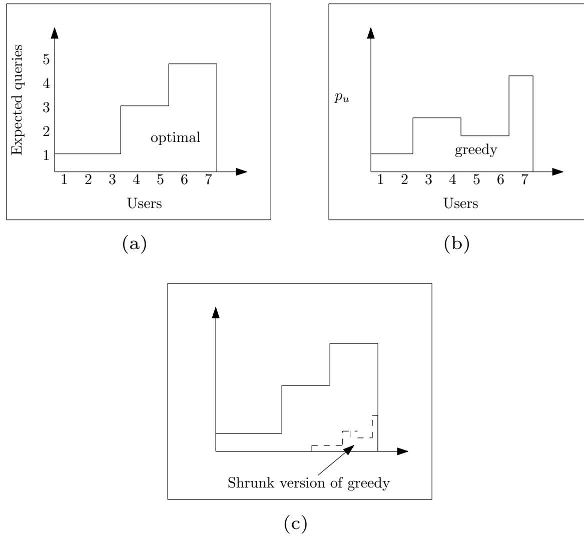


Figure 4.4: The diagram corresponding to the solution of the optimal algorithm is given on the left. The height of each column in the diagram of the optimal solution corresponds to the expected number of queries to the corresponding user. The diagram of the greedy algorithm is shown on the right. In this diagram, the height of each column is the price of the corresponding user. The shrinking and aligning of the diagram of the greedy algorithm into the diagram of the optimal algorithm is illustrated in Figure 4.4(c).

now occupies the space of columns of OPT from $|U_0|/2 + 1$ up to $|U_0|$ (where $|U_0|$ is assumed w.l.o.g. to be even).

Consider any point q' in the diagram of the greedy algorithm. Let u be the user that corresponds to this column and let i be the step when this user is satisfied. Thus, the height of point q' is at most $|R_i|/|X_i|$ and the distance to the right-hand boundary is at most $|R_i|$. The shrinking maps q' to another point q, where the height h of q satisfies $h \leq |R_i|/(12|X_i|)$ and the distance to the right boundary r satisfies $r \leq |R_i|/2$.

We now show that q lies within the histogram of OPT. To prove this, we need to show that there is at least one user in the first $|R_i|/2$ users who is queried at least $|R_i|/(12|X_i|)$ times. Consider now only the first $|R_i|/2$ users in R_i and denote these users by R_i^f. By Theorem 4.1, it takes at least

$$\frac{|R_i^f|^2}{6|X_i|} = \frac{\left(\frac{|R_i|}{2}\right)^2}{6|X_i|} = \frac{|R_i|^2}{24|X_i|}$$

rounds in expectation to satisfy all of these users. Therefore, there is at least one user $u \in R_i^f$ that is queried at least

$$\frac{\frac{|R_i|^2}{24|X_i|}}{|R_i^f|} = \frac{\frac{|R_i|^2}{24|X_i|}}{\frac{|R_i|}{2}} = \frac{|R_i|}{12|X_i|}$$

times in expectation. Since the users are ordered according to the number of queries, every user $v \in R_i \setminus R_i^f$ is queried at least $|R_i|/(12|X_i|)$ times. Thus, q indeed lies within the histogram of the optimal algorithm, yielding Theorem 4.2.

5

Conclusion

In the first part of this thesis, we studied three different kind of recommendation problems. First, we studied an ignorant recommendation problem, where the input is an unknown binary preference matrix and the task is to find at least one 1-entry in each row by querying the entries in the matrix. We observed that there are inputs for the ignorant recommendation problem where any algorithm performs badly and therefore, we analyzed it in a competitive manner. However, it turns out that an offline algorithm that has access to the whole input matrix always has a cost of exactly n regardless of the input, which results in a trivial competitive ratio of $\Omega(m)$ for any algorithm. Therefore, we introduced the concept of quasi-competitiveness, where the online algorithm is compared to a quasi-offline algorithm that has a restricted access to the input matrix.

To find an appropriate quasi-offline algorithm, we turned our attention to the second variant of the recommendation problem, i.e., the static

recommendation problem. In the static recommendation problem the algorithm solving the problem is given access to the probability distribution of the preference vectors of the users. We observed that in this model, the best that the quasi-offline algorithm can do is to compute an ordered list of items that minimizes the expected number of items the algorithm has to recommend per user, given that each user is recommended books according to this list. Computing this list of items is equivalent to solving the **mssc** problem. It is well known that a greedy algorithm for **mssc** is a 4-approximation and therefore, we compared our solution to the greedy algorithm instead of the optimal.

We introduced an algorithm that achieves a cost of $\mathcal{O}(n\sqrt{n\mathcal{E}}\log^2 n)$, where $\mathcal{E}$ is the expected cost for the greedy **mssc** algorithm to cover a single element. Since the total cost of the greedy **mssc** algorithm is $n\mathcal{E}$, the quasi competitive ratio of our algorithm is

$$\mathcal{O}\left(\frac{\sqrt{n}\log^2 n}{\sqrt{\mathcal{E}}}\right) .$$

Therefore, the worst case is obtained when $\mathcal{E}$ is a constant and it follows that our algorithm is $\mathcal{O}(\sqrt{n}\log^2 n)$-quasi-competitive.

Finally, we extended this result by studying a third recommendation model, i.e., the anonymous recommendation model. In the anonymous model, we reveal the input matrix to the algorithm, but we hide the identities of the users. This is performed by shuffling the labels that identify the users according to an arbitrary permutation. Furthermore, in the anonymous model, we allowed the algorithm to decide the sequence in which the users are presented.

We showed that the quasi-competitive ratio of our online algorithm stays asymptotically the same even when we compare our ignorant algorithm to the stronger quasi-offline algorithm. We achieved this by showing that the greedy algorithm for the **mssc** problem is a $\mathcal{O}(1)$-approximation algorithm for the anonymous recommendation problem.

Part II

Adaptivity in the Stone Age Model

6

Fault Tolerance

The biological form of close-range (juxtacrine) message passing relies on designated messenger molecules that bind to crossmembrane receptors in neighboring cells and trigger a signaling cascade that eventually affects gene expression, thus modifying the neighboring cells' states. This mechanism should feel familiar as it resembles the message passing schemes of distributed digital systems. In contrast to nodes in distributed digital systems, however, biological cells are not believed to be Turing complete, rather each biological cell is pretty limited in computation as well as communication. In attempt to cope with these differences, Emek and Wattenhofer [36] introduced the *Stone Age* model of distributed computing (a.k.a. networked finite state machines), where each node in the network is a very weak computational unit with limited communication capabilities and showed that several fundamental distributed computing problems can be solved efficiently under this model.

An important topic left outside the scope of [36] is that of node *failures*. Just like distributed digital systems, biological systems may suffer from local failures and the ability of the system to recover from these failures is crucial to its survival. However, the desired failure recovery features in biological cellular networks typically differ from those traditionally studied in the distributed computing literature. In particular, a major issue in the context of biological cellular networks, that is rarely addressed in the study of distributed digital systems, is that of *confining* the failures: while nodes in the vicinity of the failures are doomed to be affected by them (hopefully, to a bounded extent), isolating the nodes sufficiently far away from any failure so that their operation remains unaffected is often a critical matter.

An example for the essential role that this confinement plays in biological systems is found in the transition of tumors from a benign state to a malignant one. Cancerous tumors require an extensive creation of new blood vessels to obtain nutrients and oxygen and to evacuate metabolic wastes and carbon dioxide. These blood vessels are created through an angiogenesis process from healthy endothelial cells by signaling a normally quiescent vasculature to continually sprout new vessels [52]. Indeed, many forms of cancer treatment are based on blocking these signals, thus isolating the cancerous (faulty) cells from the healthy (correct) ones, eventually causing the tumor to suffocate and die.

The goal of this chapter is to take a step towards bringing the models from computer science closer to biology by extending the Stone Age model to accommodate node failures and introducing a new method for measuring the failure recovery performance of networks that takes into account the aforementioned confinement property. This new method first measures the number of "computationally-meaningful" steps made by the individual nodes, which are essentially all steps in which the node participates (in the weakest possible sense) in the global computational process. Then, an algorithm is said to be *effectively confining* if (1) the runtime of the nodes that are not adjacent to any failed node is $\mathcal{O}(\log^{\mathcal{O}(1)} n)$; and (2) the global runtime (including all correct nodes) is $\mathcal{O}((C+1)\log^{\mathcal{O}(1)} n)$, where C is the number of crash failures. In other words, the total runtime is $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ when amortized over the number of failures.

Following that, we turn our attention to the extensively studied *max-*

imal independent set (MIS) problem and design a randomized effectively confining algorithm for it under the Stone Age model, extended to accommodate failures. This is achieved by carefully augmenting the MIS algorithm introduced in [36] with a new 'failure handling' component. Being a first step in the study of failure recovery under the Stone Age model, our algorithm assumes a *synchronous* environment, where the network is subject to *crash* failures only.[1] Nevertheless, the former assumption is justified by the findings of Fisher et al. [40] that model cellular networks as being subject to a *bounded asynchrony* scheduler, which is equivalent to a synchronous environment from an algorithmic perspective.

The chapter is organized as follows. An extension of the Stone Age model of [36] to node failures is presented in Section 6.2 together with our new method for evaluating the failure recovery performance of distributed algorithms. In Section 6.3, we analyze the runtime of the MIS protocol introduced in [36] under our crash failure model and show that each node not affected by a crash failure will reach an output state in $\mathcal{O}(\log^2 n)$ rounds and the corresponding output configuration is correct given that there are no failures. Then, in Section 6.4, we show how to extend this protocol to fix any incorrect output configurations induced by the crash failures. We contrast the runtime of $\mathcal{O}((C+1)\log^2 n)$ of the fault-tolerant MIS protocol by showing that the global runtime is $\Omega(C)$ for any algorithm.

6.1 Related Work

Recently, scientists in biology and computing have been flirting with each other. Distributed computing in particular seems to be a valuable tool towards understanding biological phenomena, as both often deal with networks of simple nodes, collaborating by means of minimal communication. Please see the recent survey from Navlahka and Bar-Joseph for more details [75].

[1] Notice that the analogy between cancerous cells and faulty nodes in the aforementioned example requires Byzantine rather than crash failures. In that sense, it falls outside the direct scope of this paper and serves only as an example for the importance of confining failures in biological systems.

In distributed computing, the standard model for a network of communicating devices is the message passing model [68, 78]. There are several variants of this model, where the power of the network has been weakened. Perhaps the best-known variant of the message passing model is the congest model, where the message size is limited to logarithmic size with respect to the size of the input graph [78]. A step to weaken the model further is to consider interference of messages, i.e., a node only hears a message if it receives a single message per round. An example of such a model is the radio network model [24]. In the beeping model [26, 41], the communication capabilities are reduced even further by only allowing to send beeps that do not carry any information, where a listening node cannot distinguish between a single beep and multiple beeps transmitted by its neighbors.

The models mentioned above focus on limiting the communication but not the computation, i.e., the nodes are assumed to be sufficiently strong to perform unlimited (local) Turing computations in each round. Networks of nodes weaker than Turing machines have been considered for example in the context of cellular automata [44, 83, 85]. The drawback of the usual approach to the networks of cellular automata is that they only consider highly regular graph structures, such as the grid. Motivated by this drawback, the Stone Age model of [36] accommodates arbitrary network topologies.

There also exists a lot of work on dynamic network models. A classic result from Awerbuch and Sipser states that under the message passing model, any algorithm designed to run in a static network can be transformed into an algorithm that runs in dynamic networks with only a constant multiplicative runtime overhead [14]. However, the transformation of Awerbuch and Sipser requires storing the whole execution history and sending it around the network, which is not possible under the Stone Age model.

In the context of the dynamic networks literature, it is often assumed that the dynamic changes are somehow restricted and behave "nicely". A typical assumption is that the changes only occur up to some point in time [11, 84]. Another common assumption is that the dynamic updates are spaced in time so that the system has an opportunity to recover before another change appears [58, 59, 72].

We consider adversarial network updates that can occur at any point in time, however, we restrict ourselves to only node removals. Protocols that work with such continuous dynamic updates have been considered, for example, in the context of peer-to-peer overlay networks [54, 62, 66].

Our work focuses on the MIS problem under the Stone Age model with crash failures. The MIS problem has a long history in the context of distributed algorithms [7, 18, 25, 68, 71, 77, 82]. Arguably the most significant breakthrough in the study of message passing MIS algorithms was the $\mathcal{O}(\log n)$ algorithm of Luby [71] (developed independently also by Alon et al. [7]).

For growth bounded graphs, it was shown by Schneider et. al. [79] that MIS can be computed in $\mathcal{O}(\log^* n)$ time. In the radio networks realm, it is known that with F channels, an MIS can be computed in $\Theta(\log^2 n/F) + \tilde{\mathcal{O}}(\log n)$ time [27]. The MIS problem was extensively studied also under the beeping model [1, 2, 80]. Afek et al. [1] proved that if the nodes are provided with an upper bound on n or if they are given a common sense of time, then the MIS problem can be solved in $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ time. This was improved to $\mathcal{O}(\log n)$ by Scott et al. [80] assuming that the nodes are equipped with sender collision detectors.

On the lower bound side, the seminal work of Linial [68] provides a lower bound of $\Omega(\log^* n)$ for computing an MIS in the message passing model [68]. Kuhn et al. [61] established a stronger lower bound stating that it takes $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right) + \Omega\left(\frac{\log \Delta}{\log\log \Delta}\right)$ rounds to compute an MIS, where Δ is the maximum degree of the input graph. For uniform algorithms in radio networks and therefore, also for the beeping model, there exists a lower bound of $\Delta(\sqrt{n/\log n})$ communication rounds [1].

Containing faults within a small radius of the faulty node has been previously studied for example in the context of self-stabilization [46]. It is known that the MIS problem can be solved with an elegant algorithm if the wake-up times of the nodes are controlled by a so-called centralized daemon [67, 81], who wakes up the nodes one at a time. In our model, the nodes are controlled by a distributed daemon that wakes up every node simultaneously. In the field of self-stabilization, the performance of a protocol is typically measured as a function of some network parameter, such as the cardinality of the network or the maximum degree, whereas

we allow the performance to depend on the number of failures.

The works perhaps closest to ours are by Kutten and Peleg, where they introduce the concepts of *mending algorithms* and *tight fault locality* [63, 64]. The idea behind a tight fault local mending algorithm is to be able to recover a legal state of the network after a fault occurs and, similarly to us, they measure the performance in terms of number of faults C. The term tight fault locality reflects the property that an algorithm running in time $\mathcal{O}(T(n))$ without failures is able to mend the network in time $\mathcal{O}(T(C))$. Their algorithm recovers an MIS in time $\mathcal{O}(\log C)$, but they use techniques that require the nodes to count beyond constant numbers, which is not possible in the Stone Age model. Furthermore, they consider transient faults, whereas we consider changes in the network topology.

The optimization version of MIS, i.e., MaxIS, is a classic combinatorial optimization problem that is known to be NP-hard [45, 57] and hard to approximate [53].

6.2 Model

We describe the network as an undirected graph $G = (V, E)$, where V is the ground set of nodes and E the ground set of edges. The execution is synchronous, i.e., it can be divided into discrete rounds. In each round r, node v first receives messages sent in round $r - 1$, then performs local computations and finally, broadcasts a message to all neighbors.

The nodes $\{v_1, \ldots, v_n\}$ correspond to the computational units in the network and the edges represent bidirectional communication channels. Adopting the (static) model of [36], each node $v \in V$ runs a protocol depicted by the 7-tuple

$$\Pi = \langle Q, Q_I, Q_O, \Sigma, \sigma_0, b, \delta \rangle \ ,$$

where Q is a fixed set of *states* and $Q_I \subseteq Q$ is the set of *input states.* The input states are bijectively mapped to the possible input values of the given problem. In the beginning of the execution, each node resides in one of the input states. Furthermore, $Q_O \subseteq Q$ is the set of *output states.* The output states are bijectively mapped to the possible output values of the given problem. The system is said to be in an *output configuration*

if all non-crashed nodes reside in an output state. In addition, Σ is a fixed *communication alphabet*, $\sigma_0 \in \Sigma$ is the *initial letter* and $b \in \mathbb{Z}_{>0}$ is a *bounding parameter*, where

$$B = \{0, 1, \ldots, b-1, {}^{\geq}b\}$$

is a set of $b+1$ distinguishable symbols. Finally,

$$\delta : Q \times B^{|\Sigma|} \to 2^{Q \times (\Sigma \cup \{\varepsilon\})}$$

is the *transition function.*

The nodes communicate by transmitting messages that consist of a single letter $\sigma \in \Sigma$. Each neighbor v of node u has a port $\phi_u(v)$ in which v stores the last message received from u. Transmitting the designated empty symbol ε corresponds to the case where u does not transmit any message. In the beginning of the execution, all ports contain the initial letter σ_0.

The execution of any protocol proceeds in discrete rounds indexed by the positive integers. In each round r, node v is in some state $q \in Q$. Let $\sharp(\sigma)$ be the number of appearances of $\sigma \in \Sigma$ in v's ports in round r. Furthermore, let $(\beta_b(\sharp(\sigma)))_{\sigma\in\Sigma}$ be a vector that indicates the multiplicities of the query letters, where

$$\beta_b(x) = \begin{cases} x & \text{if } 0 \leq x \leq b-1 \\ {}^{\geq}b & \text{otherwise.} \end{cases}$$

Then the message σ' that v sends in round r and the state q' in which v resides in round $r+1$ are chosen uniformly at random among the pairs in

$$\delta(q, (\beta_b(\sharp(\sigma)))_{\sigma\in\Sigma}) \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \ .$$

We say that a state transition is *deterministic* if

$$|\delta(q, (\beta_b(\sharp(\sigma)))_{\sigma\in\Sigma})| = 1 \ .$$

To make sure that our state transitions are well-defined, we require that

$$|\delta(q, (\beta_b(\sharp(\sigma)))_{\sigma\in\Sigma})| \geq 1$$

for all q and $(\beta_b(\sharp(\sigma)))_{\sigma\in\Sigma}$.

Crash Failures. In our network model, the nodes may crash arbitrarily. We assume that the schedule of these crash failures is controlled by an *oblivious* adversary. Formally, the strategy of the adversary is a mapping from the round indices (natural numbers) to node subsets, where this mapping may depend on the protocol Π, but not on the random choices made during the execution. A crash failure of node u in round r indicates that all nodes including u execute round r and after round r and before round $r+1$, node u (along with all of its adjacent edges) is removed from G. Throughout the paper, we denote the number of crash failures by C. We point out that if an edge $e = (u, v)$ is removed due to a crash failure of u or v, then the corresponding ports $\phi_u(v)$ and $\phi_v(u)$ of the adjacent nodes u and v are removed as well and the messages stored there cannot be read anymore.

Correctness. A protocol Π for problem P is said to be correct if the following holds for every instance of P and for any strategy of the adversary. With probability 1, Π eventually reaches an output configuration where the output of the correct nodes is a valid solution to P. In other words, if a valid output configuration becomes invalid due to crash failures, then the protocol must eventually change the output configuration into a valid one.

Runtime. We call a round r *silent*, if all the non-crashed nodes are in an output state and the output configuration corresponds to a valid solution to P in round r. The *global* runtime of a correct protocol Π for problem P is the number of non-silent rounds in an execution. In other words, we count the number of rounds in an infinite execution ignoring the silent rounds, where the configuration of the nodes corresponds to a valid solution to P.

In addition, we measure a runtime of each individual node. We say that node u is *active* whenever it is in state $q \in Q \setminus Q_O$. The runtime of node u is defined to be the number of rounds in which u is active.

We say that node u is *affected* if a crash failure occurs in the immediate neighborhood of u. Futhermore, we say that a protocol Π is effectively confining, if the following properties hold. For all non-affected nodes, the runtime of Π is $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ and the global runtime of Π is $\mathcal{O}(\log^{\mathcal{O}(1)} n)$,

when amortized over the number of crash failures, i.e., $\mathcal{O}((C+1)\cdot\log^{\mathcal{O}(1)} n)$ in total. Note that a bound on the global runtime directly implies the same bound for the runtime of any affected node.

Restrictions on the Output States. In [36], the nodes are not allowed to change their output once they have entered an output state. However, they do allow for many output states that correspond to one output o. In other words, it is required that the output states $P_o \subseteq Q_O$ that correspond to each problem output o form a sink, i.e., there does not exist any state transition from P_o to $Q \setminus P_o$. Since our model accommodates crash failures, which might change a correct output state into an incorrect one, we lift this restriction, allowing transitions from an output state to any other state, thus providing the protocol designer with the possibility to escape output configurations that become invalid. We introduce the following new restrictions whose role is to prevent nodes in an output state from taking any meaningful part in the computation process:

1. each possible problem output is represented by a single output state;
2. all transitions originating from an output state must be deterministic; and
3. a transition from an output state to itself always transmits ϵ (the empty letter).

Notice that these additional restrictions are necessary for the soundness of our runtime definition.

6.3 Maximal Independent Set

The main goal of this paper is to design a protocol under the Stone Age model for the Maximal Independent Set (MIS) problem that is able to tolerate crash failures. A set of nodes $I \subseteq V$ is independent if for all $u, v \in I$, $(u, v) \notin E$. An independent set I is maximal if there is no other set $I' \subseteq V$ such that $I \subset I'$ and I' is independent.

Following the terminology introduced in the model section, we show that our protocol is effectively confining. In Section 6.5, we provide a

straightforward lower bound example that shows that under our model, our solution is within a polylogarithmic factor from optimal. In other words, the linear dependency on the number of failures is inevitable.

The basic idea behind our protocol is that we first use techniques from [36] to come up with an MIS quickly and then we fix any errors that the crash failures might induce. In addition, we want to minimize the potential damage that a crash failure might induce, i.e., the number of nodes that decided not to be in the MIS and do not have an MIS neighbor after a failure.

In other words, our goal is to first come up with a *proportional* MIS, where the likelihood of a node to join the MIS is inversely proportional to the number of neighbors the node has that have not yet decided their output. We partition the state set of our protocol into two components. One of the components contains the input state and is responsible for computing the proportional MIS. Once a node has reached an output state, it transitions into the second component, which is responsible for fixing the errors, and never enters an active state of the first component. As the next step, we explain the logic behind the first component. The logic of the fixing component is explained in Section 6.4.

6.3.1 The Proportional Component

The component used to compute the proportional MIS follows the design from [36] and next, we show that the runtime of their protocol does not (asymptotically) suffer from failures.

The state set of the **proportional component** consists of states $\{D_1, D_2, U_0, U_1, U_2, W, L\}$, where $Q_a = Q \setminus \{W, L\}$ are referred to as *active* states and $Q_O = \{W, L\}$ as *passive* states. We set D_1 as the initial state. The communication alphabet is identical to the set of states, i.e., node u transmits letter q in round r whenever it resides in state q in round $r + 1$. The transition function of the **proportional component** is depicted in Figure 6.1.

In the **proportional component**, each state $q \in Q$ is delayed by a set $D(q)$ of delaying states of q. For state $q \in Q$ the set of delaying states corresponds to the states $S \subseteq Q \setminus q$ from which there is a state transition to q. State q being delayed by state q' indicates that a node u stays in

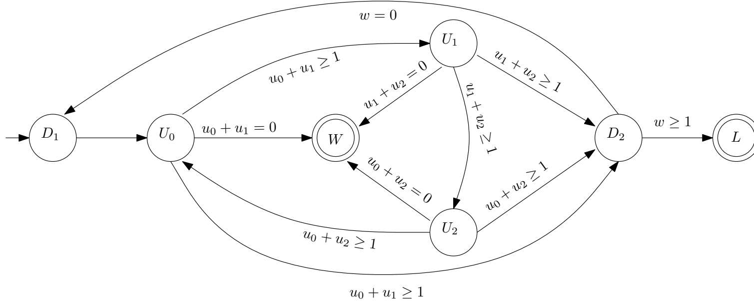


Figure 6.1: The transition function of the **proportional component**. Each edge describes a possible transition and is labeled by the condition that is associated with the corresponding state transition. In addition, each state q is delayed by state q' if there exists a transition $q' \to q$ (the rules associated with the delays are omitted from the figure for clarity).

state q as long as there is at least one letter in its ports that corresponds to q'.

The main idea of the **proportional component** is that each node goes into state U_0 along with its active neighbors and then competes against them. During each round every node u in state $U_j, j \in \{0, 1, 2\}$, assuming that it is not delayed, tosses a fair coin and proceeds to $U_{j+1 \mod 3}$ if the coin heads and to D_2 otherwise. If a node observes that it is the only node in its neighborhood in a U-state, it enters state W, which corresponds to joining the MIS. In the case where, due to unfortunate coin tosses, u moves to state D_2 along with all of its neighbors, node u restarts the process by entering state D_1.

We call a maximal contiguous sequence of rounds u spends in state $q \in Q_a$ a *q-turn* and a maximal contiguous sequence of turns starting from a D_1-turn and not including another D_1-turn a *tournament*. We index the tournaments and the turns within the tournament by positive integers. For a more detailed description of the protocol in a static environment, and especially for the following observation, we refer to [36].

Observation 6.1. *[36] Consider some active node $v \in V$ in turn $j \in Z_{>0}$ of tournament $i \in Z_{>0}$ and some active node $u \in N(v)$.*

- *If j is a D_1-turn of v, then u is in either* (1) *the last D_2-turn of tournament $i-1$;* (2) *turn 1 of tournament i; or* (3) *turn 2 of tournament i.*

- *If j is an U-turn of v, then u is in either* (1) *turn $j-1$ of tournament i;* (2) *turn j of tournament i;* (3) *turn $j+1$ of tournament i; or* (4) *the last D_2-turn $j' \leq j+1$ of tournament i.*

- *If j is a D_2 turn of v, then u is in either* (1) *an U-turn $j' \geq j-1$ of tournament i;* (2) *the last D_2-turn of tournament i; or* (3) *turn 1 of tournament $i+1$.*

6.3.2 An Auxiliary Execution of the Proportional Component

In this section we present an auxiliary execution $\mathcal{E}$ of the **proportional component** that we use to bound the maximum number of tournaments any node u participates in. As a by-product, we get an upper bound for the number of rounds u spends executing the **proportional component**. We note that $\mathcal{E}$ is coupled with an execution η of the **proportional component** with the same sequences of coin tosses, i.e., when we compare η and $\mathcal{E}(\eta)$, where $\mathcal{E}(\eta)$ is the auxiliary execution coupled with η, we assume that the (infinite) sequence of coin tosses by any node u is the same in η and in $\mathcal{E}(\eta)$.

We begin by observing that the adversarial strategy combined with the sequences of coin tosses made by all nodes uniquely determine the tournament index i and the turn of this tournament in which u is crashed (if at all). The crux of $\mathcal{E}(\eta)$ is that we weaken the execution η by crashing node u in the very end of the tournament in which it would have been crashed according to η.

Let $t+1$ be the first round when node u resides in either state D_2 or W of tournament i. In other words u resides in state U_j for some $j \in \{0,1,2\}$ in round t, is not delayed and either throws tails in round t or t is the first round in which there are no neighbors of u in an U-state. Then crashing node u in the end of tournament i indicates that u is crashed after round t but before round $t+1$, which ensures that all neighbors of u will observe all U-turns of u and no node will observe u transitioning into D_2 or W.

To ensure that $\mathcal{E}(\eta)$ never progresses more than η, we restrict the active nodes in each tournament i to the nodes that are active in tournament i according to η. That is, let V_i be the set of nodes for which tournament i exists in η. Then, only the nodes in V_i will execute tournament i in $\mathcal{E}(\eta)$.

Let $X^u(i)$ and $Y^u(i)$ denote the number of U-turns in tournament i of node u according to η and $\mathcal{E}(\eta)$, respectively. Let $N_i(u)$ denote the (exclusive) neighborhood of u in tournament i. We say that node u wins in tournament i in η if $X^u(i) > X^w(i)$ for all $w \in N_i(u)$ and if u is not crashed in tournament i and analogously for $\mathcal{E}(\eta)$.

The following lemma plays a key role in showing that the number of tournaments in $\mathcal{E}(\eta)$ is equal to the one in η.

Lemma 6.2. *Let V_i be the set of nodes for which tournament i exist in η. If $u \in V_i$ wins in tournament i in $\mathcal{E}(\eta)$, then u wins in tournament i in η.*

Proof. First, we show that $X^v(i) \leq Y^v(i)$ for any tournament i and any node v. We observe that if v is not crashed in tournament i, the number of U-turns is uniquely determined by the coin tosses in both executions, i.e., $X^v(i) = Y^v(i)$. Then if v is crashed in tournament i, there are at least as many U-turns in $\mathcal{E}(\eta)$ as in η. Thus $X^v(i) \leq Y^v(i)$.

Assume that u wins in tournament i according to $\mathcal{E}(\eta)$. Since u can only win in tournament i if it does not crash in this tournament, we get that $X^u(i) = Y^u(i)$. In addition, u winning in tournament i implies

$$X^u(i) = Y^u(i) > Y^v(i) \geq X^v(i)$$

for any $v \in N_i(u)$ and therefore u also wins in η. □

6.3.3 Runtime

Next, we analyze the number of tournaments in **proportional component**. Once we have a bound on the number of tournaments executed in $\mathcal{E}(\eta)$, it is fairly easy to obtain a similar bound for η. To bound the runtime, we first introduce some notation. Similarly as before, the set of nodes participating in tournament i is indicated by V_i. Furthermore, we denote $E_i = E \cap (V_i \times V_i)$ and the subgraph induced by nodes in V_i by $G_i = (V_i, E_i)$.

According to the design of the **proportional component**, no node that once enters a passive state can become active again. Since V_i is the set of nodes participating in tournament i according to the **proportional component**, we get that $V_{i+1} \subseteq V_i$ for any i. The following lemma plays a crucial role in the runtime analysis of the **proportional component**. In the case of no failures, the proof is (almost) the same as the one in [36]. By a delicate adjustment in the details of the proof, we show that the same argument holds in the presence of failures and only affects the constant ℓ by a constant factor.

Lemma 6.3. *There are two constant* $0 < p, \ell < 1$ *such that* $|E_{i+1}| \leq \ell |E_i|$ *with probability* p*.*

Proof. We say that a node v is *good* in G_i if

$$|\{u \in N_i(v) \mid d_i(v) \geq d_i(u)\}| \geq d_i(v)/3 \ .$$

It is known, that at least half of the edges of any graph are adjacent to good nodes [7].

Let us now consider some good node $v \in G_i$. Our goal is to show that at least 1/6 of the edges connected to v in tournament i do not exist in V_{i+1} with a constant probability. We split our analysis into two cases, where the first case considers the option that in tournament i, the adversary either crashes v or at least half of the edges connected to nodes in $\{u \in N_i(v) \mid d_i(v) \geq d_i(u)\}$ by crashing the corresponding neighboring node. It directly follows that at least 1/6 edges connected to v do not exist in V_{i+1} with probability 1.

Let us then assume for the second case that the adversary crashes at most half of the edges connected to

$$\{u \in N_i(v) \mid d_i(v) \geq d_i(u)\}$$

in tournament i. Let $A(u, i)$ be the event that u does not crash in tournament i and

$$\hat{N}_i(v) = \{u \in N_i(v) \mid A(u, i) \wedge (d_i(u) \geq d_i(v))\} \ .$$

We say that node $u \in \hat{N}_i(v)$ wins v in tournament i if

$$X^u(i) > \max\{X^w(i) \mid w \in N_i(u) \cup \hat{N}_i(v) - \{u\}\}$$

and denote this event by $\mathrm{WIN}_i(u,v)$. The idea is that if u wins v in tournament i, then u enters the WIN state and v enters the LOSE state. Furthermore, the events $\mathrm{WIN}_i(u,v)$ and $\mathrm{WIN}_i(w,v)$ are disjoint for every $u, w \in \hat{N}_i(v), u \neq w$.

Let us now fix a node $u \in \hat{N}_i(v)$. We denote the event that the maximum of $\{X^w \mid w \in N_i(u) \cup \hat{N}_i(v)\}$ is attained at a single $w \in N_i(u) \cup \hat{N}_i(v)$ by $B_i(u,v)$. Since

$$|N_i(u) \cup \hat{N}_i(v)| \leq 2d_i(v)$$

by the definition of a good node and $X^w(i)$ are independent geometric random variables for all $w \in N_i(u) \cup \hat{N}_i(v)$, we get that

$$\Pr(\mathrm{WIN}_i(u,v)) = \Pr(\mathrm{WIN}_i(u,v) \mid B_i(u,v)) \cdot \Pr(B_i(u,v)) \geq \frac{1}{2d_i(v)} \cdot \frac{2}{3} .$$

Since v is good in G_i and the events $\mathrm{WIN}_i(u,v)$ disjoint, we get that

$$\begin{aligned} \Pr\left(v \notin V_{i+1}\right) &\geq \Pr\left(\bigvee_{u \in \hat{N}_i(v)} \mathrm{WIN}_i(u,v)\right) \\ &= \sum_{u \in \hat{N}_i(v)} \Pr\left(\mathrm{WIN}_i(u,v)\right) \\ &\geq \frac{d_i(v)}{6} \cdot \frac{1}{2d_i(v)} \cdot \frac{2}{3} = \frac{1}{18} . \end{aligned}$$

Recalling that half of the edges are connected to good nodes, we get that $\mathbb{E}[|E_{i+1}|] < 35/36|E_i|$. The claim now follows by Markov's inequality. $\square$

Theorem 6.1. *Any node $u \in V$ participates in $\mathcal{O}(\log n)$ tournaments before becoming passive with high probability*[2] *and in expectation.*

Proof. Let $Z = \min\{0 \leq i \in \mathbb{Z} \mid |E_i| = 0\}$. By Lemma 6.3, Z is dominated by a random variable that follows distribution

$$\mathcal{O}(\log n) + \mathrm{NB}(\mathcal{O}(\log n), 1-p) .$$

[2] Recall that an event occurs *with high probability*, if the event occurs with probability at least $1 - n^{-c}$, where c is an arbitrarily large constant.

Therefore, $Z = \mathcal{O}(\log n)$ in expectation and with high probability.

According to the design of the protocol, a node that is not crashed and does not have any active neighbors in tournament i goes into state W with probability 1. Therefore, all active nodes in tournament with index Z will either become passive or crash. To conclude the proof, we recall that according to Lemma 6.2 and the construction of $\mathcal{E}$ any node that becomes passive in tournament i according to $\mathcal{E}(\eta)$ also becomes passive in tournament i according to η. □

The last step of our runtime analysis is to bound the actual number of rounds that any node u might spend in an active state of the **proportional component**. Consider the following modification of the **proportional component**: before starting tournament $i+1$, every node waits for every other node to finish tournament i or to become passive. We emphasize that we do not claim that we know how to implement such a modification, but clearly the modified process is not faster than the original one.

The length of tournament i is determined by $\max_v\{X^v(i)\}$. Given that the random variables $X^v(i)$ are independent and follow the geometric distribution with parameter 1/2, we get that $\max_v\{X^v(i)\} \in \mathcal{O}(\log n)$ with high probability and in expectation.

Corollary 6.4. *There exists a time t such that no node is in an active state $q \in Q_a$ of the* ***proportional component*** *and $t \in \mathcal{O}(\log^2 n)$ with high probability and in expectation.*

6.3.4 The Quality of an MIS

The **proportional component** provides us with an efficient MIS protocol that ensures that the nodes that are unaffected by the crash failures form an MIS and that the competitions between these nodes are fair. In Section 6.4, we extend this protocol to cope with the crash failures that break the MIS, i.e., remove an MIS node u from the graph leaving at least one of the neighbors of u without any other neighbors in the MIS. Intuitively, the adversary should aim for failures that leave many nodes without MIS neighbors, so that fixing the MIS takes as long as possible. Before introducing the extension to the **proportional component**, we

first take a closer look at the properties of nodes in the passive states and show that if a node has a high degree, it is either unlikely for this node to be in the MIS or that the neighbors of this node have more than one MIS node in their respective neighborhoods.

Let i be the index of the tournament in which node u enters state W. We say that node u *covers* node $v \in N(u)$ if v entered state L in tournament i. In other words, u won and v lost in tournament i.

Definition 6.5. The quality $q(u)$ of node u is given by

$$q(u) = |\{v \in N(u) \mid u \text{ covers } v\}| \ .$$

The quality $q(U)$ of a set of nodes $U \subseteq V$ is defined as $\sum_{u \in U} q(u)$.

The quality of a node u gives an upper bound on the number of nodes in the neighborhood of u such that u is the only node that covers them. As the next step, we bound the expected quality of any node u.

Lemma 6.6. *For any node u, $\mathbb{E}[q(u)] \in \mathcal{O}(\log n)$.*

Proof. Consider node u and tournament i. Let $\hat{N}_i(u) \subseteq N_i(u)$ be the neighbors of u that are not crashed in tournament i. We note that u can only cover $v \in N_i(u)$ if v is not crashed in tournament i. Let $q_i(u)$ denote the random variable that counts the number of nodes that u covers in tournament i. Since $X^u(i)$ and $X^v(i)$ for all $v \in \hat{N}_i(u)$ are independent random variables that obey the same distribution, and u can only win in tournament i if $X^u(i) > X^v(i)$ for all $v \in \hat{N}_i(u)$, we get that

$$\begin{aligned}\mathbb{E}[q_i(u)] &\leq \Pr\left[X^u(i) > \max\{X^v \mid v \in \hat{N}_i(u)\}\right] \cdot |\hat{N}_i(u)| \\ &\leq \frac{1}{|\hat{N}_i(u)| + 1} |\hat{N}_i(u)| \leq 1 \ .\end{aligned}$$

Consider the random variable

$$Z = \min\{j \in \mathbb{Z}_{\geq 0} \mid |E_j| = 0\} \ .$$

The total number of tournaments is at most Z and therefore

$$\mathbb{E}[q_i(u)] = \mathbb{E}[q_i(u) \mid Z \geq i] \cdot \Pr[Z \geq i] \ .$$

Thus

$$\begin{aligned}\mathbb{E}[q(u)] &= \sum_{i=0}^{\infty} \mathbb{E}[q_i(u) \mid Z \geq i] \cdot \Pr[Z \geq i] \\ &\in \mathcal{O}(1) \cdot \left(\sum_{i=0}^{\infty} \Pr[Z \geq i] \right) \subseteq \mathcal{O}(\log n) \ .\end{aligned}$$

□

Let v be a node in state L. If t is the first round such that there are no nodes in $N(v)$ in state W, we say that v is *released* at time t. Each node can only be released once, i.e., node v still counts as released even if it eventually again has a neighboring node in state W.

The next step is to bound the number of released nodes. The idea is that for any set $U \subseteq V$, the quality $q(U)$ gives an upper bound to the expected number of released nodes when the nodes in U crash.

Lemma 6.7. *Let H be the set of nodes that are eventually released. Then $\mathbb{E}[|H|] \in \mathcal{O}(C \log n)$.*

Proof. For each node $u \in V$, the quality $q(u)$ of u yields an upper bound to the expected number of nodes released when this node is crashed. Let $V_c \subseteq V$ be the set of nodes are eventually crashed. By Lemma 6.6 the quality of any node u is $\mathcal{O}(\log n)$. Therefore

$$\mathbb{E}[q(V_c)] = \sum_{u \in V_c} \mathbb{E}[q(u)] \in |V_c| \cdot \mathcal{O}(\log n) \ .$$

Thus, the expected number of released nodes is at most

$$|V_c| \cdot \mathcal{O}(\log n) \subseteq \mathcal{O}(C \log n) \ .$$

□

6.4 The Fixing Component

Now we extend the **proportional component** to fix the MIS in the case of nodes being released. To detect if all the neighbors in the MIS have crashed, each node u in state L checks in every round that there is at least one message w in its ports. If not, u tries to join the MIS in a similar fashion as in the **proportional component**, but without waiting for its neighbors in the L state.

The state set of the protocol is extended by $Q_2 = \{U', D'\}$, where nodes in either state U' or D' are referred to as active. We refer to the extension of the **proportional component** as the **greedy component**. The state transition function of the protocol is illustrated in Figure 6.2. Note that in the **proportional component**, the output states W and L are sinks, i.e., the only state transitions are self-loops. To implement the detection of crashed neighbors, the **greedy component** adds a state transition from the L state into state D' in the case that a node u resides in state L and does not have any letters w in its ports. State W remains a sink.

The logic of the new states U' and D' is the following: upon wake up, node u in state D' goes into state U' if none of its neighbors is yet in state U'. Then u and its neighbors that transitioned from D' to U' during the same global round compete similarly to the **proportional component**. In every round, u tosses a fair coin and goes back to state D' if the coin tails. We emphasize that nodes in state U' are not delayed by nodes in the state D'. Then, if u is the only node in its neighborhood in the U' state, it declares itself as a winner and moves into the W state. Similarly, if a neighbor of u wins, u goes into state L.

Similarly to the **proportional component**, we call a maximal contiguous sequence of rounds in which node v resides in state U' a *greedy tournament*. Unlike with the **proportional component**, we index these greedy tournaments globally by the time this particular tournament starts. In other words, if node v resides in state D' in round $t-1$ and in state U' in round t, we index this tournament by t. We say that a greedy tournament r is *active* if there is at least one node u in state U' of greedy tournament r.

To bound the total number of non-silent rounds, we first bound the

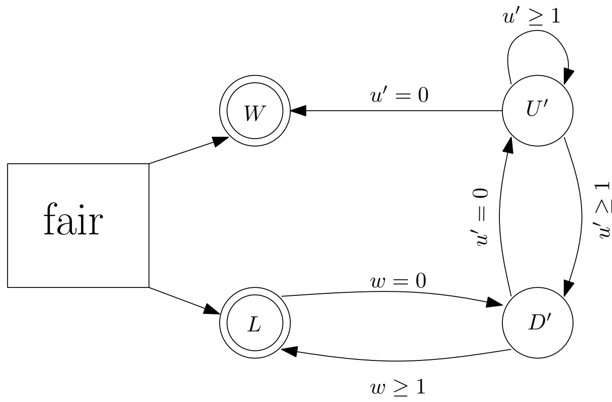


Figure 6.2: The transition function of **greedy component**. The output states W and L are shared among the components. Similarly as in the **proportional component**, the conditions for the transitions are denoted by the labels on the edges. Unlike in the **proportional component** part of the protocol, a transition from q' to q does not imply that q' delays q.

length of each greedy tournament. Then, we give a bound on the total number of greedy tournaments as a function of n and C.

Observation 6.8. *There are at most $\mathcal{O}(\log n)$ rounds in which greedy tournament t is active in expectation and with high probability.*

Proof. Let $V_t \subseteq V$ be the set of nodes that participate in greedy tournament t. Let $t' > t$ be a round in which node $u \in V_t$ is in state U' of greedy tournament t. According to the design of the greedy protocol, u performs a state transition to U' in round with probability at most $1/2$.

We denote the number of transitions from U' to U' in greedy tournament t by node $u \in V_t$ by $X^u(t)$. We note that the number of state transitions in greedy tournament is exactly $X^u(t)$. Furthermore, $X^u(t)$ follows distribution Geom(1/2). Since all $X^v(t), v \in V_t$ are independent and the maximum of $X^u(t)$ for at most n variables is $\mathcal{O}(\log n)$ with high probability and in expectation, the claim follows. □

Lemma 6.9. *The expected number of greedy tournaments throughout the execution is $\mathcal{O}(C \log n)$.*

Proof. To prove the claim, we bound from above the sum of state transitions into state W by the released nodes. Since state W is a sink, we know that each node can enter W at most once. Therefore, after at most $|H|$ transitions into state W by the released nodes, all released nodes are in state W.

We say that a crash *occurs* during greedy tournament t' if any node u is crashed while u is in an active state of greedy tournament t'. If no crash failures occur during greedy tournament t', we say that greedy tournament t' is *clean*.

Assuming that greedy tournament t' is clean and recalling that the random variables $X^u(t')$ are independent, the probability that the maximum of $X^u(t'), u \in V_t$ is attained in a single node is at least $1/3$. Let X_k be the random variable that counts the number of clean greedy tournaments we need until the sum of state transitions to W by the released nodes is k. It is easy to see that X_k is dominated by random variable Y that obeys distribution $k + \mathrm{NB}(\mathrm{k},\ 2/3)$.

Now we set $k = |H|$ and get that $\mathbb{E}[Y] \in \mathcal{O}(k) \subseteq \mathcal{O}(|H|)$. Let T be a random variable that counts the number of released nodes. By Lemma 6.7, $T \in \mathcal{O}(C \log n)$ expectation. Since variables T and X_k are independent, we get that

$$\mathbb{E}[X_T] \in \mathcal{O}(|H|) \subseteq \mathcal{O}(C \log n) .$$

Finally, we observe that there can be at most C greedy tournaments where a crash occurs and thus, the total expected number of greedy tournaments is

$$\mathcal{O}(C \log n) + C \subseteq \mathcal{O}(C \log n) .$$

□

Observation 6.10. *Let k be the total number of greedy tournaments. There can be at most $2k + 2C$ non-silent rounds without an active greedy tournament.*

Proof. Consider a non-silent round t, assume that no crash failures occur and assume that no greedy tournament $t' < t$ is active during rounds t or $t + 1$. Then the logic of the L state ensures, that at least one node will be active in round $t + 1$. Assume then that greedy tournament t is not active in round $t + 1$. This indicates that there has to be a node in state D' and no nodes in state U'. Now the logic of the D' state ensures that greedy tournament $t + 1$ will become active. □

Now we are ready to prove our main result, i.e., that our MIS protocol is indeed effectively confining. In Section 6.6, we extend this result by showing that the protocol is *pseudo-local*, i.e., the runtime of a node only depends on its $(\log n)$-hop neighborhood.

Theorem 6.2. *The expected global runtime of our MIS protocol is $\mathcal{O}((C+1) \log^2 n)$.*

Proof. By Lemma 6.9, we have $\mathcal{O}(C \log n)$ active greedy tournaments in expectation and by Observation 6.8 each greedy tournament takes $\mathcal{O}(\log n)$ rounds with high probability and in expectation. Therefore, we have at most $\mathcal{O}(C \log^2 n)$ non-silent rounds where some greedy tournament is active in expectation.

Combining with Corollary 6.4 and Observation 6.10 and by linearity of expectation, the total expected runtime is

$$\mathcal{O}(\log^2 n) + \mathcal{O}(C \log n) + \mathcal{O}(C \log^2 n) \subseteq \mathcal{O}((C+1)\log^2 n) \ .$$

□

Corollary 6.11. *The runtime of any non-affected node is $\mathcal{O}(\log^2 n)$ in expectation and with high probability.*

Proof. Consider any non-affected node u. According to Corollary 6.4, u will enter a passive state in $\mathcal{O}(\log^2 n)$ rounds with high probability and in expectation. The design of the **proportional component** guarantees that u will only become passive if there is node $v \in N(u) \cup \{u\}$ that is in state W. Since u is non-affected, v will never exit state W and therefore, u will never become active again. □

6.5 Lower Bound

The runtime of our algorithm might seem rather slow at the first glance, since it is linear with the number of crash failures. In this section, we show that one cannot get rid of the linear dependency, i.e., there are graphs where the runtime of any algorithm grows linearly with the number of crash failures. In particular, we construct a graph where the runtime of any algorithm is within a polylogarithmic factor from the $\mathcal{O}((C+1)\log^2 n)$ upper bound given by Theorem 6.2.

The intuition behind the lower bound is that even for a graph with two connected nodes, any algorithm has to perform some sort of symmetry breaking. Furthermore, if the degrees of node u and its neighbors are small, it is likely that crashing u releases some of its neighbors.

Let G_ℓ be a graph that consists of n nodes and $n = 3\ell$. The graph consists of ℓ components B_i, where B_i is a 3-clique for each $0 \leq i < \ell$. In addition, let $u_0, \ldots, u_\ell$ be a set of nodes such that $u_i \in B_i$ for every i. For illustration, see Figure 6.3.

Theorem 6.3. *Let $k > 0$ be a constant. There exists a graph G and an schedule of crash failures such that the runtime of any algorithm on G is $\Omega(C)$ in expectation and with high probability for $C \geq k \log n$.*

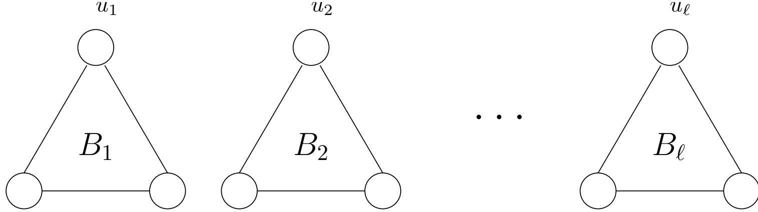


Figure 6.3: One node in each of the connected components has to be in the MIS. Since the nodes in the components are indistinguishable and execute the same protocol, the probability of u_i being in the MIS for any i is 1/3.

Proof. Consider graph G_ℓ and the following adversarial strategy. In round i, the adversary crashes node u_{i-1}, i.e., one of the nodes in component B_{i-1}. Our goal is to show that at least a constant fraction of the first C rounds are non-silent for any $C \leq n/3$. Consider round i and component B_{i-1}. There are two possibilities for round $i+1$ to be non-silent. First, it can be that nodes in B_{i-1} do not represent a valid MIS. Second, the node in B_{i-1} that would join the MIS in round $i+1$ is crashed.

Since the nodes in component B_{i-1} form a clique, their views are identical. Therefore, according to any algorithm that computes an MIS, their probability to join the MIS is equal, i.e., 1/3. Thus, the probability that the nodes in $B_{i-1} \setminus \{u_{i-1}\}$ are not in the MIS in round is 1/3.

Let X then be the random variable that counts the number of non-silent rounds. Since any round i is non-silent with at least probability 1/3, we get that $\mathbb{E}[X] \geq (1/3)C$. Now by applying a Chernoff bound, we get that

$$\Pr[X < 1/2\mathbb{E}[X]] = \Pr[X < (1/2) \cdot (1/3)C] \leq 2^{-C/12} .$$

Since $C = n/3$, we get that $\Pr[X < 1/2\mathbb{E}[X]] \in \mathcal{O}(n^{-k})$ for any constant k and thus, the claim follows. □

6.6 Pseudo-Locality

In this section, we show that the runtime of node u only depends on the number of crash failures within $1 + \log n$ distance from u. Let us denote

the number of crash failures in $N^{1+\log n}(u)$ by C_u, where $N^i(u)$ denotes the i-hop neighborhood of u. We start by using Lemma 6.6 to bound the expected number of released nodes in $N^{\log n}(u)$. A node in $N^{\log n}(u)$ can only be released when a node in

$$N^{1+\log n}(u) \cup \{u\}$$

crashes. Therefore, the quality of the nodes crashed in $N^{1+\log n}(u)$ give an upper bound to the number of released nodes in $N^{\log n}(u)$. The proof for the following lemma is analogous to the proof of Lemma 6.7 and therefore omitted.

Lemma 6.12. *Let H_u be the released nodes in $N^{\log n}(u)$. Then $\mathbb{E}[|H_u|] \in \mathcal{O}(C_u \log n)$.*

Similarly to the proof of the global runtime, we wish to bound the number of greedy tournaments before either all released nodes in $N^{\log n}(u)$ become passive or are crashed. We say that a node *participates* in greedy tournament t if it enters state U' from D' at time t. We show that if at least one node in $N(u)$ participates in greedy tournament i and greedy tournament i is clean, then with a constant probability, at least one node $v \in N^{\log n}(u)$ wins in greedy tournament i. Recall that winning indicates that v enters state W.

Lemma 6.13. *Consider a clean greedy tournament t where at least one neighbor of u participates. Then there exists node $v \in N^{\log n}(u)$ such that v wins with probability at least $1/6$.*

Proof. To prove the claim, we first show that if $|N^i(u)| \leq 2|N^{i-1}(u)|$ for any i, then there is a winning node in $N^i(u)$ with probability at least $1/6$. Let $A(u,i)$ be the event that the maximum is attained in a single node in $N^i(u)$. The probability of this event is $1/3$. Since the coin tosses in $N^i(u)$ obey the same distribution, the maximum is attained equally likely in all the nodes. Therefore, a node from N^{i-1} wins with probability

$$\frac{1}{3}\frac{|N^{i-1}(u)|}{|N^i(u)|} \geq \frac{1}{6} \ .$$

Assume now for contradiction that $|N^i(u)| > 2|N^{i-1}(u)|$ for all $0 \leq i \leq \log n$. It follows that $|N^{\log n}(u)| > 2^{\log n} = n$, which is a contradiction. □

Now with an argument analogous to the one of Lemma 6.9, we get that the expected number of greedy tournaments in which at least one neighbor of u participates is $\mathcal{O}(C_u \log n)$. Furthermore, by considering only the greedy tournaments in which some node $v \in N(u)$ is participating, by Observation 6.10 the expected number of rounds in which u is active and no node in $N(u)$ is participating in a greedy tournament is bounded by $2(C_u \log n)+2C_u$. Finally, employing Observation 6.8, we get the following theorem.

Theorem 6.4. *The expected runtime is $\mathcal{O}((C_u + 1)\log^2 n)$ for any node u.*

7

Mobile Agents

In the previous chapter, we introduced the Stone Age model of distributed computing and furthermore, extended this model to accommodate crash failures. In this chapter, we study a different variant of this model, where the nodes of the distributed system are mobile and thus increase the level of dynamicity.

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. We consider the so-called *Ants Nearby Treasure Search (ANTS)* problem, a natural benchmark for ant-based distributed algorithms where n mobile agents or *ants* try to efficiently find a food source at distance D from the nest. We present a 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, our goal is to allow up to $f \in \mathcal{O}(n)$ 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 [39]. 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 [35]. That approach, however, is rather fragile and requires the agents to be absolutely reliable. The failure of just a single agent can already result in not finding the treasure.

In our model, we allow (in total) up to f of the agents to fail at any point in time, where f is allowed to be at most some fixed constant fraction of the number of agents n. This is, asymptotically, the best we could hope for, since it is clearly impossible to solve the task if all n of the agents fail. 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.

7.1 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 [38,39]. 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 [65]. 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. Thus, the communication and computation model corresponds to a mobile version of the Stone Age model [35].

Searching the plane is a special case of graph exploration. In the general graph exploration setting, the goal is to traverse all the edges of a graph starting from any node. Graph exploration has been extensively studied in the literature and the studies can be divided into two settings. One of the settings is to assume that the graphs are directed, i.e., an edge can only be traversed to one direction, not vice versa [4, 19, 29]. In the other, the edges can be traversed to both directions [10, 30, 33].

Furthermore, there are two main types of performance measures regarding graph exploration. The first measure is the time complexity, i.e., how long does it take for the agent(s) to finish the exploration task [76]. The other one is to measure the bit complexity, i.e., how many bits of memory does the agent(s) require to solve the exploration task [42]. Furthermore, the aforementioned graph exploration tasks can be considered with return, stop, or perpetual properties, i.e., whether the agent is required to return to the starting cell, stop the search after finishing, or if

the agent is not required to terminate [30, 43].

In the case of finite graphs, it is known that a random walk visits all nodes in expected polynomial time [5]. 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 a treasure 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 [16]. The problem has also been studied in a multi-agent setting [70].

Searching graphs with finite state machines was studied earlier by Fraigniaud et al. [43]. Other work considering distributed computing by finite automata includes for example *population protocols* [8, 9].

7.2 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 [35], 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. Furthermore, we consider a synchronous environment, where all agents act simultaneously. In all other aspects, our model is identical to the one of [35].

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* $\text{dist}(c, c')$ 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 $\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}$, 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 $\tau \in \{\mathrm{N}, \mathrm{S}, \mathrm{E}, \mathrm{W}, \mathrm{P}\}$ 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 with high probability 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).

7.3 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. 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 chapter 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 $2n/(g+3)$ with high probability[1]. Hence, a protocol that relies on the survival of its g giants can tolerate $n/(g+3)-1$ failures and still operate correctly.

7.3.1 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 (with high probability 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.

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 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

[1] Recall that an event occurs *with high probability* if the event occurs with probability at least $1 - n^{-c}$, where c is an arbitrarily large constant.

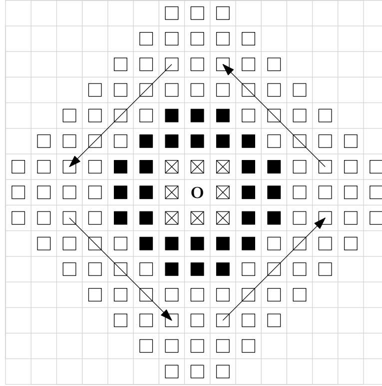


Figure 7.1: This figures shows the ring of cells that is supposed to be explored by the ExpoSweep protocol in iteration 1 (crossed boxes), 2 (filled boxes) and 3 (empty boxes). The width of the ring increases by factor of (roughly) two in each iteration and the agents move further outwards.

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 7.1 gives an illustration of the execution.

7.3.2 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 pa-

rameters 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 7.2 for an illustration.

7.3.3 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 7.1. *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.

7.3.4 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 7.2 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

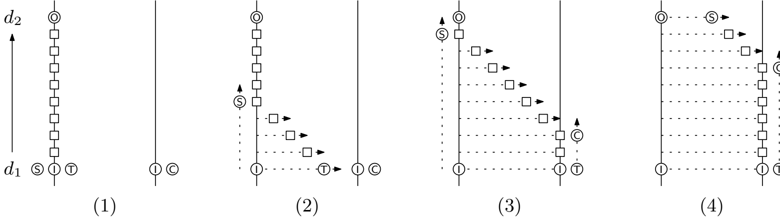


Figure 7.2: This figure illustrates different stages of a QSweep. The two (perpendicular) axes between which the QSweep is performed are aligned parallel to each other for the sake of clarity. (1) shows the (d_1, d_2)-basis while in (2) the StartGiant (S) has already sent on their way several Explorers (□) 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.

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 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 7.2. *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 7.3. *If no agent failed during a* Compact *execution and the subsequent* SegSweep*, then the* SegSweep *was successful.*

Proof. The execution of Compact 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 7.2. □

7.3.5 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 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 $d_3 > d_2 + 1$.

7.3.6 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 jth 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 with high probability

Lemma 7.4. *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 $Z^j < 2/e \cdot 2^j$ with high probability.*

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 [73]. 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 random indicator 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 [73] 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 7.5. *Let E be any subset of (surviving) Explorers of size n_e. After the jth iteration of* UniSplit *for $0 < j < \log(n_e)$, the Explorers in E are members of $\Omega(2^j)$ different teams with high probability.*

Proof. Lemma 7.4 states that after the jth iteration there are at most $2/e \cdot 2^j$ empty bins with high probability. Thus, there are at least $(e-2)/e \cdot 2^j \in \Omega(2^j)$ many non-empty bins with high probability, 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 7.5 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 jth pass of UniSplit for $0 < j < \log(n_e)$ with high probability

Corollary 7.6. *The number of teams after the jth pass of* UniSplit *is $\Omega(2^j)$ for $0 < j < \log(n_e)$ with high probability.*

7.3.7 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 7.1.

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.

1. The StartGiant triggers the execution of COMPACT as described in Section 7.3.3.
2. The StartGiant triggers the execution of SEGSWEEP as described in Section 7.3.4. 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 7.3.5.
4. The StartGiant triggers the execution of UNISPLIT as described in Section 7.3.6.

Algorithm 7.1: EXPOSWEEP

7.4 Runtime

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

Lemma 7.7. *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 7.8. *Any iteration i of* EXPOSWEEP *before the treasure was found lasts at most $\mathcal{O}(D)$ rounds.*

Proof. By Lemma 7.7, $d_2 \leq 2D$ 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 7.1. *The runtime of the* ExpoSweep *protocol is* $\mathcal{O}(D+D^2/n+Df)$ *for* $f = n/13$ *with high probability.*

Proof. By Lemma 7.8, 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 ith iteration of ExpoSweep. Using Corollary 7.6, we can bound the maximum distance explored by the preceding iterations from below by $d(i) = \sum_{j=0}^{i-1} \Omega(2^j) \subseteq \Omega(2^i)$. 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

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

Now let us consider the case when $D \in \Omega(n)$. By Corollary 7.6, 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 7.8, any iteration takes at most $\mathcal{O}(D)$ rounds. The total runtime is therefore

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

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

8 Labyrinth Search

In this chapter, we turn our attention away from the crash failures. Instead, we focus on a different type of challenge for the ants, that is, an obstructed search environment. In other words, instead of an infinite grid, we study a restricted version of the grid, where the ants are not allowed to enter every cell.

Since we are restricting our underlying graph to $\mathbb{Z}^2$ and the obstacles in our domain essentially block the ants from entering specific cells, our graphs correspond to a concept widely studied in literature called *labyrinths* [21, 31]. Exploration of a labyrinth corresponds to the task of getting as far from the starting point as possible, for any starting point. It was shown by Budach that a single automaton cannot explore every finite labyrinth, where a finite labyrinth has only a finite amount of blocked cells [23]. On the positive side, it is known that every finite labyrinth can be explored by a finite automaton using 4 pebbles and that all co-

finite (number of non-blocked cells is finite) labyrinths can be explored with a finite state machine using 2 pebbles [22]. Finally, Hoffman showed that the problem cannot be solved in neither finite nor co-finite labyrinths by using only 1 pebble [55]. Note that our goal differs from the one of labyrinth exploration, i.e., our goal is to visit all non-blocked cells.

In the case of an infinite grid without obstacles, it was discovered by Emek et al. that two deterministic finite state machines cannot discover every cell [34]. In the same work, it was also shown a randomized finite state machine requires infinite time in expectation and that 4 (deterministic) finite state machines are always enough to discover the treasure. Since the unobstructed infinite grid is a special case of a labyrinth, the same lower bounds hold for our problem. In this chapter, our goal is to derive an upper bound for the number of ants required to discover the treasure in the more difficult setting.

8.1 Model

The model description for this chapter follows closely to the one in Chapter 7.4. There are three fundamental differences. First, we consider now an asynchronous model, i.e., one ant can execute an unbounded amount of steps before any other ant performs a single step. Furthermore, the underlying graph is no longer a 4-regular infinite graph and the ants are no longer prone to crash failures.

The set of cells $B \subset \mathbb{Z}^2$ represents the *blocked* cells, which cannot be entered by an ant. All other cells are called *free*. For simplicity, we assume that B neither contains the origin nor any of the cells within distance at most 3 from the origin. We note that assuming the origin free is necessary and that our protocols can easily be modified to work in an environment where we do not assume that the nearby cells around the origin are free. This assumption merely allows for a cleaner and more reader friendly initialization of our protocols.

To make the exploration of the grid feasible, we require that the cells in B do not fully enclose any free cell, i.e., that any free cell is reachable from any other free cell by a path of neighboring free cells. The set B induces a set $\mathcal{O}$ of *obstacles*. An obstacle $O \in \mathcal{O}$ is a maximal set of connected cells, where two cells are connected if both their x- and y-coordinates

each differ by at most one (diagonally adjacent cells are connected!). We require each obstacle to be of finite size.

Similarly to before, all ants are controlled by the same *finite automaton* (FA). However, since we design a protocol for a constant number of ants, we allow each ant to run a different individual protocol. This is modeled by assigning to each ant an individual initial state in the shared automaton. This assumption allows us to consider deterministic protocols, where each state transition is deterministic (recall the definition from Section 6.2).

In an *asynchronous environment*, the execution of each ant progresses in discrete (asynchronous) steps indexed by the non-negative integers. We denote the time at which ant a completes step $i > 0$ by $t_a(i) > 0$ and call $t_a(i)$ an *activation time*. Following common practice, we assume that the activation times $t_a(i)$ are determined by the policy ψ of an adversary that knows the protocol but is oblivious to its random bits, whereas the ants do not have any sense of time. In order to prevent the adversary from delaying a single ant arbitrarily long, we require a policy to activate each ant at least once every time unit, i.e., for all ants $t_a(1) \leq 1$ and $t_a(i+1) - t_a(i) \leq 1$ for $i > 0$. The set of activation times determined by the adversary is called a *schedule* and we will use the terms synchronous/asynchronous policy and -/- schedule interchangeably in the rest of the paper, despite their subtle difference. The special case of a synchronous environment corresponds to the case where $t_a(i) = i$ for all ants a and all $i > 0$.

Formally, the ants' protocol is captured by the 3-tuple

$$\Pi = \langle Q, s_0^a, \delta \rangle \ ,$$

where Q is the finite set of *states*; $s_0^a \in Q$ is the *initial state* of ant a; and

$$\delta : Q \times 2^Q \times \{\top, \bot\}^4 \to 2^{Q \times \{\mathrm{N,E,S,W,P}\}}$$

is the *transition function*. At time 0, all ants are positioned at the origin and their FAs are in the respective initial states. Suppose that at time $t_a(i)$, ant a is in state $q \in Q$ and positioned in cell $z \in \mathbb{Z}^2$. Then, the state $q' \in Q$ of a at time $t_a(i+1)$ and its corresponding position transition $\tau \in \{\mathrm{N,E,S,W,P}\}$ are determined by the transition function

$\delta(q, Q_a, b) = (q', \tau)$, where $Q_a \subseteq Q$ contains state $p \in Q$ if and only if there exists some (at least one) ant $a' \neq a$ such that a' is in state p and positioned in cell z at time $t_a(i)$, and b is a 4-tuple indicating which of the neighboring cells N/E/S/W are blocked ($\top$) or free ($\perp$). If the transition function dictates that an ant enters a blocked cell, the ant stays put instead. For simplicity, we assume that while the state subset Q_a (input to δ) is determined based on the status of cell z at time $t_a(i)$, the actual application of the transition function δ occurs instantaneously at the end of the step, i.e., ant a is considered to be in state q and positioned in cell z throughout the time interval $[t_a(i), t_a(i+1))$.

8.2 Basic Idea

In order to find the treasure, the ants have to visit every free cell. The high level idea is that the ants walk in growing squares counter-clockwise around the origin. To this end, each ant is given a specific task. An *explorer* explores the plane by walking along squares of increasing sizes, whereas four other ants, called *guides*, mark the four corners of the square that the explorer should walk along. We identify the four guides by the cardinal direction of their respective corner NE, NW, SW, SE. Upon entering a cell with a guide, the explorer accompanies the guide to the correct position for the next square before continuing the search. After updating the position of the last guide, the explorer starts a new search along the next bigger square. We define *square*(d) as the square given by the four corner cells $(d, d), (d, -d), (-d, -d), (-d, d)$.

In the presence of obstacles, the subroutines get more involved. Obstacles can obstruct the path of the explorer or hinder a guide to mark the cell it is supposed to. To solve the former of the aforementioned problems we provide a subroutine that essentially allows the explorer to walk "through" the obstacle. For the second problem we change the conditions for the guides. Instead of marking the corner of the square, a guide has to either mark the correct y-coordinate or the correct x-coordinate, depending on the guide. The NE- and SW-guides mark the y-coordinates of the corners of the square whereas the NW- and SE-guides mark the x-coordinates of said corners (see Figure 8.2).

Let us describe the new condition for the NE-guide. Consider the NE-

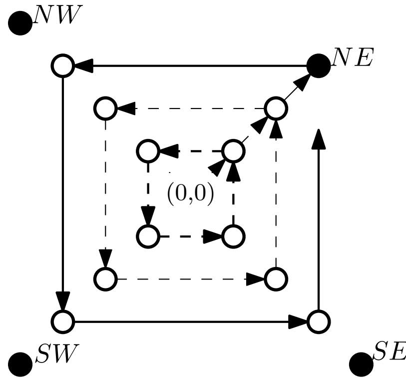


Figure 8.1: The filled black dots represent the corner ants (N, E, S, W), marking the next spot, where the exploring group should turn counter-clockwise in order to walk a square. The hollow dots represent where the corner ants were in earlier stages. The arrows present the way the exploring group was taking so far.

guide that is supposed to mark the cell $z = (d, d)$ for some value of d and assume further that z is blocked. Then, the *surrogate cell* for the cell z is given by $z' = (x', d)$ where $x' = \min\{x \mid x \geq d \wedge (x, d) \notin B\}$. Informally, z' is the first free cell with the same y-coordinate as z further away from the origin. As the obstacles are of finite size we can guarantee that such a cell always exists. With this condition, we make sure that the guide is either on the corner (if it is free) or outside the square on which the explorer is walking.

The condition for the other three guides is analogous. Consider now square(d) and a guide responsible for the corner $x \in \{\text{NE}, \text{NW}, \text{SW}, \text{SE}\}$ of said square. Then, we denote by $Z_x(d)$ the cell where this guide will be positioned during the exploration of the square.

$$
\begin{aligned}
Z_{\text{NE}}(d) &= (x', d) \text{ where } x' = \min\{x'' \mid x'' \geq d \wedge (x'', d) \notin B\},\\
Z_{\text{NW}}(d) &= (-d, y') \text{ where } y' = \min\{y'' \mid y'' \geq d \wedge (-d, y'') \notin B\},\\
Z_{\text{SW}}(d) &= (x', -d) \text{ where } x' = \max\{x'' \mid x'' \leq -d \wedge (x'', -d) \notin B\},\\
Z_{\text{SE}}(d) &= (d, y') \text{ where } y' = \max\{y'' \mid y'' \leq -d \wedge (d, y'') \notin B\}
\end{aligned}
$$

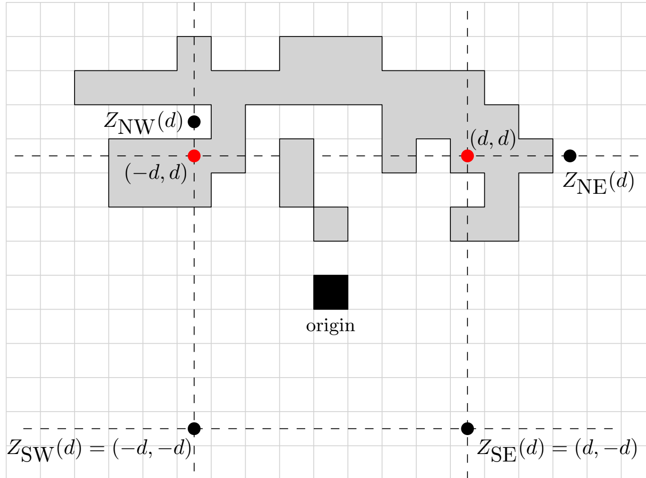


Figure 8.2: The red dots indicate where the NE- and NW-guide would be if there was no obstacle. The black dots indicate the cells, that the guides actually mark. The dashed lines indicate the side of the square that the respective guide is marking and altogether mark the square that the explorer is supposed to walk along.

8.3 Basic Capabilities

Our protocol requires the ants and in particular the explorer to be able to perform various advanced maneuvers. They have to be able to walk along the boundary of an obstacle, memorize their offsets from other cells, be able to find back to a cell they previously occupied, update the position of a guide to the next square, and, most importantly, to virtually walk through an obstacle. In this section, we will present the basic routines which are then combined in Section 8.4 to obtain the more complex ones.

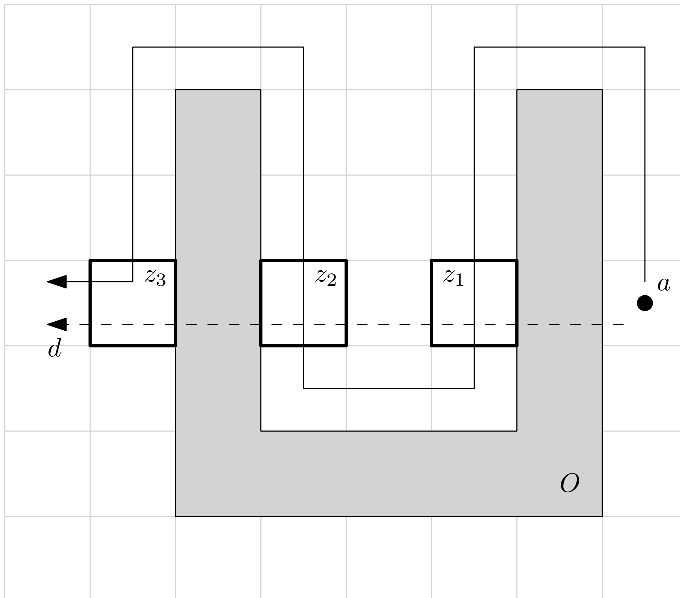


Figure 8.3: ant a wants to "walk through" an obstacle in a straight line in direction d, which is accomplished by tracing the boundary of the obstacle along the path p to locate the cell where the straight line exists the obstacle and then continue walking straight.

8.3.1 Walking Around an Obstacle

Consider an ant a that currently walks into direction h where h can be N/E/S/W and is called the *heading* of a. We say that a turns right or left as shorthand for a changing its heading to an adjacent cardinal direction. Now suppose that ant a is in cell $z = (x, y)$ and the cell $z + h$ is blocked by the obstacle O that a intends to walk around. In the very first step, a turns right so that the obstacle is on its left side — an invariant that will be maintained during the process of walking around the obstacle. Then, in every following step, a first checks if the cell on the left side with respect to the current heading is blocked. If this is the case, a walks once towards its heading, if possible. In case the cell towards the heading is blocked, a turns right. In the case that the cell on the left is free, a turns left and walks once towards the new heading. We can verify that this case only occurs if in the previous step, a moved towards its current heading and therefore, the cell on the left was blocked. Thus, the obstacle will again be left of a in the next step. Assuming that ant a is positioned in a cell along the border of the obstacle O and the cell left of a (with respect to h) is blocked by the obstacle O, the details of the method for a single step are given in Procedure 8.1. As the procedure ensures the aforementioned invariant, ant a can execute it repeatedly to traverse the complete boundary of the obstacle.

```
ant a is located in (x, y) and has heading h.
if cell on left is free then
  turn left.
else if (x, y) + h is blocked then
  while (x, y) + h is blocked do
    turn right.
  end while
end if
move once towards h.
return h.
```

Algorithm 8.1: StepCounterClockwise()

8.3.2 Bounded Offset Counter

In this section, we explain how the ants can simulate a bounded counter suitable to memorize offsets to cells while moving along the boundary of an obstacle. The counter provides the basic operations ON, OFF, ISNULL, ISPOSITIVE, ISNEGATIVE, INCREMENT, and DECREMENT, which activate/deactivate the counter, allow the ant to determine whether the offset is zero/positive/negative, or to increment/decrement it, respectively. It is important to note that our implementation of the offset counter is only available while the ant is adjacent to an obstacle and while this obstacle stays the same. As soon as the ant moves to a cell that is not adjacent to the obstacle anymore, the value of the counter becomes invalid. Hence, our protocols ensure that the counter is always turned off before leaving an obstacle. Moreover, the value of the counter only works correctly as long as its value is bounded by the circumference of the obstacle. This does not pose a problem, however, as all offsets that the ants need to store are bounded appropriately.

We first give an informal description of our implementation and then specify how the basic operations can be implemented. Consider an ant a located in a cell (x, y) adjacent to an obstacle O. Ant a is equipped with the counter c represented by the auxiliary ants a_c, a_b, and a_m called *count ant*, *base ant*, and *messenger ant*, respectively. When the counter is turned off, the auxiliary ants are in the *follow mode*, which implies that they simply follow ant a and do not perform any specific task. When the counter is turned on, the auxiliary ants enter the *counter mode* and perform special tasks. The job of a_b is to mark the cell where the counter has been turned on the last time. Ant a_c's task is to store an offset value v by residing in the cell that is reached when starting in the cell containing a_b and walking $|v|$ cells clockwise along the boundary of the obstacle O. In order to distinguish positive and negative offsets, a_c encodes the sign of v in its states. Ant a_m generally resides in the same cell as ant a and moves to a_c and a_b when the counter is to be changed or read. Either of the basic operations can only be executed when the previous operation has been completed, which is the case when a_m is in the same cell as a.

For the purpose of argumentation, we denote the value represented by counter c as *val*(c). We remark, however, that this value is not directly

accessible to any of the ants.

Operation On(c). When a activates the counter, it signals this to the auxiliary ants using a special state, upon which they enter their respective counter mode states.

Operation Off(c). Ant a_m moves clockwise around the obstacle, instructs a_c and a_b to move along the obstacle to the cell containing a, and finally does the same. The auxiliary ants then enter the follow mode.

Operation IsNull(c). Ant a_m walks clockwise until it locates the cell containing ant a_b. It checks whether ant a_c occupies the same cell and reports this information to ant a.

Operation IsPositive/IsNegative(c). Ant a_m walks clockwise until it locates the cell containing the ant a_c. If the cell also contains ant a_b — the value of the counter is zero — ant a_m reports `false` to a . Otherwise, a_m senses the sign of c through the state of a_c and reports the result to a accordingly.

Operation Increment/Decrement(c). Ant a_m walks clockwise until it locates the cell containing ant a_c. It then instructs a_c to increment/decrement and returns to ant a. Depending on whether the state of a_c corresponds to a positive or negative sign, a_c moves one cell clockwise or counter-clockwise along the obstacle. If a_c resides in the same cell as a_b, it also needs to change its sign state accordingly.

These operations complete the specification of the counter functionality.

8.3.3 Combining Offset Counters

The ants in our protocol sometimes employ a constant number of offset counters c_1 to c_k on the same obstacle, where the respective counters are activated in the same cell. This functionality can be provided by having one base ant a_b and one messenger ant a_m and k count ants for the different counters. To ensure that the messenger interacts with the correct count ant, they encode an index in their states such that the messenger ant can distinguish them. Correspondingly, the messenger ant encodes the index of the counter that it is operating on in its state. As only a constant number of offsets are used, this is possible with a finite

automaton. We distinguish the count ants of different counters by their index as superscript, i.e., a_c^i is the count ant of the counter c_i.

When an ant uses several counters, it has access to two additional operations. Operation LessThan(c_i, c_j) compares the value of two counters and returns a boolean indicating whether $\text{val}(c_i) < \text{val}(c_j)$. The operation Set$(c_i, c_j)$ sets the value of counter c_i to $\text{val}(c_j)$.

Operation LessThan(c_i, c_j). Ant a_m moves clockwise around the obstacle until it locates the cell containing a_b. Then, a_m walks further clockwise around the obstacle until having located both a_c^i and a_c^j. Based on the signs encoded in the states of a_c^i and a_c^j and the order in which these ants were located, a_m infers the result of the comparison, then returns to a and signals it.

Operation Set(c_i, c_j). Ant a_m walks along the obstacle to the cell containing a_c^i and instructs a_c^i to walk to the cell containing a_c^j, while a_m accompanies a_c^i on its way. When a_c^i enters the cell containing a_c^j, ant a_c^i updates its sign to the sign of a_c^j and ant a_m returns to a to finish the operation.

8.4 Advanced Procedures

In this section, we combine the basic functionalities described in the previous section into the complex procedures, that eventually constitute our search protocol. The most important functionality is the ability to virtually walk through an obstacle following a horizontal or vertical straight line. The ants do this by locating the closest cell that lies on the straight line through the obstacle and then continue the walk from there. This functionality is realized by the procedures Shift and Probe that will be described next.

8.4.1 Shifting the Position Along an Obstacle

The procedure Shift(c_x, c_y) allows an ant a positioned in cell $z = (x, y)$ next to the obstacle O and equipped with two counters c_x and c_y to move to the cell $z' = (x + \text{val}(c_x), y + \text{val}(c_y))$ where z' must be also next to O. During the process, ant a continuously updates the counters to reflect the

new offsets, so that when a has reached cell z', the values of both counters c_x and c_y are zero. Consequently, both counters are then turned off.

```
while ¬IsNull(c_x) ∨ ¬IsNull(c_y) do
  h ← StepCounterClockwise()
  Increment(c_x) / Decrement(c_x) according to h
  Increment(c_y) / Decrement(c_y) according to h
end while
Off(c_x); Off(c_y)
```

Algorithm 8.2: SHIFT(c_x, c_y)

8.4.2 Probing Target Cells

While the procedure STEPCOUNTERCLOCKWISE allows the ant a to walk around an obstacle O, it still needs to figure out which of the cells visited along the walk is the next free cell t along the straight path through O. There are two main difficulties that we face when trying to identify t. First, the circumference of O can be arbitrarily large and therefore, a single ant cannot keep track of its relative location with respect to its starting cell $z = (x_b, y_b)$. Second, there might be many possible cells along the edges of O that are hit by the straight line through O. We refer to all these cells along the border of O as *potential target cells* (cf. Figure 8.4).

The procedure PROBE allows an ant a located at cell z to locate the closest potential target cell z^* in direction of the heading h and returns a counter representing the distance of z^* relative to z. The exact formulation of PROBE depends on the heading h of a. Procedure 8.3 gives a pseudo-code description for the case of $h = W$, the other cases are analogous.

The idea is that ant a employs three counters c_x, c_y and $c_{\min}$ while walking along the boundary of O. The counters c_x and c_y track the offset of a from the initial cell (x_b, y_b). Whenever c_y is zero, a has located a cell with the same y-coordinate and the value of c_x is stored in $c_{\min}$ if it is smaller than the previous $c_{\min}$. This process is iterated until the ant returns to the starting position (it meets ant a_b again). Then it turns off

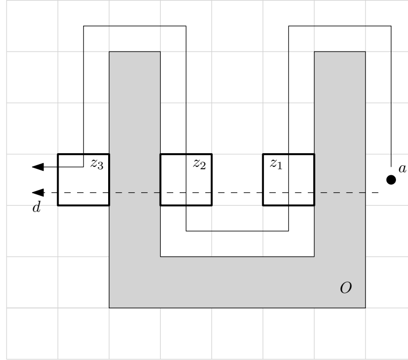


Figure 8.4: Ant a wants to walk west but the direct path (dashed arrow) is obstructed by an obstacle O. Thus, a walks counter-clockwise around the boundary of O (continuous arrow) and uses offset counters to detect the potential target cells z_1, z_2, and z_3.

counters c_x and c_y and returns $c_{\min}$.

8.4.3 Procedure Scan

A detail that we have to be careful with is, when traveling from one guide to another, that each cell along the current square gets discovered and that we eventually reach the guide. To this end, the explorer visits each cell on the boundary of an obstacle that it meets using the procedure Scan.

When an ant a executes Scan, it first activates two counters c_x and c_y. Then, it walks once around the obstacle by repeatedly invoking StepCounterClockwise and updating c_x and c_y according to its actual movements. If a meets the next guide along the way, it does not update the counters anymore. When a returns to the cell containing the base ant a_b of its counter, the walk is finished. If both c_x and c_y equal 0, no guide was not found during Scan. Otherwise, the values of the counters represent the offset to the guide and the procedure "returns" the two counters c_x and c_y. Since a might meet different guides, it stores

```
ON(c_x); ON(c_y); ON(c_min)
repeat
  h ← STEPCOUNTERCLOCKWISE().
  INCREMENT(c_x) / DECREMENT(c_x) according to h
  INCREMENT(c_y) / DECREMENT(c_y) according to h
  if ISNULL(c_y) ∧ (ISNULL(c_min) ∨ LESSTHAN(c_x, c_min)) then
    SET(c_min, c_x).
  end if
until a meets a_b
OFF(c_x); OFF(c_y)
return c_min
```

Algorithm 8.3: $\text{PROBE}_W()$

the index of the next guide that it is supposed to meet according to the protocol in its state, thereby allowing it to ignore all other guides.

8.4.4 Procedure Update

As the last building block of our algorithm, we establish the procedure UPDATE(M) that updates the location $Z_M(d)$ of some guide M to $Z_M(d+1)$ for any $d > 0$.

Consider UPDATE(NW) on position $Z_{\text{NW}}(d)$. We access to counter c_y, denoting the y-offset to the line $y = d$ (as seen later, the protocol using UPDATE provides this counter if the offset is greater than 0, otherwise the explorer turns on the counter c_y). Procedure UPDATE first locates a cell z whose x-coordinate is $-(d+1)$ and whose y-coordinate is greater than d. If the neighbor cell z' west of $Z_{\text{NW}}(d)$ is blocked, the explorer walks around the obstacle containing z' until an appropriate cell z is found. Then, the explorer repeatedly uses SHIFT(0, PROBE) to find the cell $Z_{\text{NW}}(d+1)$. To update the position of the guide, the explorer instructs the guide to follow it on every step until cell $Z_{\text{NW}}(d+1)$ is reached.

Lemma 8.1. *The procedure* UPDATE(M, c_y) *enables the the explorer to move from cell* $Z_M(d)$ *to cell* $Z_M(d+1)$ *and back to cell* $Z_M(d)$.

The description of UPDATE consists of many special cases and the

correctness generally follows from carefully examining the details of the description. Therefore, we dedicate the rest of the section to provide the details of UPDATE and simultaneously establish Lemma 8.1. Consider UPDATE in the case of the NW-guide currently occupying cell $Z_{\mathrm{NW}}(d) = (-d, y^*)$. We assume that the explorer has access to a counter c_y, denoting the y-offset to the line $y = d$. To initialize the update, the explorer leaves another ant to mark $Z_{\mathrm{NW}}(d)$ and instructs the NW-guide to follow the explorer. To locate the cell $Z_{\mathrm{NW}}(d+1)$, our first task is to find a cell $z \in L$, where L is the set of cells whose x-coordinate equals to $-(d+1)$ and whose y-coordinate is at least as large as $d+1$, i.e.,

$$L = \{(i,j) \in \mathbb{Z}^2 \mid (i = -(d+1)) \wedge (j \geq d+1))\} .$$

We divide our description of UPDATE into several cases. First, we consider the case that the cell $z_w = (-(d+1), y^*)$ west to $Z_{\mathrm{NW}}(d)$ is blocked by obstacle O. The explorer turns on the c_x counter. Then, it increments its value by 1 to correspond to the offset from $Z_{\mathrm{NW}}(d+1)$. Also the c_y counter is decremented by 1, to mark the next desired y-coordinate. Refer to Figure 8.5 for an illustration.

To reach a cell $z \in L$, the explorer now simply turns its heading to north to initialize a walk counter-clockwise around O. Now since O is finite, it has to be the case that there is at least one cell from L on the boundary of O. The explorer successively executes STEPCOUNTERCLOCKWISE, updates counters c_x and c_y accordingly, and always checks if $c_x = 0$ and if c_y is positive. If the check returns true, the explorer has reached a cell $z \in L$.

To now find the cell $Z_{\mathrm{NW}}(d+1)$, the explorer first turns its heading towards south and then successively executes SHIFT(0, PROBE), and updates c_y accordingly after every SHIFT, until PROBE returns a value greater than the current c_y. If the next cell found by PROBE is further away than c_y we know that we are in the cell $Z_{\mathrm{NW}}(d+1)$ at the moment. As the last step of this case, the explorer instructs the NW-guide to remain in this cell, and walks counter-clockwise around O until it finds the ant denoting cell $Z_{\mathrm{NW}}(d)$.

Then, consider the case that cell z_w is not blocked. We further split into two cases and we first consider the case that $c_y > 0$, which can be asserted by the explorer by checking if ISPOSITIVE(c_y) returns true. Then,

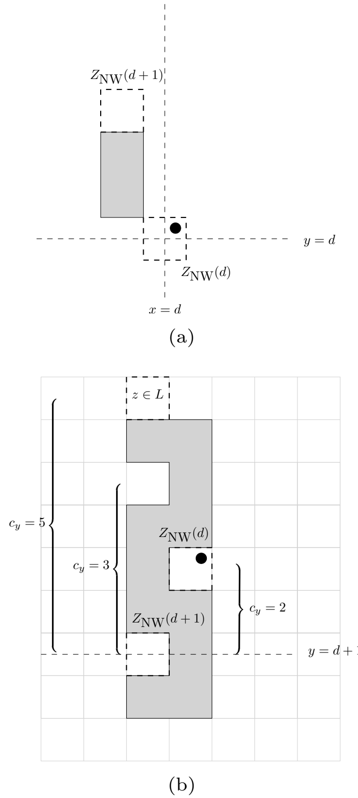


Figure 8.5: Two special cases of UPDATE that considers the case of updating the NWguide. Figure 8.5(a) represents the case where the first cell to the west from $Z_{\mathrm{NW}}(d)$ is free, c_y equals 0, and $Z_{\mathrm{NW}}(d+1)$ is located by moving once west and then executing PROBE and SHIFT with heading N. Figure 8.5(b) illustrates a more complicated case, where there initially is an offset of 2 from the north side of the square($d+1$) (stored in the c_y counter). First, the explorer locates cell z and then executes PROBE and SHIFT until $Z_{\mathrm{NW}}(d+1)$ is located. When $Z_{\mathrm{NW}}(d+1)$ is reached, the value of c_y is 0 and therefore smaller than the value of the counter returned by PROBE.

it has to be the case that all cells $(-d, y^* - i)$, for $i \leq y^* - d$, are blocked by some obstacle O due to the invariant that $Z_{\mathrm{NW}}(d)$ has the smallest y-coordinate among free cells $(-d, y \geq d)$. Thus, the explorer can move to z_w and the counter c_y is still valid. Furthermore, cell $Z_{\mathrm{NW}}(d+1)$ has to be on the boundary of O.

Next, the explorer decrements c_y by 1. If $c_y = 0$, then we have reached cell $Z_{\mathrm{NW}}(d+1)$. Otherwise, similarly to the previous case, the explorer now turns its heading towards south and executes SHIFT$(0, \text{PROBE})$ until PROBE returns a value greater than c_y. When PROBE returns a value greater than c_y, the explorer has reached cell $Z_{\mathrm{NW}}(d+1)$. Similarly to the previous case, the explorer instructs the NW-guide to mark this cell and travels back to $Z_{\mathrm{NW}}(d)$ by walking around obstacle O.

Consider now the case where z_w is not blocked and $c_y \leq 0$. Note that due to the invariant that $Z_{\mathrm{NW}}(d)$ has the smallest y-coordinate among free cells $(-d, y \geq d)$, we get that $c_y = 0$. Therefore, the explorer can turn off both counters c_x and c_y without losing any information. Then, the explorer along with the other ants, moves to cell z_w. After reaching z_w, the explorer turns its heading towards north and if $(-(d+1), d+1)$ is not blocked, it moves once north reaching the cell $Z_{\mathrm{NW}}(d+1)$. After instructing NW to mark $Z_{\mathrm{NW}}(d+1)$, the explorer can find back to $Z_{\mathrm{NW}}(d)$ simply by reversing its movements.

If $(-(d+1), d+1)$ is blocked, the explorer executes SHIFT$(0, \text{PROBE}())$ once, so that it reaches the free cell with the smallest y-coordinate at least $d+1$, i.e., the cell $Z_{\mathrm{NW}}(d+1)$. The explorer again instructs the NW-guide to remain in $Z_{\mathrm{NW}}(d+1)$ and travels back to $Z_{\mathrm{NW}}(d)$ by turning its heading south, executing SHIFT$(0, \text{PROBE})$ once, and moving once east. See Figure 8.5 for an illustration.

In all of the above cases, the guide was left in a cell $Z_{\mathrm{NW}}(d+1)$ yielding the correctness of the update procedure for the NW-guide and the explorer found its way back to the cell $Z_{\mathrm{NW}}(d)$. This concludes the description of UPDATE for the NW-guide. The procedure UPDATE works analogously for other guides. Note that when updating the NE-guide, the explorer does not return back to cell $Z_{\mathrm{NE}}(d)$ and therefore does not leave an ant in that cell either. Thus, Lemma 8.1 follows.

8.5 Searching the Plane

When executing the search protocol SquareWalk, the ants begin the search by four ants moving into the cells $(1,1), (-1,1)$, $(-1,-1)$, and $(1,-1)$, corresponding to $Z_{\mathrm{NE}}(1), Z_{\mathrm{NW}}(1), Z_{\mathrm{SW}}(1)$, and $Z_{\mathrm{SE}}(1)$. Recall that these ants, the guides, essentially mark the corners of the square that the explorer will explore next and that we identify each guide with the cardinal direction of its corner (NE, NW, SW, SE). The explorer e, equipped with a set of counters in follow mode, moves to the NE-guide in the cell $Z_{\mathrm{NE}}(1)$. It then starts to explore square(1) by moving west until it meets the NW-guide in cell $Z_{\mathrm{NW}}(1)$ and, together with the NW-guide, moves to cell $Z_{\mathrm{NW}}(2)$. Then, the explorer returns to $Z_{\mathrm{NW}}(1)$ and moves south towards the SW-guide. It proceeds analogously with the other guides and eventually returns to the NE-guide. After moving the NE-guide to cell $Z_{\mathrm{NE}}(2)$, the explorer does *not* return to $Z_{\mathrm{NE}}(1)$ but instead starts to explore square(2)

Starting from the next iterations, things get more involved as obstacles might be in the way of the explorer or of the guides. Consider the situation that the next square to be searched by the explorer is square(d), every guide M is in the corresponding cell $Z_M(d)$, and the explorer is in cell $Z_{\mathrm{NE}}(d)$. We explain how e can walk from the NE-guide to the NW-guide and thereby explore the north side of square(d); the three other sides of the square are analogous. Procedure 8.4 gives a pseudo-code description in which $z_e = (x_e, y_e)$ denotes the current cell of the explorer while an explanation follows below.

The explorer e sets its heading towards west and, as long as the cell in front is free, moves forward. If e senses an obstacle in front in cell z, e executes Probe to find the next free cell z' in the direction of its heading, resulting in the counter c_{probe} representing the distance between z_e and z'. Then e scans the obstacle using Scan yielding the counters c_x and c_y. If Scan was not successful, i.e., the NW-guide was not located along the obstacle, the counters c_x and c_y are both zero. Now, e moves to z' using Shift($c_{\mathrm{probe}}, 0$) (c_y is reset and used as second parameter) if

```
h ← W //set heading
repeat
  if (z_e + h) ∉ B then
    move(h) //next cell is free
  else
    c_probe ← Probe()
    (c_x, c_y) ← Scan()
    if (IsNull(c_x) ∧ IsNull(c_y)) ∨ LessThan(c_probe, c_x) then
      Off(c_y); On(c_y); Off(c_x)
      Shift(c_probe, c_y) //move to next free cell
    else
      Off(c_probe); On(c_update). //re-use ants from c_probe
      Set(c_update, c_x)
      Shift(c_x, c_y) //move to NW-guide
    end if
  end if
until e meets NW
Update(NW, c_update).
```

Algorithm 8.4: ExploreNorthSide

(i) Scan was not successful, i.e., the NW-guide was not located along the obstacle (corresponding to (IsNull$(c_x) \wedge$ IsNull$(c_y) =$ `true`)) or

(ii) Scan found the next guide but it is further west than the next target cell (corresponding to LessThan$(c_{\text{probe}}, c_x) =$ `true`)

and repeats the above. We note that item *(ii)* is necessary to ensure that all cells are discovered during the search and refer to Figure 8.6 for an example of a problematic case. If $\text{val}(c_{\text{probe}}) \geq \text{val}(c_x)$, which corresponds to LessThan$(c_{\text{probe}}, c_x) =$ `false`, the explorer executes Shift(c_x, c_y) to move to Z_{NW} to meet the NW-guide.

Finally, e uses Update to update the position of the NW-guide from $Z_{\text{NW}}(d)$ to $Z_{\text{NW}}(d+1)$ and returns to $Z_{\text{NW}}(d)$. Then, it sets its heading to south, turns off all counters and starts the analogous procedure ExploreWestSide, this time walking south towards the SW-guide.

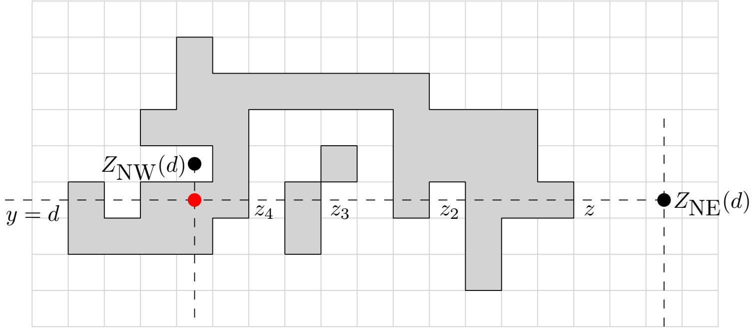


Figure 8.6: Even though ant e encounters the NW-guide already when it scans in cell c there are many more cells to be visited, another obstacle has to be circumvented, before e turns south with the help of the guide.

The above procedure is repeated for all four sides of the square until the explorer arrives back at the NE-guide and updates its position to $Z_{\text{NE}}(d+1)$. Now e does *not* return to $Z_{\text{NE}}(d)$ but instead starts a search of square$(d+1)$ using ExploreNorthSide.

Correctness. In this section, we establish the correctness of the protocol SquareWalk, i.e., that it guarantees that the explorer eventually visits all free cells of the grid. We define the concept of a *configuration* $C : A \mapsto \mathbb{Z}^2$ as an assignment of a cell to each ant. A configuration is a snapshot of the positions of the ants at a given time. The *start configuration for distance* d, denoted by $\mathbf{Z}(d)$, is the configuration where each guide M is in its corresponding cell $Z_M(d)$ and the explorer and the auxiliary ants are in cell $Z_{\text{NE}}(d)$ with the NE-guide. We furthermore define the set

$$F_i = \{(x, y) \notin B \mid (|x| = i \wedge |y| \leq i) \vee (|y| = i \wedge |x| \leq i)\}$$

as the free cells of square(i) for some $i \geq 1$. We are now ready to prove the following theorem which establishes the correctness of SquareWalk.

Theorem 8.1. *The protocol* SquareWalk *guarantees that every cell* $z \in \mathbb{Z}^2$ *is visited by the explorer within finite time.*

Proof. We show by induction over d that for any d, there is a time such that the explorer has visited all cells in $\mathcal{F}_d = \bigcup_{i \leq d} F_i$ and the ants are in $\mathbf{Z}(d+1)$.

The induction base holds by design of the protocol as the ants start the search in configuration $\mathbf{Z}(1)$ and the cells in distance 2 from the origin are free. Hence, the explorer visits all cells and afterwards the ants are in $\mathbf{Z}(2)$.

For the inductive step, assume that the ants are in configuration $\mathbf{Z}(d)$ and all cells in $\mathcal{F}_{d-1}$ have been explored. We consider the walk along the north side of square(d). Let

$$V_d^{\mathrm{N}} = \langle z_0 = Z_{\mathrm{NE}}(d), z_1 = (x_1, d), \ldots, z_k = (x_k, d), z_{k+1} = Z_{\mathrm{NW}}(d) \rangle$$

be the sequence of free cells of the north side of square(d) excluding the corners $\{(-d,d),(d,d)\}$ extended by the cells $z_0 = Z_{\mathrm{NE}}(d)$ and $z_{k+1} = Z_{\mathrm{NW}}(d)$, ordered by descending x-coordinates. Initially, the explorer is located in z_0 and we show that for any $i < k$, the explorer moves to z_{i+1} in finite time.

Consider the case of $i \leq k-1$ and thus $z_{i+1} \neq Z_{\mathrm{NW}}(d)$. If z_{i+1} is neighbor to z_i, the explorer moves to z_{i+1}. If the cell west of z_i is blocked, then Probe finds z_{i+1} and the explorer moves there.

Now consider the case of $i = k$ and thus $z_{i+1} = Z_{\mathrm{NW}}(d)$. If $(-d,d) = Z_{\mathrm{NW}}(d)$ and thus a free cell, the explorer moves there either directly or through Probe/Shift. If $(-d,d)$ is blocked, $Z_{\mathrm{NW}}(d)$ is located along the boundary of the obstacle that blocks the cell $(-d,d)$ by definition. As the explorer explores the boundary of said obstacle using Scan, the explorer is guaranteed to arrive at $Z_{\mathrm{NW}}(d)$. Consequently, all cells in V_d^{N} are visited by the explorer and by Lemma 8.1, we know that the explorer can execute Update in cell $Z_{\mathrm{NW}}(d)$ to move the NW-guide to $Z_{\mathrm{NW}}(d+1)$ and then return to $Z_{\mathrm{NW}}(d)$.

The argumentation for the three other sides of the square is analogous and thus the explorer visits all cells in F_d and then has visited all cells in $\mathcal{F}_d$. After moving the NE-guide to $Z_{\mathrm{NE}}(d+1)$, the explorer stays in the same cell. Hence, the ants are in configuration $\mathbf{Z}(d+1)$, which concludes the inductive step.

The design of our protocol ensures that the ants cannot enter an infinite loop and thus, in every time unit, at least one ant — and thus the

execution of the protocol — progresses. Consequently, the explorer visits every cell in finite time. □

9 Conclusion

In the second part of the thesis, we studied several variants of the Stone Age model introduced in [36], where the nodes are operated by finite state machines and are only allowed to send constant sized messages. We considered several different settings, where the nodes or the agents/ants operating in the system have to adapt to unknown/unpredictable features of the environment. First, we extended their model by accommodating crash failures, i.e., nodes are subject to crash failures in any point of the execution. We presented a fault-tolerant protocol for the MIS problem that works in an arbitrary finite network.

To measure the performance of our protocol, we introduced a new metric, where the runtime of a node is measured in the computationally meaningful steps it performs during the execution and the runtime is amortized over the total number of crash failures. We say that a protocol is effectively confining if the runtime is polylogarithmic for nodes that do

not have any crashed nodes in their neighborhood and if the runtime for other nodes is $\mathcal{O}((C+1)\log^{\mathcal{O}(1)} n)$ for other nodes, where C is the number of failures. In other words, the runtime is polylogarithmic for all nodes when amortized over the number of crash failures.

We introduced an effectively confining algorithm for the MIS problem in our variant of the Stone Age model. We contrasted this upper bound with an almost matching lower bound that states that there are no fault-tolerant MIS protocols with better runtime than $\Omega(C)$. Finally, we showed that the runtime of any node u only depends on the $1+\log n$ neighborhood of u, i.e., our protocol is pseudo-local.

Then, we studied a mobile variant of the Stone Age model, where n ants cooperatively search for a treasure hidden into an infinite grid by an adversary. We presented an algorithm that solves the problem in time $\mathcal{O}(D + D^2/n + Df)$ with high probability, while tolerating $f \in \mathcal{O}(n)$ failures during the execution. Our algorithm uses a combination of a constant number of fault-tolerant giants and $\Theta(n)$ explorer ants, 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.

Then, we turned our attention to a constant number of ants and to a different type of adaptivity. Instead of crash failures, we considered the effects of obstacles in their search environment. We presented the protocol SquareWalk that allows a group of finite state machines to locate the treasure in an infinite grid obstructed by arbitrary obstacles of finite circumference. Our search protocol employs the weak communication capabilities of the ants to simulate a sufficient amount of memory to ensure progress in the search.

Our search protocol requires ten ants in total, where one of the ants acts as an explorer, who performs the searching. The protocol uses three offset counters, requiring five ants. The other four ants mark the sides of a square around the origin that bounds the area discovered so far. In other words, we established that a constant amount of ants is able to locate the treasure in finite time in an arbitrary labyrinth.

Bibliography

[1] Afek, Y., Alon, N., Bar-Joseph, Z., Cornejo, A., Haeupler, B., Kuhn, F.: Beeping a Maximal Independent Set. In: Proceedings of the 25th International Conference on Distributed Computing (DISC). (2011) 32–50

[2] Afek, Y., Alon, N., Barad, O., Hornstein, E., Barkai, N., Bar-Joseph, Z.: A Biological Solution to a Fundamental Distributed Computing Problem. Science **331**(6014) (2011) 183–185

[3] Albers, S.: A Competitive Analysis of the List Update Problem with Lookahead. Theoretical Computer Science **197** (1998) 95–109

[4] Albers, S., Henzinger, M.: Exploring Unknown Environments. SIAM Journal on Computing **29** (2000) 1164–1188

[5] 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 (FOCS). (1979) 218–223

[6] Alon, N., Awerbuch, B., Azar, Y., Patt-Shamir, B.: Tell Me Who I Am: an Interactive Recommendation System. In: Proceedings of

the 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). (2006) 261–279

[7] Alon, N., Babai, L., Itai, A.: A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. Journal of Algorithms **7** (December 1986) 567–583

[8] Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in Networks of Passively Mobile Finite-State Sensors. Distributed Computing (2006) 235–253

[9] Aspnes, J., Ruppert, E.: An Introduction to Population Protocols. In Garbinato, B., Miranda, H., Rodrigues, L., eds.: Middleware for Network Eccentric and Mobile Applications. Springer-Verlag (2009) 97–120

[10] Awerbuch, B., Betke, M., Rivest, R., Singh, M.: Piecemeal Graph Exploration by a Mobile Robot. Information and Computation **152** (1999) 155–172

[11] Awerbuch, B., Patt-Shamir, B., Peleg, D., Saks, M.E.: Adapting to Asynchronous Dynamic Networks. In: Proceedings of the 24th annual ACM Symposium on Theory of Computing (STOC). (1992) 557–570

[12] Awerbuch, B., Patt-Shamir, B., Peleg, D., Tuttle, M.: Collaboration of Untrusting Peers with Changing Interests. In: Proceedings of the 5th ACM Conference on Electronic Commerce. (2004) 112–119

[13] Awerbuch, B., Patt-Shamir, B., Peleg, D., Tuttle, M.R.: Improved Recommendation Systems. In: Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA). (2005) 1174–1183

[14] Awerbuch, B., Sipser, M.: Dynamic Networks are as Fast as Static Networks. In: 29th Annual Symposium on Foundations of Computer Science (FOCS). (1988) 206–219

[15] Azar, Y., Gamzu, I.: Ranking with Submodular Valuations. In: Proceedings of the 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA). (2011) 1070–1079

[16] Baeza-Yates, R.A., Culberson, J.C., Rawlins, G.J.E.: Searching in the Plane. Information and Computation **106** (1993) 234–252

[17] Bar-Noy, A., Bellare, M., Halldórsson, M.M., Shachnai, H., Tamir, T.: On Chromatic Sums and Distributed Resource Allocation. Information and Computation **140**(2) (1998) 183–202

[18] Barenboim, L., Elkin, M., Pettie, S., Schneider, J.: The Locality of Distributed Symmetry Breaking. In: Proceedings of the 53rd Annual Symposium on Foundations of Computer Science (FOCS). (2012) 321–330

[19] Bender, M., Fernandez, A., Ron, D., Sahai, A., Vadhan, S.: The Power of a Pebble: Exploring and Mapping Directed Graphs. In: Proceedings of the 30th annual ACM Symposium on Theory of Computing (STOC). (1998) 269–278

[20] Bikhchandani, S., de Vries, S., Schummer, J., Vohra, R.V.: An Ascending Vickrey Auction for Selling Bases of a Matroid. Operations Research **59**(2) (2011) 400–413

[21] Blum, M., Kozen, D.: On the Power of the Compass (or, Why Mazes Are Easier to Search Than Graphs). In: Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS). (1978) 132–142

[22] Blum, M., Sakoda, W.J.: On the Capability of Finite Automata in 2 and 3 Dimensional Space. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS). (1977) 147–161

[23] Budach, L.: Automata and Labyrinths. Mathematische Nachrichten (1978) 195–282

[24] Chlamtac, I., Kutten, S.: On Broadcasting in Radio Networks–Problem Analysis and Protocol Design. Communications, IEEE Transactions on [legacy, pre - 1988] **33**(12) (1985) 1240–1246

[25] Cole, R., Vishkin, U.: Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. Information and Control **70**(1) (1986) 32–53

[26] Cornejo, A., Kuhn, F.: Deploying Wireless Networks with Beeps. In: Proceedings of the 24th International Conference on Distributed Computing (DISC). (2010) 148–162

[27] Daum, S., Ghaffari, M., Gilbert, S., Kuhn, F., Newport, C.: Maximal Independent Sets in Multichannel Radio Networks. In: Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC). (2013) 335–344

[28] Dean, B., Goemans, M., Vondrák, J.: Approximating the Stochastic Knapsack Problem: The Benefit of Adaptivity. Mathematics of Operations Research **33** (2008) 945–964

[29] Deng, X., Papadimitriou, C.: Exploring an Unknown Graph. Journal of Graph Theory **32** (1999) 265–297

[30] Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree Exploration with Little Memory. Journal of Algorithms **51** (2004) 38–63

[31] Döpp, K.: Automaten in Labyrinthen. Elektronische Informationsverarbeitung und Kybernetik **7**(2) (1971) 79–94

[32] Drineas, P., Kerenidis, I., Raghavan, P.: Competitive Recommendation Systems. In: Proceedings of the 34th ACM Symposium on Theory of Computing (STOC). (2002) 82–90

[33] Duncan, C.A., Kobourov, S.G., Kumar, V.S.A.: Optimal Constrained Graph Exploration. ACM Transactions on Algorithms (TALG) **2**(3) (2006) 380–402

[34] Emek, Y., Langner, T., Stolz, D., Uitto, J., Wattenhofer, R.: How Many Ants Does it Take to Find the Food? In: 21th International Colloquium on Structural Information and Communication Complexity (SIROCCO). (2014) 263–278

[35] Emek, Y., Langner, T., Uitto, J., Wattenhofer, R.: Solving the ANTS Problem with Asynchronous Finite State Machines. In: Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP). (2014) 471–482

[36] Emek, Y., Wattenhofer, R.: Stone Age Distributed Computing. In: Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing (PODC). (2013) 137–146

[37] Feige, U., Lovász, L., Tetali, P.: Approximating Min Sum Set Cover. Algorithmica **40** (2004) 219 – 234

[38] Feinerman, O., Korman, A.: Memory Lower Bounds for Randomized Collaborative Search and Implications for Biology. In: Proceedings of the 26th International Conference on Distributed Computing (DISC), Berlin, Heidelberg, Springer-Verlag (2012) 61–75

[39] 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). (2012) 77–86

[40] Fisher, J., Henzinger, T., Mateescu, M., Piterman, N.: Bounded Asynchrony: Concurrency for Modeling Cell-Cell Interactions. In: Proceedings of the 1st International Workshop on Formal Methods in Systems Biology (FMSB). (2008)

[41] Flury, R., Wattenhofer, R.: Slotted Programming for Sensor Networks. In: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN). (2010) 24–34

[42] Fraigniaud, P., Ilcinkas, D.: Digraphs Exploration with Little Memory. In: 21st Symposium on Theoretical Aspects of Computer Science (STACS). (2004) 246–257

[43] Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph Exploration by a Finite Automaton. Theoretical Computer Science **345**(2-3) (2005) 331–344

[44] Gardner, M.: The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. Scientific American **223**(4) (1970) 120–123

[45] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)

[46] Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.: Fault-containing Self-stabilizing Algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC). (1996) 45–54

[47] Goemans, M., Vondrák, J.: Stochastic Covering and Adaptivity. In: Proceedings of the 7th Latin American Conference on Theoretical Informatics (LATIN). (2006) 532–543

[48] Goldman, S.A., Schapire, R.E., Rivest, R.L.: Learning Binary Relations and Total Orders. SIAM Journal of Computing **20**(3) (1993) 245–271

[49] Golovin, D., Krause, A.: Adaptive Submodularity: Theory and Applications in Active Learning and Stochastic Optimization. Journal of Artificial Intelligence Research (JAIR) **42** (2011) 427–486

[50] Grove, E.: Online Bin Packing with Lookahead. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). (1995) 430–436

[51] Gupta, A., Nagarajan, V., Ravi, R.: Approximation Algorithms for Optimal Decision Trees and Adaptive TSP Problems. In: Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP), Springer-Verlag (2010) 690–701

[52] Hanahan, D., Weinberg, R.A.: Hallmarks of Cancer: The Next Generation. Cell **144**(5) (2011) 646 – 674

[53] Hastad, J.: Clique is Hard to Approximate within $n^{1-\epsilon}$. In: Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS), Washington, DC, USA, IEEE Computer Society (1996) 627–636

[54] Hayes, T., Saia, J., Trehan, A.: The Forgiving Graph: A Distributed Data Structure for Low Stretch Under Adversarial Attack. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC). (2009) 121–130

[55] Hoffmann, F.: One Pebble Does Not Suffice to Search Plane Labyrinths. In: Fundamentals of Computation Theory. Springer Berlin Heidelberg (1981) 433–444

[56] Kaplan, H., Kushilevitz, E., Mansour, Y.: Learning with Attribute Costs. In: Proceedings of the 37th ACM Symposium on Theory of Computing (STOC). (2005) 356–365

[57] Karp, R.M.: Reducibility Among Combinatorial Problems. In Miller, R.E., Thatcher, J.W., eds.: Complexity of Computer Computations. The IBM Research Symposia Series, Plenum Press, New York (1972) 85–103

[58] König, M., Wattenhofer, R.: On Local Fixing. In: 17th International Conference On Principles Of Distributed Systems (OPODIS), Nice, France. (2013) 191–205

[59] Korman, A.: Improved Compact Routing Schemes for Dynamic Trees. In: Proceedings of the 27th Annual ACM Symposium on Principles of Distributed Computing(PODC). (2008) 185–194

[60] Kranton, R., Minehart, D.: A Theory of Buyer-Seller Networks. American Economic Review **91** (2001) 485–508

[61] Kuhn, F., Moscibroda, T., Wattenhofer, R.: What Cannot be Computed Locally! In: Proceedings of the 23th Annual ACM Symposium on Principles of Distributed Computing (PODC). (2004) 300–309

[62] Kuhn, F., Schmid, S., Wattenhofer, R.: A Self-repairing Peer-to-peer System Resilient to Dynamic Adversarial Churn. In: Proceedings of the 4th International Conference on Peer-to-Peer Systems (IPTPS). (2005) 13–23

[63] Kutten, S., Peleg, D.: Fault-Local Distributed Mending. Journal of Algorithms **30**(1) (1999) 144–165

[64] Kutten, S., Peleg, D.: Tight Fault Locality. SIAM Journal on Computing **30**(1) (2000) 247–268

[65] 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) 252–261

[66] Li, X., Misra, J., Plaxton, C.: Active and Concurrent Topology Maintenance. In Guerraoui, R., ed.: Distributed Computing. Volume 3274 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 320–334

[67] Lin, J.C., Huang, T.: An Efficient Fault-Containing Self-Stabilizing Algorithm for Finding a Maximal Independent Set. IEEE Transactions on Parallel and Distributed Systems **14**(8) (2003) 742–754

[68] Linial, N.: Locality in Distributed Graph Algorithms. SIAM Journal on Computing **21**(1) (1992) 193–201

[69] Liu, Z., Parthasarathy, S., Ranganathan, A., Yang, H.: Near-Optimal Algorithms for Shared Filter Evaluation in Data Stream Systems. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. (2008) 133–146

[70] López-Ortiz, A., Sweet, G.: Parallel Searching on a Lattice. In: Proceedings of the 13th Canadian Conference on Computational Geometry (CCCG). (2001) 125–128

[71] Luby, M.: A Simple Parallel Algorithm for the Maximal Independent Set Problem. SIAM Journal on Computing **15** (1986) 1036–1055

[72] Malpani, N., Welch, J.L., Waidya, N.: Leader Election Algorithms for Mobile Ad Hoc Networks. In: Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M). (2003) 96–103

[73] Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York, NY, USA (2005)

[74] Munagala, K., Babu, S., Motwani, R., Widom, J.: The Pipelined Set Cover Problem. In: Proceedings of the 10th International Conference on Database Theory (ICDT). (2005) 83–98

[75] Navlakha, S., Bar-Joseph, Z.: Distributed Information Processing in Biological and Computational Systems. Communications of the ACM **58**(1) (December 2014) 94–102

[76] Panaite, P., Pelc, A.: Exploring Unknown Undirected Graphs. In: Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). (1998) 316–322

[77] Panconesi, A., Srinivasan, A.: On the Complexity of Distributed Network Decomposition. Journal of Algorithms (1996) 356–374

[78] Peleg, D.: Distributed Computing: a Locality-sensitive Approach. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2000)

[79] Schneider, J., Wattenhofer, R.: A Log-star Distributed Maximal Independent Set Algorithm for Growth-bounded Graphs. In: Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC). (2008) 35–44

[80] Scott, A., Jeavons, P., Xu, L.: Feedback from Nature: an Optimal Distributed Algorithm for Maximal Independent Set Selection. In: Proceedings of the 32nd Symposium on Principles of Distributed Computing. (2013) 147–156

[81] Shukla, S., Rosenkrantz, D., Ravi, S.S.: Observations on Self-Stabilizing Graph Algorithms for Anonymous Networks (Extended Abstract). In: Proceedings of the Second Workshop on Self-Stabilizing Systems. (1995) 1–15

[82] Valiant, L.: Parallel Computation. In: 7th IBM Symposium on Mathematical Foundations of Computer Science. (1982)

[83] von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, Champaign, IL, USA (1966)

[84] Walter, J., Welch, J.L., Vaidya, N.: A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks. Wireless Networks **7** (1998) 585–600

[85] Wolfram, S.: A New Kind of Science. Wolfram Media, Champaign, Illinois (2002)

[86] Yao, A.C.C.: Probabilistic Computations: Toward a Unified Measure of Complexity. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS). (1977) 222–227

Curriculum Vitae

1987	Born in Kerava, Finland
2006-2011	Studies in computer science, University of Helsinki
2011	M.Sc. in computer science, University of Helsinki
2009-2011	Research Assistant, Helsinki Institute for Information Technology (HIIT)
2011-2015	PhD Student, Distributed Computing Group, supervised by Professor Roger Wattenhofer, ETH Zürich, Switzerland
2015	PhD Degree, Distributed Computing Group, ETH Zürich, Switzerland

Publications

The following list contains all publications that I have co-authored during my PhD studies. The authors are listed in an alphabetical order.

1. Randomness vs. Time in Anonymous Networks. Jara Uitto, Jochen Seidel and Roger Wattenhofer. *29th International Symposium on Distributed Computing (DISC), October 2015*
2. Ignorant vs. Anonymous Recommendations. Jara Uitto and Roger Wattenhofer. *23rd European Symposium on Algorithms(ESA), September 2015*
3. Lower Bounds for the Capture Time: Linear, Quadratic, and Beyond. Klaus-Tycho Förster, Rijad Nuridini, Jara Uitto and Roger Wattenhofer. *22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, July 2015
4. How Many Ants Does It Take to Find the Food? Yuval Emek, Tobias Langner, David Stolz, Jara Uitto and Roger Wattenhofer. *Theoretical Computer Science*. To appear
5. On Competitive Recommendations. Jara Uitto and Roger Wattenhofer. *Theoretical Computer Science*. To appear
6. SpareEye: A Smart Phone App that Enhances the Safety of the Inattentionally Blind. Klaus-Tycho Förster, Alex Gross, Nino Hail, Jara Uitto and Roger Wattenhofer. *13th International Conference on Mobile and Ubiquitous Multimedia (MUM), November 2014*

7. Towards More Realistic ANTS. Yuval Emek, Tobias Langner, David Stolz, Jara Uitto and Roger Wattenhofer. *2nd Workshop on Biological Distributed Algorithms (BDA), October 2014*
8. Fault-Tolerant ANTS. Tobias Langner, David Stolz, Jara Uitto and Roger Wattenhofer. *28th International Symposium on Distributed Computing (DISC), October 2014*
9. How Many Ants Does It Take to Find the Food? Yuval Emek, Tobias Langner, David Stolz, Jara Uitto and Roger Wattenhofer. *21th International Colloquium on Structural Information and Communication Complexity (SIROCCO), July 2014*
10. Solving the ANTS Problem with Asynchronous Finite State Machines. Yuval Emek, Tobias Langner, Jara Uitto and Roger Wattenhofer. *41st International Colloquium on Automata, Languages, and Programming (ICALP), July 2014*
11. On Competitive Recommendations. Jara Uitto and Roger Wattenhofer. *24th International Conference on Algorithmic Learning Theory (ALT), 2013*