On the Deterministic Gathering of Mobile Agents

Sébastien Bouchard

Thèse présentée pour obtenir le grade de docteur
Sorbonne Université

❧

Laboratoire d'Informatique de Paris 6
École Doctorale Informatique, Télécommunications et Électronique (Paris)

Discipline : Informatique

On the Deterministic Gathering of Mobile Agents

À propos du rassemblement déterministe d'agents mobiles

Sébastien Bouchard

Rapporteurs :
Paola Flocchini, Full Professor, University of Ottawa
Pierre Fraigniaud, Directeur de Recherche CNRS, Université Paris Diderot

Examinateurs :
Shantanu Das, Maître de Conférences, Aix Marseille Université
David Ilcinkas, Chargé de Recherche (HDR), Université de Bordeaux
Maria Potop-Butucaru, Professeure des Universités, Sorbonne Université

Encadrants de thèse :
Yoann Dieudonné, Maître de Conférences, Université de Picardie Jules Verne
Swan Dubois, Maître de Conférences, Sorbonne Université

Directeur de thèse :
Franck Petit, Professeur des Universités, Sorbonne Université

Date de soutenance : 26 septembre 2019

Résumé

Les systèmes distribués sont un modèle théorique capable de représenter une multitude de systèmes bâtis autour de la coopération d'entités autonomes dans le but d'accomplir une tâche commune. Leur champ applicatif est immense, et s'étend de l'informatique ou de la robotique, en modélisant des processus partageant la mémoire d'un ordinateur, des ordinateurs communiquant par envois de messages, ou encore des cohortes de robots, à la compréhension du comportement des animaux sociaux.

Les agents mobiles font partie des entités étudiées dans ce domaine. Ils se distinguent des autres notamment par leur capacité à se déplacer spontanément. L'une des tâches les plus étudiées les mettant en scène est celle du rassemblement. Les agents mobiles sont dispersés dans un environnement inconnu. Aucun d'eux n'a d'informations à propos des autres, ou la capacité de communiquer avec eux, à moins de se trouver au même endroit. Chacun d'eux découvre peu à peu les environs, rencontre d'autres agents et se coordonne avec eux jusqu'à ce que tous soient rassemblés et le détectent. Une fois tous les agents rassemblés, ils peuvent communiquer et se coordonner pour une autre tâche.

Cette thèse s'intéresse à la faisabilité et à l'efficience du rassemblement, en particulier face à deux difficultés majeures: l'asynchronie et l'occurrence de fautes Byzantines. Dans un contexte asynchrone, les agents n'ont aucun contrôle sur leur vitesse, qui peut varier arbitrairement et indépendamment des autres. Se coordonner est alors un défi. Quand une partie des agents subit des fautes Byzantines, on peut considérer ces agents comme malicieux, se fondant parmi les autres (bons) agents pour les induire en erreur et empêcher que le rassemblement ait lieu.

Abstract

Distributed systems are a theoretical model with a huge application field. It can represent a multitude of systems in which several autonomous entities cooperate to achieve a common task. The applications range from computer science related ones like processes sharing memory inside a computer, computers exchanging messages, and cohorts of robots to understanding social animals behavior.

When the entities involved are able to move spontaneously, they are called mobile agents, and one of the most studied problems regarding mobile agents is gathering. The mobile agents are spread in an unknown environment, with no a priori information about the others and without the ability to communicate with other agents, unless colocated. Each of them gradually discovers its surroundings, meets some other agents, coordinates with them, until all agents are gathered and detect it. Once all agents gathered, they can communicate and coordinate for some future task.

This thesis addresses the feasibility and complexity of gathering, in particular when facing two major difficulties: asynchrony and occurrence of Byzantine faults. When tackling the former, the agents have no control over their speed, which can vary arbitrarily and independently from each other. This makes coordination more challenging. When facing the latter, some of the agents are Byzantine, they can be viewed as malicious and using the difficulty to distinguish them from other (good) agents to try to prevent the gathering.

Contents

Chapter 1

Introduction

Contents

1.1 Context and State of the Art

1.1.1 Distributed Systems

This thesis studies systems composed of multiple autonomous computing entities. A same algorithm is executed locally by each entity of the system. Locality refers to the knowledge of the system each agent has a priori: it is partial and only related to what is close (either just spatially or according to other metrics) to the entity itself. Such systems are called distributed.

The area of computer science dedicated to their study is called distributed systems. The challenge in the latter lies in the cooperation between the entities in constrained environments. The motivation for studying such systems is twofold.

First, assigning some tasks to a team of entities instead of only one entity may result in several improvements, even if the replacements are weaker, in terms of complexity and fault tolerance. The time required to achieve the task may be reduced once the latter is split between the entities. Moreover, the more entities there are in the team, the less impact the crash of one entity may have on the success of the task. In particular, a task assigned to only one entity may fail as early as it is subject to any fault.

Furthermore, there are several practical scenarios involving multiple entities for which distributed systems may provide good models. There are instances related to computer science such as processes sharing memory on a same computer, or computers linked by a communication network. Other instances involve robots spread in some area to explore it, or to form successive patterns for entertainment purposes. But such scenarios do not necessarily involve technology. Animals exhibiting social behavior from ants to humans have been cooperating for centuries.

This variety of applications explains the variety of model variants found in the distributed systems literature. One variant differs from another in particular by the nature of the environment (e.g., discrete or continuous) and the abilities of the entities (e.g., communication and perception). A simple change from one variant to another may result in deep modifications of the approaches and results. This motivates research for reductions between model variants [10, 34].

Among this variety of distributed systems, this thesis studies those made of mobile entities i.e., able to spontaneously move in their environment. These systems are sometimes called mobile distributed systems. They mainly model both teams of mobile robots, and software agents, that are mobile pieces of software that travel in a communication network to perform maintenance of its components or to collect data distributed in nodes of the network [75]. The most studied tasks involving mobile entities are pattern formation (i.e., reaching positions which draw some input

pattern) [36, 58, 100], exploration (i.e., visiting every location of the environment or finding some target) [74, 92], gathering (i.e., from distant locations, gathering at a same location), and scattering (i.e., from a same location, spreading in the environment) [15].

1.1.2 Related Research Fields

Before giving more details about the distributed systems of mobile entities and the tasks considered in this thesis, it is worth noticing that other fields address the interactions of several autonomous entities and taking a look at the relationship between them, computer science and distributed systems.

The entities considered in game theory [95] are rational decision makers involved as players in mathematical models called games. The most notorious applications of this field concern economics, but game theory also influences computer science and distributed systems [68]. Interestingly enough, the first mention of the gathering is attributed to a book authored by Thomas Schelling [97]. This book mostly addresses game theory but the author illustrates his point with the task of gathering which appears under the name of rendezvous and praises it as a candidate for epitomizing tacit coordination i.e., without means of communication.

In particular, Thomas Schelling makes use of the following instance. A couple lost each other, without any prior understanding on where to meet if they get separated. They are very likely to think of an "obvious" focal point, where they will meet. The author emphasizes the need not only to predict where the other will go, since the other will go where he predicts the first to go, but answer the following question. "What would I do if I were she wondering what she would do if she were I wondering what I would do if I were she ...?" According to the author, "they must mutually recognize some unique signal that coordinates their expectations of each other".

The latter explanations by Schelling highlight the importance of knowledge and of its sharing. This makes the study of knowledge, epistemology another area related to game theory and distributed systems.

In particular, the requirement from the above instance, for the couple to find each other could be formulated using the notions and model introduced by Fagin and his coauthors in their book [57]. The main of these notions is called "common knowledge".

The authors explain: "When Alice tells Bob "Let us meet tomorrow as usual", the meeting place and hour have to be common knowledge. This means that not only do Alice and Bob have to know the meeting place and hour, but Alice must know that Bob knows to be sure he will give a reasonable answer, and Bob must know that Alice knows that Bob knows for him to be sure that she will correctly understand his answer, and so on ad infinitum."

The focus of Fagin and his coauthors is on understanding the process of reasoning about knowledge in a group and using this knowledge to analyze complicated systems. They introduce a formal semantic model for knowledge, and a language for reasoning about knowledge. The basic idea underlying the model is that of possible worlds. The intuition is that if an entity does not have complete knowledge about the world, it will consider a number of worlds possible. These are its candidates for the way the world actually is.

1.1.3 The Tasks and Model Considered throughout this Thesis

The rest of this introduction focuses on the mobile distributed systems. More precisely, the current section addresses the task considered in this thesis: gathering (described in Section 1.1.3.1). The main features of the model in which the latter is investigated are discussed in Section 1.1.3.2. Section 1.1.3.3 gives some insight about this task by presenting reductions to related tasks whose state of art is presented in Sections 1.1.3.4 to 1.1.3.7.

1.1.3.1 Studied Task: Gathering

Among the aforementioned tasks involving mobile entities, hereafter called mobile agents, this thesis addresses the one known as gathering. It is studied as a good candidate for epitomizing coordination without prior agreement in spite of locality. When considering this task, locality refers in particular to the different (initial) positions of the agents, and their limited range of vision and communication. The mobile agents are spread in an unknown environment, with no a priori information about the others (neither their number, nor their positions ...), and without the ability to communicate which other agents, unless collocated. Each of them can see what is close to it, and has to move to gradually discover its surroundings. Little by little, it meets some other agents, coordinates with them, until all agents are gathered and detect it.

The second requirement, the detection of termination, is not trivial to meet. In some cases, the agents may be unable to declare that the gathering is achieved, the first time this occurs. Some other agents may still be somewhere else, looking for their peers. It is necessary that the algorithm the agents execute ensures that, at some point, this cannot be the case anymore. The mobile agents may have to spread once gathered, if they are not sure that it is the case, and gathering may be achieved several times, before the agents are able to detect it.

1.1.3.2 Key Features of the Model

As mentioned in Section 1.1.1, distributed systems have a wide application field. This explains the great variety of ways gathering is addressed in the literature. Indeed, there are a lot of different model variants generated by the combinations possible to make e.g., by playing on the environment in which the agents are supposed to evolve, or the ability to perceive their environment, communicate with their peers, or leave some traces in the visited locations.

In spite of this variety of assumptions, most approaches share several features. First of all, Alpern [7] explains that they differ from the illustration of Schelling (refer to Section 1.1.2) in that the task is not seen as a one shot tacit attempt to agree on some focal point, which may either fail or succeed. On the contrary, the solutions presented are dynamic: the players keep trying to meet until succeeding, and the region in which the agents evolve is assumed to be symmetric which prevents common focal points to be determined. Formalizing this symmetry assumption is the aim of some articles [6]. In other words, if some focal point exists, gathering is reducible to exploration: it is enough that each agent visits the whole environment to find a focal point which will be unambiguously recognized by its peers.

This symmetry however has a major drawback. If all agents are anonymous i.e., without any kind of identifier, then they may be unable to meet deterministically due to symmetry. This occurs for instance between two agents in a ring shaped network. Since they execute the same deterministic algorithm with the same input, they make the same choices, move to same looking places concurrently, the distance between them does not change, and so on. For this reason, it is often assumed that each agent is given a positive integer called label, which can be viewed as an identifier, an input for the algorithm, allowing the behaviors of any two agents to eventually differ.

Another feature of the model needs to be explained to understand the state of art about the gathering. Indeed, even when all agents move at the same constant speed i.e., in synchronous settings, all agents are not assumed to start executing the algorithm at the same time. They are all assumed to be initially dormant, idle, unaware of their environment and to wake up at some point to start executing the algorithm. In some articles, the event which triggers this waking up is completely external. In others, some agents are woken up by some external signal, but the others can be woken up when some non-dormant agent is close enough to them.

Two kinds of environments are assumed throughout the literature: a continuous one, the plane and a discrete one, the graphs. In the literature, the discrete environment is the most studied one. Sections 1.1.3.5 and 1.1.3.6 focus on the state of the art in graphs while Section 1.1.3.7

addresses the state of art in the plane.

1.1.3.3 Reductions to Rendezvous and Treasure Hunt

Once all agents gathered, they can communicate and coordinate for some future task. For this reason, the task of gathering can be viewed as a building block allowing to perform more complex tasks. However, gathering algorithms are themselves often built upon other building blocks, thanks to reductions to simpler versions of this task.

The particular case of the gathering in which there are precisely two agents is often referred to as rendezvous, although this name designates gathering in some articles and books. When considering this task, detecting an occurrence of the meeting, and declaring that the rendezvous is achieved is not a real issue. Indeed, it is enough for each agent to notice the other one: since their algorithm is designed for rendezvous, there is no other agent they have to wait or look for together.

It is known [77] that a rendezvous algorithm can be built upon to gather an arbitrary number of agents. The strategy to do so consists in "sticking the agents together" whenever they meet. After some agents meet, they can communicate and choose one of them as a leader, and all continue the rendezvous algorithm of this leader, as if there had been no rendezvous. However, it is worth noticing that although this strategy ensures gathering, detecting its occurrence requires some other tool.

In a similar manner, the treasure hunt task can be viewed as a variant of the rendezvous studied to derive results for the latter. The treasure hunt involves one agent and one treasure. It is achieved when the agent discovers the treasure i.e., when it is close enough to it. The treasure can be viewed as a second mobile agent, either waiting or made somehow very slow. Thus, the strategy known as "wait for mummy" [7, 98] builds upon a treasure hunt algorithm to solve the rendezvous as follows. Among the two agents, one waits while the other one executes the treasure hunt algorithm. Unfortunately, it is not that easy for two agents too far away from each other to communicate, without prior agreement, to coordinate and choose which will act as the treasure hunter (or mummy) and which will act as the treasure (or child). Hence, the rendezvous algorithms [48] are often built as a sequence of attempts, such that during each attempt, each agent chooses one of the two roles. Most attempts fail because the two agents choose the same role, but the algorithms ensure some eventual attempt during which the agents play different roles.

Besides the aforementioned lack regarding the detection of an occurrence of the gathering, it should be highlighted that these two reductions are not valid in every model considered in the literature. For instance, if the agents were unable to communicate, it is unlikely that they could use the "stick together" strategy. Moreover, if the two agents performed the attempts mentioned in the previous paragraph at different or even varying speeds, ensuring some eventual attempt performed concurrently, during which they play different roles requires more involved techniques.

The next sections focus on the state of the art for these three tasks, starting with the simplest of them: treasure hunt (Section 1.1.3.4).

1.1.3.4 State of the Art for Treasure Hunt

When designing a treasure hunt algorithm, the goal is often to propose a solution which is not only asymptotically optimal but whose competitive ratio is optimal. The competitive ratio of an algorithm is the ratio of the distance which would be traveled by an optimal algorithm if the agent knew the position of the treasure from the beginning, over the distance which is traveled in the worst case by the algorithm (when the agent does not know this position). It can be thought of as a measurement of the detour implied by the absence of knowledge of the position of the treasure.

An early paper [17] shows that the best competitive ratio for deterministic treasure hunt on a line is 9. This is often referred to as the cow path. The optimal competitive ratio is obtained by a doubling zigzag strategy: go left at distance 1, then right at distance 2, left at distance 4, and so on. In [47] the authors generalized it, considering a model where, in addition to travel length, the cost includes a payment for every turn of the agent. Searching on a line has been generalized to searching from the center of a star [71].

When generalizing to treasure hunt in the plane, it is not asked that the agent reaches the exact position of the treasure (as it is the case on the line), but that it reaches a position whose distance to the treasure is at most some constant, which can be viewed as a range of detection of the agent. It can be proved that treasure hunt in the plane requires to travel a distance belonging to $\Theta(\Delta^2)$, where Δ denotes the initial distance between the agent and the treasure. Indeed, by following a spiral from its original location, the mobile agent can search the disc containing every point at distance at most Δ (without knowing this value). Moreover, unless it searches this disc, there are points it does not see, and in which the treasure can lie. In order to circumvent this bound, some additional information can be initially provided to the agent. In particular, it is shown that a priori knowing a line on which the treasure lies enables to reach a lower complexity than a priori knowing the distance to it [11]. This leads to the idea of searching for a line in the plane [81], or in an arrangement of lines (partition of the plane formed by a collection of lines) [30]. Furthermore, when the target is a point, some work is dedicated to the scenario when finding the treasure means reaching a point p such that the treasure lies on the segment between the initial position of the agent o and p (instead of reaching a point from which the treasure is within some range). For this scenario, the optimal competitive ratio of approximately 17.289 is obtained by following the logarithmic spiral [70, 80].

A possible generalization of treasure hunt consists in considering several pursuers either competing [99] or cooperating to find the target [65]. In the latter settings, spiral search is still at the root of the approaches, except if the memory of the agents is bounded, in which case other approaches are needed [56, 82].

The scenario in which the target is mobile and tries to escape from its pursuers has also been investigated. The tasks consisting in finding it are often referred to as pursuit-evasion games [18, 35] which can be viewed as opposing two players the pursuers and the target. One of the most basic ones is known as the cops and robbers game. The cops win the game if they can move onto the robbers vertex. Cops and robbers move along the graph edges in turns. In Parson's game, the evader is considered as arbitrarily faster than the pursuers. They have to surround it in order to catch it. In these games two main questions are investigated. Given a graph, how many pursuers are needed to catch the evaders regardless of the initial positions? What is the class of graphs with a given number as answer to the previous question?

1.1.3.5 State of the Art for Deterministic Rendezvous in Networks Modeled as Finite Graphs

For deterministic rendezvous in networks modeled as graphs [94], attention concentrates on the study of the feasibility of rendezvous, and on the time required to achieve this task, when feasible. This task has been considered in the literature under two alternative scenarios: *weak* and *strong*. Under the weak scenario [43, 52, 86], agents may meet either at a node or inside an edge. Under the strong scenario [48, 77, 98], they have to meet at a node, and they do not even notice meetings inside an edge. Each of these scenarios is appropriate in different applications. The weak scenario is suitable for physical robots in a network of corridors, while the strong scenario is needed for software agents in computer networks.

Rendezvous algorithms under the strong scenario are known for synchronous agents, where time is slotted in rounds, and in each round each agent can either wait at a node or move to an adjacent node. Thus, in synchronous settings, the strong scenario is implicitly considered. One of earliest algorithms for rendezvous in synchronous settings [48] guarantees a meeting of

the two involved agents after a number of rounds that is polynomial in the size n of the graph, the length $|\ell_{\min}|$ of the shortest of the two labels and the time interval θ between their wake-up times. As an open problem, the authors asked whether it was possible to obtain a polynomial algorithm to this task which would be independent of θ. A positive answer to this question was given independently in two articles [77, 98]. To do so, one of them [98] explicitly builds upon a treasure hunt algorithm.

While these algorithms ensure rendezvous in polynomial duration (i.e., a polynomial number of rounds), they also ensure it at polynomial cost because the cost of a rendezvous protocol in a graph is the number of edges traversed by the agents until they meet and each agent can make at most one edge traversal per round. Note that despite the fact a polynomial duration implies a polynomial cost in this context, the reciprocal is not always true as the agents can have very long waiting periods, sometimes interrupted by a movement. Thus these parameters of cost and time are not always linked to each other. This was highlighted in [89] where the authors studied the trade-offs between cost and time for the deterministic rendezvous task.

More recently, some efforts have been dedicated to analyze the impact on time complexity of rendezvous when in every round the agents are provided some pieces of information by making a query to some device or some oracle [45, 88]. Along with the work aiming at optimizing the parameters of duration and/or cost of rendezvous, some other work have examined the amount of memory required to achieve the task in trees [63, 64] and arbitrary finite graphs [42]. The task has been approached in a fault-prone framework [33], in which the adversary can delay an agent for a finite number of rounds, each time it wants to traverse an edge of the network. Furthermore, some articles assume that the agents are equipped with tokens used to mark nodes [16, 79].

Apart from the synchronous scenario, the academic literature also contains several studies focusing on a scenario in which the agents move at constant speed, which are different from each other, or even move asynchronously. In asynchronous settings, each agent decides to which neighbor it wants to move but the adversary totally controls the walk of each agent and can arbitrarily vary its speed.

The scenario of possibly different fixed speeds of the agents is considered in several articles [78] and in this scenario, only weak rendezvous is ensured. However, it is not known whether strong rendezvous is possible in these settings.

Several authors investigated asynchronous rendezvous in network environments [43, 52, 86]. Under this assumption, rendezvous under the strong scenario cannot be guaranteed even in very simple graphs, and hence the rendezvous requirement is weakened by considering the scenario called *weak* in the present thesis. In the earliest study of this task [86], the authors investigated the cost of rendezvous for both infinite and finite graphs. In the former case, the graph is reduced to the (infinite) line and bounds are given depending on whether the agents know the initial distance between them or not. In the latter case (finite graphs), similar bounds are given for ring shaped networks. They also proposed a rendezvous algorithm for arbitrary graphs provided the agents initially know an upper bound on the size of the graph. This assumption was subsequently removed [43]. However, in these studies, the cost of rendezvous was exponential in the size of the graph. A third article [52] presented the first rendezvous algorithm working for arbitrary finite connected graphs at cost polynomial in the size of the graph and in the length of the shortest label.

1.1.3.6 State of the Art for Gathering in Networks Modeled as Graphs

Rendezvous is much more studied than gathering. This is partly due to the fact that gathering can be obtained thanks to a rendezvous algorithm by applying the "stick together" strategy as described in Section 1.1.3.3. However, the task of gathering has been studied when sticking together is not that easy.

This is the case when the agents are anonymous [50]. However, in this case, as explained in

Section 1.1.3.2, there are initial positions of the agents which are not gatherable i.e., from which no deterministic algorithm can achieve gathering due to symmetry. The authors characterize the gatherable positions and provided two gathering algorithms for every gatherable position.

More precisely, the characterization of gatherable positions relies on the notion of view of a node v in the graph G. Intuitively, this is the view that an agent at v would have if it looked at the whole graph from v. More formally, it is a infinite tree rooted at v and such that the children of any node u in the tree are all its neighbors in G but its father in the tree. This notion well captures symmetry in graphs since two nodes having different views can be thought as not symmetrical.

Apart from this interesting characterization, the two algorithms show an interesting trade-off. The first algorithm relies on the knowledge by every agent of a polynomial upper bound on the number of nodes in the graph and ensures detection of termination. On the contrary, the authors show that no algorithm can ensure gathering with detection of termination without this information and their second algorithm does not require any additional information but does not detect termination. The agents eventually stop but do not know whether there are other agents.

Another case in which the "stick together" strategy cannot be easily applied occurs when some of the agents are Byzantine i.e., prone to Byzantine faults. First introduced at the beginning of the 80s [93], a Byzantine fault is an arbitrary fault occurring in an unpredictable way during the execution of a protocol. Due to its arbitrary nature, such a fault is considered as the worst fault that can occur. Byzantine faults have been extensively studied for "classical" networks i.e., in which the agents are fixed nodes of the graph [14, 85].

When some agents are Byzantine, gathering all agents, including the Byzantine ones, can not be ensured because the Byzantine agents may never be with the other agents, called good, or may declare that the gathering is achieved at any time. Thus, the task known as Byzantine gathering is considered instead. It consists in gathering all good agents, and having them all declare that the gathering is achieved, in the presence of f Byzantine agents.

The latter task is introduced [51] via the following question: *what is the minimum number $\mathcal{M}$ of good agents that guarantees Byzantine gathering in all graphs of size n?* The authors provided several answers to this problem by firstly considering a relaxed variant, in which the Byzantine agents cannot lie about their labels, and then by considering a harsher form in which Byzantine agents can lie about their identities. For the relaxed variant, it is proved that the minimum number $\mathcal{M}$ of good agents that guarantees Byzantine gathering is precisely 1 when the each agent knows both n and f and $f+2$ when each agent knows f only. The proof that both these values are enough, relies on polynomial algorithms using a mechanism of blacklists that are, informally speaking, lists of labels corresponding to agents having exhibited an "inconsistent" behavior. Of course, such blacklists cannot be used when the Byzantine agents can change their labels and in particular steal the identities of good agents. The authors also give, for the harsher form of byzantine gathering, a lower bound of $f+1$ (resp. $f+2$) on $\mathcal{M}$ and a deterministic gathering algorithm requiring at least $2f+1$ (resp. $4f+2$) good agents, when each agent knows both n and f (resp. when it knows f only). Both these algorithms have a huge complexity as they are exponential in n and $\ell_{\max}$, where $\ell_{\max}$ is the largest label of a good agent evolving in the graph.

Some advances are subsequently made [23], via the design of two algorithms, one for each case, that work with a number of good agents that perfectly matches the lower bounds of $f+1$ and $f+2$ shown in the first article. However, these algorithms also suffer from a complexity that is exponential in n and $\ell_{\max}$.

1.1.3.7 State of the Art for Rendezvous and Gathering in the Plane

A good starting point for presenting the state of art of rendezvous and gathering in the plane is paradoxically the article [43] which shows that asynchronous rendezvous (bringing two

asynchronous agents at the exact same point) in the plane is impossible. This leads the authors to introducing the task of approximate rendezvous (also known as approach), which considers two agents with some constant range of vision and consists in bringing them not to the same position but within each other's range. This article also provides the first algorithm ensuring asynchronous approach of two agents in the plane. To provide the latter, the authors discretize the plane into an infinite graph: the task of approach is reduced to that of rendezvous in an infinite graph. Unfortunately, the cost of their algorithm is super-exponential.

The rest of the state of art in the plane can be viewed as successive attempts to improve on this result. The same reduction to asynchronous rendezvous in infinite graphs such as the infinite grid i.e., the infinite graph in which each node has degree 4, is used. However, no algorithm achieving approach, with a cost polynomial in the initial distance Δ between the agents and the length of the smallest label, in asynchronous settings, without providing additional information to the agents, has been proposed yet.

A first article restricts asynchrony to a model in which each agent is given a constant speed at which it travels during the whole execution [49]. It is also worth mentioning an algorithm whose cost is polynomial in Δ but which relies on the assumption that the agents are location aware [13, 40]: each agent knows the coordinates of its initial position in some common coordinate system. This hypothesis enables the authors not only to reach polynomiality but also a fine-grained complexity since their cost belongs to $O(\Delta^2 \log(\Delta))$.

1.1.4 Another Model: Look-Compute-Move Robots

The previous section focuses on the literature which, because it addresses the gathering task in a similar model, is the closest to this thesis. However, mobile distributed systems have been considered in other models and other tasks have been studied. A comprehensive book about mobile distributed systems, the various model variants in which they are considered, the results in gathering and several other tasks appeared shortly before the writing of this thesis [59]. The authors are among the leading figures of this area of research. The model which is the most investigated both in the latter book, and in literature is called Look-Compute-Move robots [58, 100].

In this model, the execution of its algorithm by each robot can be viewed as a cycle of three phases namely Look, Compute and Move. During the first one, the robot observes its environment, taking a snapshot. Then, it executes its algorithm to compute its next position. Lastly, it moves to the chosen location. Even though this three phases cycle does not really differ from the model described in this thesis and can be viewed as a low level description of the execution of their algorithm by the mobile agents assumed in this thesis, it is often associated with a set of abilities different from the one considered in this thesis. More precisely, these Look-Compute-Move robots are often assumed to be anonymous i.e., have no kind of identifier and oblivious i.e., forget previous observations and computations, but to take snapshots of the whole environment, including the positions of the others. In such a context locality refers to the possibly different coordinate systems the robots have. Some agreement is often reached thanks to geometric properties of the set of positions, independent from the local coordinate systems such as the smallest enclosing circle [31].

Since the introduction of this model [100], pattern formation in the plane has been the most studied task in these settings [36, 58, 103]. The input for this task is common to all robots: the pattern to draw. The task is considered completed whenever the robots have reached positions whose relative positions are the same as those of the points in the input pattern, regardless of enlargements, rotations...

Failing to give as much insight about the literature on Look-Compute-Move robots as the recent book [59], this section aims at giving pointers to some of the reasearch directions of the state of the art related to the gathering of Look-Compute-Move robots. The synchronous gathering of robots has been addressed assuming the occurrence of several kinds of faults

(crash, Byzantine faults, and self-stabilization) [2, 46, 53]. Asynchronous rendezvous and gathering of robots have been considered both in graphs [44, 67] and in the plane [37]. Another article [69] studies the impact of the differences between the coordinate systems of the robots (somehow the impact of locality) on the feasibility of gathering in synchronous and asynchronous settings. Several other studies of the latter case investigate different sets of assumptions including inaccuracy in perception and movement [39], restricted visibility [60], occurrence of faults [2, 41, 46], or pay attention to avoiding collisions [91]. In the Look-Compute-Move model, since robots take snapshots of the whole system, the set of positions of the agents can be viewed as a memory in which they write by moving and read with these snapshots. This in particular permits to establish an equivalence between gathering in a variant of the Look-Compute-Move model, and consensus between processes sharing memory [4]. The latter task is the most studied in distributed systems. Each process is given a binary input and they have to agree in finite time on one of the inputs.

1.2 Contributions

This section presents the contributions of this thesis, lists the publications in which they resulted, and mentions some results obtained when the author was a Master student.

Strong rendezvous in finite graphs. As explained in Section 1.1.3.5, rendezvous algorithms under the strong scenario are known for synchronous agents in finite graphs [48, 77, 98]. For asynchronous agents, rendezvous under the strong scenario is impossible even in the two-node graph, and hence only algorithms under the weak scenario were constructed [52]. Chapter 3 shows that rendezvous under the strong scenario is possible in finite graphs for agents with asynchrony restricted in the following way: agents have the same measure of time but the adversary can impose, for each agent and each edge, the speed of traversing this edge by this agent. The speeds may be different for different edges and different agents but all traversals of a given edge by a given agent have to be at the same imposed speed. A deterministic rendezvous algorithm is presented for such agents, working in time polynomial in the size of the graph, in the length of the smaller label, and in the largest edge traversal duration.

Asynchronous approach in the plane. The literature considering the problem of asynchronous approach [13, 40, 43, 49] has left open the following question. Let Δ and $|\ell_{\min}|$ be the initial distance separating the agents and the length of (the binary representation of) the shortest label, respectively. Assuming that Δ and $|\ell_{\min}|$ are unknown to the agents, does there exist a deterministic approach algorithm always working at a cost that is polynomial in Δ and $|\ell_{\min}|$? Chapter 4 provides a positive answer to this question. More precisely, it shows an asynchronous weak rendezvous algorithm in the infinite grid whose cost is polynomial in the initial Manhattan distance D between the agents and in $|\ell_{\min}|$. The existence of the latter, thanks to a reduction from asynchronous approach to asynchronous weak rendezvous in the infinite grid, implies the existence of the desired algorithm.

Byzantine gathering in finite graphs. Chapter 5 addresses Byzantine gathering in finite graphs, and in synchronous settings. In literature, the existing algorithms for this problem all have an exponential time complexity in the number n of nodes and the labels of the good agents. The contributions presented in Chapter 5 include a deterministic algorithm of polynomial complexity in n and $|\ell_{\min}|$, provided the number of good agents is at least some prescribed value quadratic in the number of Byzantine agents. This requires that all good agents are initially given a same advice that can be coded in $O(\log\log\log n)$ bits: this size is shown to be of optimal order of magnitude to obtain the aforementioned result.

Treasure hunt in the plane with angular hints. Chapter 6 addresses the problem of treasure hunt in the plane. More precisely, since it is well known that without any hint the optimal (worst case) cost belongs to $\Theta(\Delta^2)$, with Δ the initial distance between the agent and the treasure, Chapter 6 investigates the question of how some additional information about the position of the agent can permit to lower the cost of finding the treasure, using a deterministic algorithm. This additional information takes the form of hints obtained in the beginning and after each move. Each hint consists of a positive angle smaller than 2π whose vertex is at the current position of the agent and within which the treasure is contained.

Three cases are studied depending on the measures of the angles given as hints. If all angles are at most π, then the cost can be lowered to $O(\Delta)$, which is optimal. If all angles are at most β, where $\beta < 2\pi$ is a constant unknown to the agent, then the cost is at most $O(\Delta^{2-\epsilon})$, for some $\epsilon > 0$. For both these positive results, Chapter 6 presents deterministic algorithms achieving the above costs. Finally, if angles given as hints can be arbitrary, smaller than 2π, then it is proved that cost $\Theta(\Delta^2)$ cannot be beaten.

Publications. The contributions of Chapters 4, 5, and 6 have been published in the proceedings of *International Symposium on Distributed Computing 2017* [19], *International Colloquium on Automata, Languages, and Programming 2018* [25], and *International Symposium on Algorithms and Computation 2018* [26] respectively. The strong rendezvous algorithm of Chapter 3 has been published in *Information Processing Letters* [27].

A complete version of the results regarding asynchronous approach of Chapter 4 has been published in *Distributed Computing* [21].

Lastly, the results of chapters 4 and 6 have also given rise to publications in the proceedings of *rencontres francophones sur les Aspects Algorithmiques des Telecommunications*, in 2018 and 2019 respectively [20, 28]. The latter was nominated for the Best Student Paper Award.

Preliminary results. Regarding Byzantine gathering in finite graphs, an article [23] mentioned in the state of the art presents results obtained while the author was a Master student. This contribution completes the partial answer brought by the article which introduced Byzantine gathering [51] to the question it asks and leaves partially open: *what is the minimum number $\mathcal{M}$ of good agents that guarantees Byzantine gathering in all graphs of size n?* Thus, Chapter 5 is a continuation of this work, trying to improve on its lacks.

These results have been published in the proceedings of the *International Colloquium on Structural Information and Communication Complexity 2015* [22], and in *Distributed Computing* [23], and earned the Best Student Paper Award of *rencontres francophones sur les Aspects Algorithmiques des Télécommunications 2018* [24].

Chapter 2

Model

Contents

Although mobile agents are considered throughout this thesis, each contribution considers a specific model variant by playing in particular on the speeds of the agents and the space in which they move.

Chapters 3, 4 and 5 in particular differ by the speeds at which the agents move. In the latter chapter, the agents are synchronous i.e., time is slotted in rounds, and in each round each mobile agent executing the algorithm either makes one move or waits. In Chapter 3, for each agent A, and for each edge e, there is a duration t such that crossing e always takes t rounds to A. Lastly, in Chapter 4, the agents are assumed to move asynchronously. This in particular means that a given agent can spend different amounts of time to cross a same edge at different times.

Regarding the space in which the agents move, Chapters 3 and 5 consider arbitrary finite graphs, while Chapters 4 and 6 assume that the agents move in the Euclidean plane.

Furthermore, Chapter 5 addresses a fault-tolerant variant of the gathering, in which some agents are subject to Byzantine faults. Lastly, Chapters 5 and 6 assume that the agents are provided some additional information, in two different forms.

This chapter tries to state the features shared by all contributions in a generic way, while permitting each chapter to easily derive the precise model variant it considers. However, although rather generic, these definitions are not claimed to be suitable for all studies of mobile entities e.g., the Look-Compute-Move robots. They are written with the only goal to define the model shared by all contributions of this thesis. The statement of this shared model starts with the following definitions of mobile agent and mobile agent algorithm.

Definition 2.1 (Mobile agent)**.** *A mobile agent is a computing entity with unbounded memory able to spontaneously wait, move, observe its environment and communicate with its peers.*

Definition 2.2 (Mobile agent algorithm)**.** *A mobile agent algorithm is a sequence of instructions which can be either classical computing ones, or instructions making use of the abilities of a mobile agent i.e., waiting some prescribed amount of time, moving in some prescribed direction, or assigning some information obtained through environment observation or communication to some variable. It requires one positive integer input called label.*

For simplicity, since it focuses on mobile agents algorithms, in this thesis, mobile agent algorithms are often referred to as algorithms. Definition 2.2 does not specify the input and

effects of the instructions which can compose a mobile agent algorithm. These explanations are given in Section 2.2, which explains how the execution of an algorithm by a distributed system of mobile agents occurs. To this end, it relies on the definitions of Section 2.1 related to the environment in which the agents move. Section 2.3 builds on the two previous ones to state the task of gathering and explain how the efficiency of a mobile agent algorithm is measured. Lastly, Section 2.4 describes some notations used throughout the rest of the thesis. Among them, one is needed in the definitions of the following sections. Throughout this chapter, the cardinality of any set S is denoted by $|S|$.

2.1 The Environment of the Mobile Agents

This section is dedicated to defining the elements which compose any environment in which the mobile agents move and interact, and stating some of the environments considered in this thesis. The main definitions of this section are those of environment and model variant, in Section 2.1.3. They rely on the definitions of timeline and space defined in Sections 2.1.1 and 2.1.2, respectively.

2.1.1 Modeling Time

In synchronous settings, time is regarded as discrete while in asynchronous ones, it is regarded as continuous. To define the flow of time independently of this, the following definition of a timeline is needed.

Definition 2.3 (Timeline and time)**.** *A timeline is a linearly bi-ordered group $(T, +)$. Every element of T is called time.*

Two different timelines are defined: the discrete and continuous ones, on which rely in particular the synchronous and asynchronous settings, respectively.

Definition 2.4 (Discrete timeline)**.** *The discrete timeline consists of the set $\mathbb{Z}$ of the integers, together with its addition.*

Definition 2.5 (Continuous timeline)**.** *The continuous timeline consists of the set $\mathbb{R}$ of the real numbers, together with its addition.*

In synchronous settings, the timeline which is used is the discrete one. In these settings, instead of time, the terms round or number of rounds are used to designate the elements of the timeline.

2.1.2 Modeling Space

Besides the timeline, defining the environment of the mobile agents requires to state the set of positions between which they move, the destinations available from each of them, etc. This is the subject of the following definition.

Definition 2.6 (Space, position, direction, destination and range)**.** *A space s is defined by a 6-tuple (P, Z, Q, d, c, r). Its three first components P, Z, and $Q \subseteq P \times Z$ are sets. The last component, r, is a reflexive and symmetric relation over P. Moreover, d and c are two applications from Q, to the sets P and $\mathbb{R}$, respectively. The elements of P and Z are called positions and directions, respectively. For every position p_1, the sets $\{p_2 \in P | p_1 r p_2\}$ and $\{z \in Z | (p_1, z) \in Q\}$ are called the range of p_1 and the directions available at this position. Lastly, if $d(p_1, b_1) = p_2$, then p_2 is called the destination of the move from p_1 with direction b_1.*

The previous definition relies on several elements whose utility has not been explained yet. Each of them is needed to explain how each mobile agent interacts with any space. In particular, they represent, given a current position, the set of positions to which it is possible to move, the set of positions such that it is possible to communicate which other agents in them, the cost (may model some fuel, and allows to compare the efficiency of two algorithms) of a move. This is further explained in Section 2.2, in particular by Definitions 2.20 and 2.23.

The following definitions describe the spaces considered in this thesis. The first presented ones are discrete: the graphs.

Definition 2.7 (Port labeled graph)**.** *A port labeled graph is a space* (P, Z, Q, d, c, r) *verifying the following properties. Denote by* E *the set* $\{(p_1, p_2) | (p_1 \in P) \cap (\exists z \in Z, d(p_1, z) = p_2)\}$*. The couple* (P, E) *is a simple, loopless, undirected, and connected graph (classical meaning). The relation* r *is defined by the set* $\{(u, u) | u \in V\}$*. For every* $(u, z) \in Q$*,* $c(u, z) = 1$*.*

In other words, a port labeled graph is a space verifying notably: its positions are the nodes of a simple, undirected, and connected graph and it is possible to move from a position to another if and only there is an edge between the corresponding nodes. When considering a port labeled graph, the vocabulary related to classical graphs (e.g., nodes, edges, degree) is used to ease various statements. For instance, every position is also called node or vertex. Every pair of positions $\{u, v\}$ such that there exists $z \in Z$ such that $d(u, z) = v$ is called edge, and for every position u, the number of elements z of Z such that $(u, z) \in Q$ is called the degree of u.

Furthermore, if there exist u, v, and z_1 such that $d(u, z_1) = v$, then z_1 is called the port label of edge $\{u, v\}$ at u, and the set of all $z_2 \in Z$ such that $(u, z_2) \in Q$, is called the set of port labels at u.

Definition 2.8 (Finite graph)**.** *Every finite graph is a port labeled graph whose set of vertices is finite and whose set of directions is* $\mathbb{N}$*. Moreover, the set of the port labels at every node* v *with degree* d *is* $\{0, \ldots, d-1\}$*.*

Finite graphs are just a subset of the port labeled graphs, whose set of vertices is finite, and such that each edge incident to a node is given a distinct port label, also called port number, between 0 and $d-1$ with d the degree of the node. Figure 2.1 depicts an example of one finite graph.

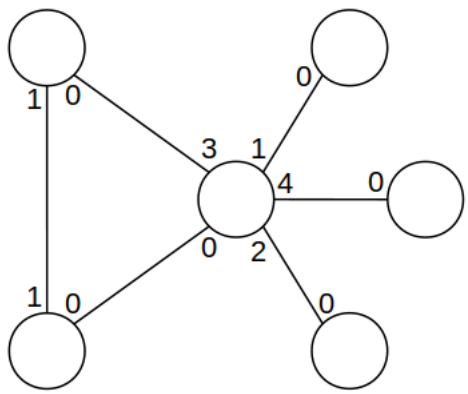


Figure 2.1: Representation of one finite graph. Circles, lines and integers represent the vertices, the edges, and the port numbers, respectively.

Definition 2.9 (Infinite grid)**.** *The infinite grid is the port labeled graph meeting the following conditions. Its set of positions is infinite, every node has degree 4, and the set of labels at every node is* $\{N, E, S, W\}$*. For every edge* $\{u, v\}$*, if its port label at* u *is* N*,* E*,* S*, or* W*, then its port label at* v *is* S*,* W*,* N*, or* E*, respectively. Furthermore, for every vertex* u*, every* $z_1 \in \{W, E\}$*, and every* $z_2 \in \{N, S\}$*,* $d(d(u, z_1), z_2) = d(d(u, z_2), z_1)$*.*

In other words, the infinite grid is one of the port labeled graphs with an infinite number of nodes such that each node has four neighbors, one per cardinal point. Figure 2.2 depicts one part of an infinite grid. The infinite grid is symmetric i.e., for each of its nodes, its neighborhood is as represented by Figure 2.2.

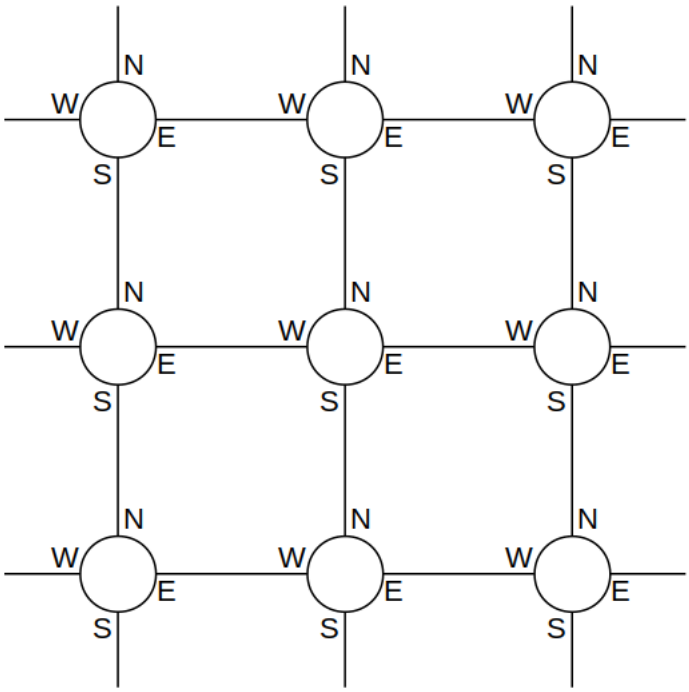


Figure 2.2: Representation of a part of the infinite grid. Circles, lines and letters represent the vertices, the edges, and the port labels, respectively.

The Euclidean plane is the other kind of space considered in this thesis. The main difference between the Euclidean plane and the graphs is related to the set of destinations available at a given position. While in graphs, this set is restricted to the neighbors of the current position, in the Euclidean plane, all positions are possible destinations. This space is defined by the following definition and the latter is illustrated by Figure 2.3.

Definition 2.10 (Euclidean plane)**.** *The Euclidean plane is the space* (P, Z, Q, d, c, r) *defined as follows. Its sets of positions and directions are the set of points in the classical Euclidean plane i.e.,* $\mathbb{R}^2$ *and* $\mathbb{R} \times (-\pi, \pi]$*, respectively. Let* p_1 *and* p_2 *be two elements of* $\mathbb{R}^2$*. Let* (ρ, θ) *be the polar coordinates of* p_2 *in the system centered at* p_1*. The set* Q *is equal to* $P \times Z$ *and* $p_1 r p_2$ *if and only if the Euclidean distance between them is at most 1. The image by* d *and* c *of* (p_1, ρ, θ) *are* p_2 *and* ρ*, respectively.*

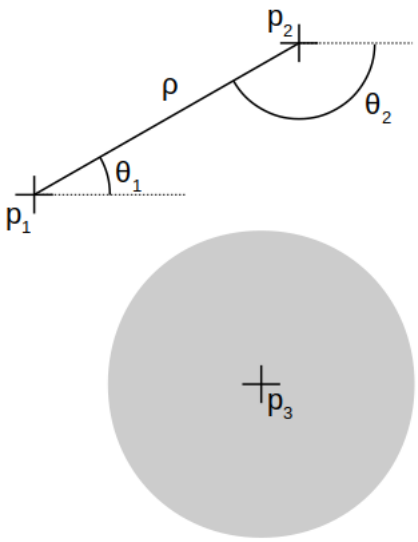


Figure 2.3: Illustration of Definition 2.10. The gray area represents the closed disc centered at p_3 with radius 1 which is the set of positions p such that $p_3 r p$. The positions p_1 and p_2 depicted are such that $d(p_1, \rho, \theta_1) = p_2$, $d(p_2, \rho, \theta_2) = p_1$, $c(p_1, \rho, \theta_1) = \rho$, and $c(p_2, \rho, \theta_2) = \rho$.

2.1.3 Defining the Whole Environment

In many studies of distributed systems, the abstract notion of an adversary is used to define a part of the environment. It encompasses all elements that the algorithm cannot control. The adversary it said to "choose" some features among sets of candidates. For instance, among a set of spaces, it chooses the one in which the agents move and interact, among all the positions of this space, it chooses their initial positions, and for every move, it chooses how much time is needed to complete it. Thus, an algorithm achieves a task if and only if whichever the choices of the adversary, the execution of the algorithm in the environment resulting of these choices indeed verifies the specifications of the task. The adversary is seen as trying to prevent (by its choices) the task or at least to minimize the efficiency of the algorithm. Often, some constraints are placed on the sets among which it chooses, it order to consider weaker adversaries and thus environments in which achieving the desired task is easier.

Although several studies formally define this adversary, and this could be done in this thesis too, another approach is preferred. The goal is to avoid this idea of choice left to an abstract entity. Instead, the environment is defined as a tuple including the timeline and space considered, a set of distinct identifiers called labels for the agents, their initial positions, another element related to the time at which each agent starts executing the algorithm, and a last element representing the duration of each move. A model variant is a set of environments, and an algorithm achieves a given task in a model variant if for every environment of the model variant, the execution of the algorithm in this environment verifies the specifications of the task. Somehow, a model variant is the set of environments among which the adversary chooses. Hence, when viewing the model this way, instead of studying a given adversary or given settings, one considers a given model variant. For instance, synchronous settings are presented as the model variant composed of all environments in which the timeline is discrete and the duration of each move is 1. However, this is just a matter of presentation.

The derivation of this model into the models considered in each contribution consists in

adding some elements to the environment, and stating the model variant studied.

The following definitions introduce several elements whose role can only be explained when linking the environment with the mobile agents. Since the mobile agents are at the heart of this model, all features of the environment only make sense when associated with them. Thus, further explanations about these elements are given in Section 2.2.

Definition 2.11 (Environment)**.** *An environment is a 6-tuple* (s, T, L, i, w, g) *such that:*

- $s = (P, Z, Q, d, c, r)$ *is a space.*
- T *is a timeline.*
- L *is a set of positive integers.*
- i *is an application from* L *to* P*.*
- w *is an application from* L *to* T*.*
- g *is an application from* $T \times Q \times L$ *to the set* $T_{>0}$ *of the positive elements of* T*.*

Based on the previous definition, it is possible to propose a formal definition for a model variant, as a set of environments.

Definition 2.12 (Model variant)**.** *A model variant is a set of environments.*

2.2 Execution of an Algorithm by a Distributed System of Mobile Agents

Two kinds of elements are needed to define the execution of an algorithm by mobile agents. On one hand, there are the theatre stage, the actors, and the play i.e., the environment, the mobile agents, and the algorithm they execute. On the other hand, there are the deterministic rules governing the effects of the instructions of the algorithm on the environment and the mobile agents. Given the environment, the algorithm, and these rules, it is possible to determine the state of the system at any time.

This section starts with the formal definition of the execution of an algorithm by mobile agents. However, this definition only includes the elements of the first kinds.

The set of rules depends in particular on the environment considered. Sections 2.2.1 and 2.2.2 describe the rules which apply in the environment described by Definition 2.11. Even though other chapters derive the present model, and change some of these rules, the environments considered are always based on those of Definition 2.11, and most of the rules keep applying in the other chapters. The nature of the changes is specified in the corresponding chapters.

Precisely, Section 2.2.1 focuses on the rules determining the initial state of the mobile agents. Section 2.2.2 lists the abilities of the mobile agents and states the rules which apply whenever an agent is instructed to use them (e.g., input and effects).

Definition 2.13 (Execution of an algorithm by mobile agents)**.** *An execution is a couple* $(e, \mathcal{A})$ *such that:*

- e *is an environment.*
- $\mathcal{A}$ *is a mobile agent algorithm.*

Remark that "execution" is thus used to designate both the local execution by one mobile agent of the sequence of instructions which compose the algorithm, and a more global view of the system composed of mobile agents all locally executing the algorithm). When using this word, it is made clear from the context which meaning it is given.

2.2.1 Initialization

The first definition explains which elements of an environment determine the initial positions of the mobile agents, and the positive integers which are used as their labels i.e., input of the algorithm.

Definition 2.14 (Link between environment and mobile agents: initial positions and labels)**.** *Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution with $s = (P, Z, Q, d, c, r)$. There are $|L|$ mobile agents spread in s. Each of them is assigned a distinct label ℓ of L, and executes $\mathcal{A}$ with its label as input parameter. For each mobile agent with label ℓ, the first position it occupies is $i(\ell) \in P$ and is called its initial position.*

The mobile agents are not assumed to start executing the algorithm all at the same time. This is even assumed to be partly controlled by the adversary. This contributes to the difficulty to coordinate the agents. More precisely, the agents are said to be initially dormant and to wake up at the time when they start executing the algorithm. The waking up is determined both by the adversary and the progress of the execution. This is formally explained by the two following definitions.

Definition 2.15 (Dormant mobile agent)**.** *Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution with $s = (P, Z, Q, d, c, r)$. For every $\ell \in L$, the mobile agent with label ℓ is initially dormant. While dormant it does nothing and in particular does not execute $\mathcal{A}$ i.e., it learns nothing, can neither communicate, nor wait, nor move.*

Definition 2.16 (Waking up)**.** *Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution with $s = (P, Z, Q, d, c, r)$, and A be any non-dormant mobile agent whose label is denoted by ℓ. The time when A stops being dormant is called its waking up and is determined as follows.*

If there is no time $t < w(\ell)$ at which some non-dormant mobile agent is at some position $p \in P$ within range of the initial position $i(\ell) \in P$ of A (i.e., $i(\ell)rp$), then the waking up of the latter agent is $w(\ell) \in T$. In this case, A is said to be woken up by the adversary at time $w(\ell)$. Otherwise, the waking up of A is the earliest time t at which some non-dormant mobile agent B is at some position p within range of $i(\ell)$. In this second case, A is said to be woken up at t by B.

2.2.2 Progress of the Execution: Abilities of the Mobile Agents

After waking up, each mobile agent is able in particular to wait and move. The inputs, progress and consequences of these two actions are developed by Definitions 2.19 and 2.20. However, the two next definitions are related to more basic rules: the flow of time between the executions of two instructions by a mobile agent, and the abilities of computing and learning.

Definition 2.17 (Sequence of instructions)**.** *Let e be any execution. The local execution of each instruction by any mobile agent is assumed to have a starting and a completion time, the latter being at least the former. Moreover, the starting time of every instruction is the same as the completion time of the previous instruction, if any. Otherwise, it is the waking up time.*

Definition 2.18 (Computing and learning)**.** *Every mobile agent is equipped with an unbounded memory, and remembers at any time, every instruction executed as well as every information learnt since its waking up.*

The assumptions that the computing instructions take no time, and the agents are equipped with an unbounded memory in Definition 2.18 stress the will to focus on the moves of the agents, their durations and costs. This is stated in Section 2.3.

Definition 2.19 (Waiting)**.** *Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution with $s = (P, Z, Q, d, c, r)$, and A be any mobile agent which is at time t at position p. Unless it is dormant, moving or already waiting, A can be instructed by $\mathcal{A}$ to wait. This action requires an input $x \in T_{>0}$: the amount of time to wait. The execution of this action ends up at time $t + x$. At every time in the interval $[t, t + x]$, A is at p, and at every time of $[t, t + x)$, A is waiting.*

The two next definitions state the rules governing the moves of every mobile agent. However, a part of these rules changes in the specific environments considered in the different contributions of this thesis. Hence, Definition 2.20 states the rules which apply in all chapters of the thesis while Definition 2.21 states some others only applied in Chapters 3 and 5 (this is further explained in the introduction of each chapter).

Definition 2.20 (General rules regarding moving)**.** *Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution with $s = (P, Z, Q, d, c, r)$, and A be any mobile agent with label $\ell \in L$ which at time t, is at position p. Unless it is dormant, waiting or already moving, A can be instructed by $\mathcal{A}$ to move at t. This requires as input one of the directions available at p. Moving from p with direction $z \in Z$ allows to reach the destination $d(p, z)$, and is said to have cost $c(p, z)$.*

Definition 2.21 (Specific rules regarding moving)**.** *Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution with $s = (P, Z, Q, d, c, r)$, and A be any mobile agent with label ℓ which at time t, is at position p and starts to move with direction $z \in Z$. The completion time of the move is $t + g(t, (p, z), \ell)$. More precisely, at every time of $[t, t + g(t, (p, z), \ell))$, A is moving, and at $t + g(t, (p, z), \ell)$ it is at position $d(p, z)$.*

The notion of cost of the move presented in the previous definitions is introduced for the sake of Section 2.3. In order to move, every mobile agent needs to know at least a subset of the directions available at its current position. This is the subject of the following definition.

Definition 2.22 (Learning the directions available)**.** *When waking up, every mobile agent is at its initial position and learns the set of directions available at this position. Moreover, at the completion time of every move, every mobile agent learns the set of directions available at the destination. Hence, at every time when it can be instructed to move, each mobile agent knows the directions available at its current position.*

Lastly, each mobile agent can communicate with its peers. Explaining when this is possible, and how this occurs is the purpose of the next definitions.

Definition 2.23 (Communicating with its peers)**.** *Communicating with its peers involves two actions: updating the information to be transmitted, and actually communicating i.e., transmitting and receiving.*

Definition 2.24 (Updating the information to be transmitted)**.** *This action can be performed only once per time by every mobile agent, if it is neither dormant, nor waiting nor moving.*

Definition 2.25 (Transmitting and receiving information previously set to be transmitted)**.** *Unlike updating the information to be transmitted, this action is not an instruction of a mobile agent algorithm. It is automatically performed by any mobile agent whenever it is neither dormant nor moving. Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution with $s = (P, Z, Q, d, c, r)$, and A be any mobile agent which at time t_1, is at position p_1, neither dormant nor moving. Let $t_2 \leq t_1$ be the latest time at which A updated the information to be transmited and x be the information assigned by doing so, if any. Otherwise, let x denote some special marker meaning that it has no information to transmit. At t, agent A transmits x, and for every peer with label ℓ which at t transmits information y at a position p_2 such that $(p_1 r p_2)$, A receives and learns the couple (ℓ, y).*

Remark that the previous definition in particular means that no mobile agent can communicate while moving. For instance, two mobile agents crossing the same edge of some port labeled graph in opposite directions would not communicate, and not notice this event. Moreover, since r is a symmetric relation, there is no couple of agents (A, B) such that at some time, A transmits to B but does not receive from B.

2.3 Tasks Specifications and Efficiency of an Algorithm

Several tasks are considered in this thesis: gathering, rendezvous and treasure hunt under different scenarios. Apart from gathering which is defined in this section, each of these tasks is specified at the beginning of the chapter which addresses it.

Definition 2.26 (Gathering)**.** *Let M be any model variant, and $\mathcal{A}$ be any mobile agent algorithm. Algorithm $\mathcal{A}$ achieves gathering in model variant M if and only if for every environment (s, T, L, i, w, g) of M, in the execution $((s, T, L, i, w, g), \mathcal{A})$, there exists a time $t \in T$ at which, all mobile agents are gathered at a same position and declare that the gathering is achieved.*

The difficulty of achieving gathering and the other tasks presented in the subsequent chapters depends on the cardinality of the model variant considered. Considering a model variant in which some feature (e.g., space, set of labels, initial positions) is fixed makes achieving the task simpler. Moreover, imposing no constraint, for instance, on the sets of labels, allows to intuitively refer to the uncertainty on this feature as if some adversary chose it. One approach in distributed systems is thus to look for some universal algorithm i.e., achieving the desired task in an unrestricted model variant (i.e., containing all possible environments).

Throughout this thesis, two measures of the efficiency of the mobile agent algorithms are used: its cost and its duration. Their definitions are based on the notion of completion of the execution by a mobile agent of an algorithm, that is the earliest moment when the mobile agent has no more instruction of the algorithm to execute i.e., when it has executed all its instructions. These measures can be viewed as summaries of the global execution.

Definition 2.27 (Cost of the execution of a mobile agent algorithm)**.** *Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution. Let $t \in T$ be the latest time at which one mobile agent completes its execution of $\mathcal{A}$. The cost of $((s, T, L, i, w, g), \mathcal{A})$ is defined as the maximum over the set of mobile agents of the sum of the costs of its moves achieved at t at the latest.*

Definition 2.28 (Duration of the execution of a mobile agent algorithm)**.** *Let $((s, T, L, i, w, g), \mathcal{A})$ be any execution. Let $t_1 \in T$ be the earliest time at which one mobile agent is non-dormant, and $t_2 \in T$ be the latest time at which one mobile agent completes its execution of $\mathcal{A}$. The duration (also called time complexity) of $((s, T, L, i, w, g), \mathcal{A})$ is defined as $t_2 - t_1$.*

2.4 Notations

The notation introduced at the beginning of this chapter to designate the cardinality of a set is used troughout this thesis. For every set S, $|S|$ is used to denote its cardinality.

The thesis also considers several sequences, among which binary strings. For every sequence s, $|s|$ denotes the length of s, its elements are indexed from 1 to $|s|$, and $s[i]$ with $1 \leq i \leq |s|$ denotes the element with index i in s. In particular, the labels of the mobile agents, which are positive integers, are regarded as binary strings: their binary representations. Hence, for every label ℓ, $|\ell|$ denotes its length and $\ell[i]$ its i-th bit ($1 \leq i \leq |\ell|$).

Lastly, for every environment (s, T, L, i, w, g), $\ell_{\min}$ denotes the smallest label i.e., the smallest element of L.

Chapter 3

Strong Rendezvous in Finite Graphs

Contents

3.1 Introduction

The task of rendezvous in finite graphs has been considered in the literature under two alternative scenarios: *weak* and *strong*. Under the weak scenario, agents may meet either at a node or inside an edge. Under the strong scenario, they have to meet at a node, and they do not even notice meetings inside an edge. Each of these scenarios is appropriate in different applications. The weak scenario is suitable for physical robots in a network of corridors, while the strong scenario is needed for software agents in computer networks. Definitions 3.1, and 3.2 formally specify these two tasks.

Definition 3.1 (Strong rendezvous)**.** *Let M be any model variant (cf. Definition 2.12), and $\mathcal{A}$ be any mobile agent algorithm. Algorithm $\mathcal{A}$ achieves strong rendezvous in M if and only if for every environment (s, T, L, i, w, g) of M such that $|L| = 2$, the execution $((s, T, L, i, w, g), \mathcal{A})$ (cf. Definition 2.13) meets the following condition. There exists a time at which the two mobile agents are gathered at a same position.*

Definition 3.2 (Weak rendezvous)**.** *Let M be any model variant (cf. Definition 2.12), and $\mathcal{A}$ be any mobile agent algorithm. Algorithm $\mathcal{A}$ achieves weak rendezvous in model variant M if and only if for every environment (s, T, L, i, w, g) of M such that $|L| = 2$, the execution $((s, T, L, i, w, g), \mathcal{A})$ (cf. Definition 2.13) verifies at least one of the three following propositions:*

- *there is a time at which the two mobile agents are at the same position.*
- *there are two positions p_1 and p_2, and a time at which one of the two mobile agents is moving from p_1 to p_2 while the other one is moving from p_2 to p_1.*
- *there are two positions p_1 and p_2, and four times $t_1 < t_2 < t_3 < t_4$ such that a first mobile agent starts moving from p_1 to p_2 at t_1 and finishes at t_4, and the second mobile agent starts moving from p_1 to p_2 too but at t_2 and finishes at t_3.*

3.1.1 Related Work

As explained in Section 1.1.3.5, rendezvous algorithms under the strong scenario are known for synchronous settings [48, 77, 98]. Several authors investigated asynchronous rendezvous [43, 52, 86]. Under this assumption, rendezvous under the strong scenario cannot be guaranteed even in very simple graphs, and hence the rendezvous requirement was weakened by considering the scenario called *weak* in the present thesis. In particular, the main result of [52] is an asynchronous weak rendezvous algorithm working in an arbitrary finite graph at cost polynomial in the size of the graph and in the logarithm of the smaller label.

3.1.2 Contribution

However, due to the fact that the strong scenario is appropriate for software agents in computer networks, and that such agents are rarely synchronous, it is important to design rendezvous algorithms under the strong scenario, restricting the asynchrony of the agents as little as possible. This is the aim of this chapter.

The scenario of possibly different fixed speeds of the agents is considered in several articles [49, 78] and in this scenario, a polynomial approach algorithm and a polynomial weak rendezvous algorithm in finite graphs are known. In this chapter, the model variant $\mathcal{DF}$, formally stated in Definition 3.3, which is closer to asynchrony, is considered. Agents have the same measure of time but the adversary can impose, for each agent and each edge, the speed of traversing this edge by this agent. The speeds may be different for different edges and different agents but all traversals of a given edge by a given agent have to be at the same imposed speed.

Definition 3.3 (Model variant $\mathcal{DF}$)**.** *This model variant (cf. Definition 2.12) is the set of all environments (s, T, L, i, w, g) (cf. Definition 2.11) such that $s = (P, Z, Q, d, c, r)$ is a finite graph (cf. Definition 2.8), T is the continuous timeline (cf. Definition 2.5), and g verifies the following property. Let q and ℓ be any elements of Q and L, respectively. There exists $y \in T$ such that for every $t \in T$, the image by g of (t, q, ℓ) is y.*

This chapter presents a deterministic strong rendezvous algorithm for model variant $\mathcal{DF}$, whose duration is polynomial in the number n of nodes of the graph, the length $|\ell_{\min}|$ of the smaller label, and the maximum $\tau_{\max}$ of all traversal durations assigned by the adversary over all agent-edge couples. In other words, given an environment (s, T, L, i, w, g) of $\mathcal{DF}$ with $s = (P, Z, Q, d, c, r)$, $\tau_{\max} = \max\{y | \exists x \in T \times Q \times L, g(x) = y\}$.

3.1.3 Roadmap

The next section is dedicated to the presentation of some basic definitions and routines needed in the rest of this chapter. Section 3.3 gives the strong rendezvous algorithm and its proof. The existence of algorithms in alternative scenarios in terms of model or complexity parameters is discussed in Section 3.4. Lastly, Section 3.5 concludes this chapter.

3.2 Preliminaries

Universal eXploration Sequences. The Universal eXploration Sequences (`UXS`) [76], derived from the Universal Traversal Sequences (`UTS`) [5], are the root of several gathering or rendezvous procedure from the literature, and of several contributions presented in this thesis. An eXploration Sequence (`XS`) is a sequence of integers any mobile agent may follow in order to move in some graph. More precisely, it requires an integer input and can be used to compute a sequence of exit port numbers. Consider a mobile agent following some `XS` s from node v with input i. After performing j edge traversals by following s, in node u, A computes its $(j+1)$-th exit port

number. It is $(e + s[j + 1]) \bmod \mathtt{d}(u)$ with e being equal to i if $j = 0$ and to the entry port number of A in u after its j-th edge traversal following s otherwise.

An `XS` is said to be Universal for some set of graphs S, if for any graph G of S, any node v of G, and any integer input i, it allows any mobile agent following it from v in G with input i to visit every node of G. A set of graphs which is usually considered is, given some positive integer n, the set of the graphs with at most n nodes.

The existence of `UXS` for the set of graphs with at most n nodes (for any n) whose length is polynomial in n has been proved [76]. Two algorithms are known to build them. The first one requires a huge computation time but allows to build short `UXS`. It consists in testing successively every string with the desired length in every graph with at most n nodes, and with every possible initial conditions. The second one [96] only requires a polynomial in n number of computation operations, and a logarithmic in n space, but although the length of the `UXS` it produces is polynomial in n, it is suspected to be larger than the length of the `UXS` produced by the first algorithm.

Still regarding `UXS`, it is worth mentioning that after following some `XS` s from some node u, it is easy for any mobile agent to come back to u by "backtracking" s, that is to say by taking the same edges but in reverse order and opposite direction.

These sequences are used in this chapter as a procedure enabling every agent, given an upper bound N on the number n of nodes in the graph, to visit every node starting from any of them, and come back to it, using a polynomial in N number of edge traversals. We denote this procedure by `Explo`(N), and by `C`(`Explo`(N)) the number of edge traversals it requires.

Label transformation. Another tool inspired by the literature [48] is used in this chapter. It is a transformation on binary strings. Consider a label ℓ_A of an agent A, with binary representation $(b_1 \ \ldots \ b_{|\ell_A|})$. Define the *transformed label* of A to be the binary string $M(\ell_A) = (b_1 \ b_1 \ b_2 \ b_2 \ \ldots \ b_{|\ell_A|} \ b_{|\ell_A|} \ 0 \ 1)$. This transformation is made to ensure the following property that is used in the proof of correctness of our algorithm in Section 3.3.

Proposition 3.1. *Let ℓ_A and ℓ_B be two labels such that $\ell_A < \ell_B$. There exists one positive integer $i \leq |M(\ell_A)|$ such that $M(\ell_A)[i] \neq M(\ell_B)[i]$.*

Terminology. The agent woken up earlier by the adversary is called the *earlier* agent and the other agent is called the *later* agent. If agents are woken up simultaneously, these appellations are given arbitrarily.

Consider executions E_A and E_B, respectively of procedures P_A and P_B by agents A and B. Executions E_A and E_B are called *concurrent*, if the time segments that they occupy are not disjoint.

3.3 The Algorithm and its Analysis

The strong rendezvous procedure is called `StrongRV` (shown in Algorithm 3.1) and its execution requires to call procedure `Phase`(h) that is described in Algorithm 3.2. At a high level, `Phase`(h) consists of executions of `Explo`(h) and carefully scheduled waiting periods of various lengths, designed according to the bits of the transformed label of the agent. The aim is to guarantee a period in which one agent stays still at a node and the other visits all nodes of the graph.

Algorithm 3.1 Algorithm `StrongRV`

```
h ← 1
while agents have not met do
    execute Phase(h)
    h ← 2h
end while
declare that the rendezvous is achieved
```

Algorithm 3.2 `Phase`(h)

```
/* Initialization */
execute Explo(h)
wait for time 4h^2(∑_{j=0}^{log(h)}(C(Explo(2^j))))
/* Core */
i ← 1
while i ≤ h do
    if M(ℓ_A)[(i mod |M(ℓ_A)|) + 1] = 0 then
        wait for time 2hC(Explo(h))
        execute twice Explo(h)
    else
        execute twice Explo(h)
        wait for time 2hC(Explo(h))
    end if
    i ← i + 1
end while
/* End */
wait for time hC(Explo(2h))
execute Explo(h)
```

The correctness and time complexity of Algorithm `StrongRV` are now analyzed. In the following statements and proofs, α denotes the smallest power of two which upper bounds the following three numbers: the size $n \geq 2$ of the graph G, the length of the smaller transformed label $2|\ell_{\min}| + 2$, and the parameter $\tau_{\max}$ the maximum of all traversal durations assigned by the adversary over all edges of G.

The following proposition directly follows from the definitions of α and UXS.

Proposition 3.2. *For any positive integers x and y such that $x \geq \alpha$, x*`C(Explo(`y`))` *upper bounds the time required by any agent to execute* `Explo(`y`)` *in G.*

Proposition 3.3. *For any positive integer x and any power of two y such that $x \geq \alpha$ and $x \geq y$, $T_{x,y} = 4xy \sum_{z=0}^{\log(y)}$* `C(Explo(`$2^z$`))` *upper bounds the time required by any agent to execute the sequence $S_y =$* `Phase(`1`), Phase(`2`), ..., Phase(`$\frac{y}{4}$`), Phase(`$\frac{y}{2}$`), Explo(`y`)` *in graph G.*

Proof. Let an arbitrary positive integer at least α be assigned to x. The proof is made by induction on y. Consider the case where $y = 1$. In this case, the sequence S_y consists only of `Explo(`1`)`. In view of Proposition 3.2, $T_{x,1}$, which is equal to $4x$`C(Explo(`1`))`, upper bounds the time required by any agent to execute `Explo(`1`)`, which proves the first step of the induction. Now, assume that there exists a power of two $1 \leq z \leq x$, such that the statement of the proposition holds for $y = z$. The next paragraph proves that if $2z \leq x$, then the statement of the lemma holds also for $2z$.

Denote by `Suffix(`z`)` the sequence of instructions of `Phase(`z`)` deprived from the first call to `Explo(`z`)`. The sequence S_{2z} is successively made of S_z, `Suffix(`z`)` and `Explo(`$2z$`)`. By

the inductive hypothesis, the time required to execute S_z is upper bounded by $T_{x,z}$. By Proposition 3.2, the time required to execute $\texttt{Explo}(2z)$ is upper bounded by $x\texttt{C}(\texttt{Explo}(2z))$. In view of Algorithm 3.2 and Proposition 3.2, the time required to execute $\texttt{Suffix}(z)$ is upper bounded by $T_{x,z} + 2z(x+z)\texttt{C}(\texttt{Explo}(z)) + z\texttt{C}(\texttt{Explo}(2z)) + x\texttt{C}(\texttt{Explo}(z))$. Hence, the maximal duration of S_{2z} is upper bounded by $2T_{x,z} + (x+z)\texttt{C}(\texttt{Explo}(2z)) + \texttt{C}(\texttt{Explo}(z))(2z(x+z)+x)$, which is at most $2T_{x,z} + (2z(x+z) + 2x + z)\texttt{C}(\texttt{Explo}(2z))$ as $\texttt{C}(\texttt{Explo}(2z)) \geq \texttt{C}(\texttt{Explo}(z))$. Moreover, $2T_{x,z} + (2z(x+z) + 2x + z)\texttt{C}(\texttt{Explo}(2z)) \leq 2T_{x,z} + 4x(2z)\texttt{C}(\texttt{Explo}(2z)) = T_{x,2z}$. This shows that the statement of the proposition also holds when $y = 2z$, which proves the proposition. □

The following theorem proves the correctness of Algorithm StrongRV.

Theorem 3.1. *Algorithm* StrongRV *guarantees rendezvous in G by the time the first of the agents completes the execution of* $\texttt{Phase}(2\alpha)$.

Proof. Assume by contradiction that the statement of the theorem is false. Note that when any agent finishes the first execution of $\texttt{Explo}(\alpha)$ of $\texttt{Phase}(\alpha)$ (line 2 of Algorithm 3.2) it has visited every node of G, and thus the other agent has been woken up before the end of this execution, or else the agents would have met.

The core of $\texttt{Phase}(\alpha)$ (lines 5-15 of Algorithm 3.2) can be viewed as a sequence of α blocks, where the x-th block (for $1 \leq x \leq \alpha$) corresponds to processing bit $M(\ell_A)[(x \bmod |M(\ell_A)|) + 1]$ of the transformed label (with A the executing agent). Each of these blocks in turn can be viewed as a sequence of 4 sub-blocks, each of which corresponds either to a waiting period of length $\alpha\texttt{C}(\texttt{Explo}(\alpha)))$, or to a single execution of $\texttt{Explo}(\alpha)$. Let $I_1, I_2, \ldots, I_{4\alpha}$ (resp. $J_1, J_2, \ldots, J_{4\alpha}$) be the sequence of the 4α sub-blocks executed by agent A (resp. agent B) in the core of $\texttt{Phase}(\alpha)$. The proof of this theorem relies on the following claim.

Claim 3.1. *For every $1 \leq y \leq 4\alpha$, I_y and J_y are concurrent.*

Proof of the claim: Assume by contradiction that $y = x$ is the smallest integer for which it does not hold. Without loss of generality, suppose that the first agent to complete its x-th sub-block is A. If $x = 1$, then in view of Proposition 3.3, when A starts and finishes I_1, A_2 is executing the first waiting period of $\texttt{Phase}(\alpha)$. Since I_1 corresponds to $\texttt{Explo}(\alpha)$, as the first bit of a transformed label is always 1, a meeting occurs by the end of the execution of I_1 because $\alpha \geq n$, which is a contradiction. So, $x > 1$ and since x is minimal, I_{x-1} and J_{x-1} are concurrent. So, when A starts I_x, B is executing J_{x-1} and when A completes I_x, the execution of J_{x-1} has not yet been completed. This implies that the time required by A to execute I_x is shorter than the time required by B to execute J_{x-1}. Hence, I_x cannot be a sub-block corresponding to a waiting period, as each of these periods has length $\alpha\texttt{C}(\texttt{Explo}(\alpha))$, which upper bounds the duration of every sub-block corresponding to $\texttt{Explo}(\alpha)$ (in view of Proposition 3.2). Thus I_x corresponds to an execution of $\texttt{Explo}(\alpha)$, and so does J_{x-1}, as otherwise rendezvous would occur by the time I_x is completed, which would be a contradiction.

Consider the time lag between executions of I_x and J_x. Let $\theta_1 = t_B - t_A$, where t_A (resp. t_B) is the time when A (resp. B) starts I_x (resp. J_x). The time required by A (resp. B) to execute $\texttt{Explo}(\alpha)$ never changes because it is always executed from the initial node of A (resp. B), and is at most θ_1 (resp. at least θ_1). Moreover, in each block there are always four sub-blocks: either two waiting periods of $\alpha\texttt{C}(\texttt{Explo}(\alpha))$ followed by two $\texttt{Explo}(\alpha)$, or vice versa. Consider each of the four positions that can be occupied by I_x in its corresponding block. If it is the first, second, or fourth sub-block of its block, then since I_x and J_{x-1} correspond to $\texttt{Explo}(\alpha)$, the number of whole sub-blocks corresponding to $\texttt{Explo}(\alpha)$ (resp. the waiting period of $\alpha\texttt{C}(\texttt{Explo}(\alpha))$) that remain to be executed by A from t_A is the same as the number of those that remain to be executed by B from t_A. If I_x is the third sub-block of its block, then J_x and J_{x+1} correspond to the waiting period while I_x and I_{x+1} correspond to the execution of $\texttt{Explo}(\alpha)$. In this case, the number of whole blocks that remain to be executed by A from t_A is the same as the number of

those that remain to be executed by B from t_A. Moreover, in view of Proposition 3.2, the time needed by B to execute J_x and J_{x+1} is at least the time needed by A to execute I_x and I_{x+1}.

As a result, whichever the position of I_x in its block, A is the first agent to finish the core of `Phase`(α) and there exists a difference of $\theta_2 \geq \theta_1$ between the times when A and B complete this core. To conclude the proof of the claim, two cases are considered: either θ_2 is longer than the time A needs to execute `Explo`(2α), or not.

In the first case, since A executes `Explo`(α) faster than B, it necessarily completes the end of `Phase`(α) at least time θ_2 ahead of B. As a consequence, it starts executing `Phase`(2α), and in particular its first instruction `Explo`(2α), at least time θ_2 ahead of B. This implies that A completes the execution of the first instruction `Explo`(2α) of `Phase`(2α) before B starts it. Hence, A starts the execution of the first waiting period of `Phase`(2α) by the time B starts the first execution of `Explo`(2α) in `Phase`(2α). In view of Proposition 3.3, this leads to a meeting before any agent starts the core of `Phase`(2α), which is a contradiction. In the second case, A completes the last waiting period of `Phase`(α) at a time θ_2 ahead of B. Moreover, θ_2 is at most the duration of this waiting period and at least the time required by A to execute `Explo`(α). Hence, while A executes entirely the last instruction `Explo`(α) of `Phase`(α), B is waiting (it executes the last waiting period of `Phase`(α)). This leads to a meeting before any agent starts `Phase`(2α), and hence the second case also results in a contradiction, which proves the claim. $\star$

Moreover, in view of Proposition 3.1 and the claim, and since the length of the smaller transformed label is at most α, there exists a period during which an agent is waiting in `Phase`(α) while the other entirely executes `Explo`(α). Hence a meeting occurs before any agent starts `Phase`(2α), which is a contradiction and proves the theorem. □

According to Theorem 3.1, rendezvous occurs by the time the first of the two agents completes `Phase`(2α), which occurs, by Proposition 3.3, before this agent has spent at most a time $T_{4\alpha,4\alpha}$ since its wake up. However, in view of the definition of α and of the transformed labels, $T_{4\alpha,4\alpha}$ is polynomial in α and, thus, in n, $|\ell_{\min}|$ and $\tau_{\max}$. This proves the following theorem.

Theorem 3.2. *The execution time of Algorithm* `StrongRV` *is polynomial in* n, $|\ell_{\min}|$ *and* $\tau_{\max}$.

3.4 Discussion of Alternative Scenarios

Algorithm `StrongRV` shows that the duration of rendezvous can be polynomial in n, $|\ell_{\min}|$ and $\tau_{\max}$, where n is the number of nodes of the graph, $|\ell_{\min}|$ is the length of the smaller label, and $\tau_{\max}$ is the maximum of all traversal durations assigned by the adversary, over all edges of the graph, when time is counted since the wake-up of the earlier agent.

It is natural to ask if it is possible to construct a rendezvous algorithm whose time depends on n, $|\ell_{\min}|$ and $\tau_{\min}$, where $\tau_{\min}$ is the *minimum* of all traversal durations assigned by the adversary, over all edges of the graph. The answer is trivially negative, if time is counted, as in the previous section, since the wake-up of the earlier agent. Indeed, suppose that there exists such an algorithm working in some time $F(n, |\ell_{\min}|, \tau_{\min})$. The adversary assigns $t(A, e) > F(n, |\ell_{\min}|, \tau_{\min})$, for the first edge e taken by agent A, starts A at some time t_0 and delays the wake-up of agent B until time $t_0 + t(A, e)$. Rendezvous cannot happen before time $t_0 + t(A, e)$, which is a contradiction.

It turns out that the answer is also negative in the easier scenario, when time is counted since the wake-up of the agent that is woken up later. Consider even a simplified situation, where $t(A) = t(A, e)$ is the same for all edges e, and $t(B) = t(B, e)$ is the same for all edges e. In other words, each of the agents has a constant speed. Thus $\tau_{\max} = \max(t(A), t(B))$ and $\tau_{\min} = \min(t(A), t(B))$. Assume that $t(A) \leq t(B)$. Call A the *slower* agent, and B – the *faster* agent.

First notice that, if it is shown that any rendezvous algorithm must take time at least $\tau_{\max}$ since the wake-up of the later agent, the negative result follows, as the adversary can assign

$\tau_{\max} > F(n, |\ell_{\min}|, \tau_{\min})$. It is now shown that, indeed, for any rendezvous algorithm, there exists a behavior of the adversary for which this algorithm takes time at least $\tau_{\max}$ since the wake-up of the later agent.

Denote by β (resp. by γ) the waiting time between the wake-up of the faster (resp. slower) agent and the time when it starts its first edge traversal. Let $d = |\beta - \gamma|$.

If $\beta \geq \gamma$, the adversary wakes up the faster agent at some time t_0 and wakes up the slower agent at time $t_0 + d$. Both agents start traversing their first edge at the same time $t_0 + \beta$ and cannot meet before time $t_0 + \beta + \tau_{\max}$. Since both agents were awake at time $t_0 + \beta$, the claim follows.

If $\beta < \gamma$, the adversary wakes up the slower agent at some time t_0 and wakes up the faster agent at time $t_0 + d$. Both agents start traversing their first edge at the same time $t_0 + \gamma$ and cannot meet before time $t_0 + \gamma + \tau_{\max}$. Since both agents were awake at time $t_0 + \gamma$, the claim follows. This concludes the justification that it is impossible to guarantee rendezvous in time depending on n, $|\ell_{\min}|$ and $\tau_{\min}$, even when time is counted since the wake-up of the later agent.

The above remark holds under the strong scenario considered in this chapter. By contrast, the answer to the same question turns out to be positive in the weak scenario. This can be justified as follows. In [52] the authors showed a rendezvous algorithm with cost (measured by the total number of edge traversals) polynomial in n and $|\ell_{\min}|$ under the weak scenario, assuming that agents are totally asynchronous. Let $\mathcal{A}$ be this algorithm and let its cost be $K(n, |\ell_{\min}|)$, where K is some polynomial. Consider algorithm $\mathcal{A}$ in the simplified model mentioned above, where each of the agents has a constant speed, the speeds being possibly different, still under the weak scenario. Consider the part of the cost of the algorithm counted since the wake-up of the later agent. This cost is $K'(n, |\ell_{\min}|)$, where K' is some polynomial. Of course, the behavior of the agents in which an agent never waits at a node and crosses each edge at its constant speed is a possible behavior imposed by a totally asynchronous adversary, and hence the result of [52] still holds (under the weak scenario). Hence, if time is counted from the wake-up of the later agent, the time of the algorithm from [52] is at most $\tau_{\min} K'(n, |\ell_{\min}|)$, and therefore it is polynomial in n, $|\ell_{\min}|$ and $\tau_{\min}$.

Hence, the following observation shows a provable difference between the time of rendezvous under the strong and the weak scenarios, even in the situation when rendezvous is possible under both of these scenarios. If time is counted from the wake-up of the later agent, and agents have constant, possibly different velocities, then rendezvous in time depending on n, $|\ell_{\min}|$ and $\tau_{\min}$ cannot be guaranteed under the strong scenario, but there is a rendezvous algorithm working in time polynomial in n, $|\ell_{\min}|$ and $\tau_{\min}$ under the weak scenario.

3.5 Conclusion

The main contribution of this chapter is a strong rendezvous algorithm in finite graphs when an adversarial traversal duration is assigned to each couple made of an edge and an agent (model variant formally stated in Definition 3.3). Its duration is polynomial in n, $|\ell_{\min}|$ and $\tau_{\max}$, where n is the number of nodes of the graph, $|\ell_{\min}|$ is the length of the smaller label, and $\tau_{\max}$ is the maximum of all traversal durations assigned by the adversary.

The chapter is concluded by stating the following problem. What is the strongest adversary under which strong rendezvous in arbitrary finite graphs is possible? This is the case under an adversary that imposes possibly different speeds for different agents and different edges, but the speed must be the same for all traversals of a given edge by a given agent. On the other hand, it is easy to see that if the adversary can impose a different speed *for each traversal* of each edge by each agent (thus the speeds may vary for different traversals of the same edge by the same agent) then strong rendezvous is impossible even in the two-node graph.

Chapter 4

Asynchronous Approach in the Plane

Contents

4.1 Introduction

4.1.1 Related Work

As explained by Section 1.1.3.7, although several articles investigate approach in the plane [13, 40, 43, 49], they discretize the environment thanks to a reduction to weak rendezvous in infinite graphs such as the infinite grid.

The recent work on rendezvous in graphs includes successive improvements aiming at providing polynomial (in the respective appropriate parameters) algorithms both in synchronous and asynchronous settings, and both in finite graphs and in the infinite grid.

For the simplest case of finite graphs, such solutions are known for both synchronous [77, 98] and asynchronous [52] settings.

In the infinite grid, a rendezvous algorithm exists for the case when each agent is given a constant speed [49]. Its time complexity is polynomial in the initial Manhattan distance D separating the agents, the length $|\ell_{\min}|$ of the smallest label, and $\tau_{\min}$ the minimum among the inverses of the speeds, and its cost is polynomial in the two first parameters and does not depend on the third. This algorithm implies the existence of an algorithm for synchronous settings.

However, no algorithm for asynchronous settings whose cost is polynomial in D and $|\ell_{\min}|$ is known, there is only an algorithm whose cost is super-exponential in these two parameters [43]. It is worth mentioning an algorithm whose cost is polynomial in D but which relies on the assumption that the agents are location aware [13, 40]: each agent knows the coordinates of its initial position in some common coordinate system. This hypothesis enables the authors not only to reach polynomiality but also a fine-grained complexity since their cost belongs to $\mathrm{O}(D^2 \log(D))$.

To close this section, it is worth mentioning that it is unlikely that the existing asynchronous rendezvous algorithm for arbitrary finite graph [52] could be used to obtain an asynchronous rendezvous algorithm for the infinite grid. First, this algorithm has not a cost polynomial in D and $|\ell_{\min}|$. Actually, ensuring rendezvous at this cost is even impossible in an arbitrary finite graph, as witnessed by the case of the clique with two agents labeled 0 and 1: the adversary can hold one agent at a node and make the other agent traverse $\Theta(n)$ edges before rendezvous, in spite of the initial distance 1. Moreover, the validity of the algorithm for finite graphs closely relies on the fact that both agents must evolve in the same finite graph, which is clearly not the case in the infinite grid. In particular, even the natural attempt consisting in making each agent apply this algorithm within bounded grids of increasing size and centered in its initial position, does not permit to claim that rendezvous ends up occurring. Indeed, the bounded grid considered by an agent is never exactly the same as the bounded grid considered by the other one (although they may partly overlap), and thus the agents never evolve in the same finite graph. In some sense, traveling in the same finite graph allows the agents to circumvent a part of the differences caused by locality: there is agreement on the finite graph in which they are. However, this is not the case in the infinite grid.

4.1.2 Model and Reduction from Asynchronous Approach in the Plane to Weak Rendezvous in the Infinite Grid

This section formally describes two model variants: a first one for approach in the plane, and a second one for weak rendezvous in the infinite grid. Moreover, for completeness, the reduction mentioned in the previous section is stated.

4.1.2.1 Asynchronous Approach in the Plane

Before stating the model variant considered and the specifications of the task of approach, it is necessary to derive the model considered in the plane from the one which is presented in Chapter 2. Indeed, in the plane, when moving from one position to another, each mobile agent passes by several others. In particular, approach may be achieved while the agents are moving. This does not appear in the model of Chapter 2 since this feature would not make sense in the general settings which encompass the graphs: There is no other position a mobile agent visits when going from one node to an adjacent one in a graph. The next definitions introduce a new environment (cf. Definition 2.11) called plane environment and specify another rule applying instead of Definition 2.21 when studying the plane.

Definition 4.1 (Plane environment)**.** *A plane environment is an environment (s, T, L, i, w, g) (cf. Definition 2.11) with T the continuous timeline (cf. Definition 2.5), and s the Euclidean plane (P, Z, Q, d, c, r) (cf. Definition 2.10), extended by the addition of an application f from $T \times Q \times L$ to the set of the applications from T to P. More precisely, let t, (p_1, z), and ℓ be any elements of T, Q, and L, respectively. Denote by x the image by g of $(t, (p_1, z), \ell)$. The image by f of $(t, (p_1, z), \ell)$ is an application j such that $j(t) = p_1$, $j(t + x) = d(p_1, z)$, j is continuous on $[t, t + x]$, and the image by j of $[t, t + x]$ is $[p_1, d(p_1, z)]$.*

Definition 4.2 (Moving in a plane environment)**.** *Let $(e, \mathcal{A})$ be any execution (cf. Definition 2.13) with $e = (s, T, L, i, w, g, f)$ a plane environment such that $s = (P, Z, Q, d, c, r)$. Let A be any*

mobile agent with label ℓ which at time t, is at position p and starts to move with direction $z \in Z$. Let x (resp. j) be the image by g (resp. f) of $(t, (p, z), \ell)$. The completion time of the move is $t + x$. More precisely, at every time of $[t, t + x)$, A is moving, and at every time y of $[t, t + x]$, agent A is at position $j(y)$.

In other words, when moving from a position to another in the plane, each mobile agent passes by each position on the line segment between the origin and the destination. Besides deciding how much time every mobile agent spends in each of its move, the adversary can be viewed as able to move each mobile agent back and forth along the latter segment during the move.

With the two previous definitions, it is possible to state the model variant in which asynchronous approach is investigated, and to formally define the latter.

Definition 4.3 (Model variant $\mathcal{AP}$)**.** *This model variant (cf. Definition 2.12) is the set of all plane environments.*

Definition 4.4 (Approach)**.** *Let M be any model variant (cf. Definition 2.12), and $\mathcal{A}$ be any mobile agent algorithm (cf. Definition 2.2). Algorithm $\mathcal{A}$ achieves approach in model variant M if and only if for every environment e of M in which there are only two labels, in the execution $(e, \mathcal{A})$, there exists a time t at which, the positions p_1 and p_2 of the agents are within each other's range.*

Hence, approach is achieved whenever the positions of the two mobile agents are within each other's range (cf. Definition 2.6 for the statement of range). When the space considered is the Euclidean plane (cf. Definition 2.10), this means that they are at distance at most 1 from each other.

4.1.2.2 Asynchronous Weak Rendezvous in the Infinite Grid

This section focuses on the task of weak rendezvous and the model variant M such that achieving asynchronous approach in the plane, as described by the previous section, reduces to achieving weak rendezvous in M. This is rather straightforward since it does not require to derive the model of Chapter 2 but only to define the model variant. In particular, the specifications of the task of weak rendezvous also appear in Chapter 3, and are only recalled below.

Definition 4.5 (Weak rendezvous)**.** *Let M be any model variant (cf. Definition 2.12), and $\mathcal{A}$ be any mobile agent algorithm. Algorithm $\mathcal{A}$ achieves weak rendezvous in model variant M if and only if for every environment (s, T, L, i, w, g) of M such that $|L| = 2$, the execution $((s, T, L, i, w, g), \mathcal{A})$ (cf. Definition 2.13) verifies at least one of the three following propositions:*

- *there is a time at which the two mobile agents are at the same position.*
- *there are two positions p_1 and p_2, and a time at which one of the two mobile agents is moving from p_1 to p_2 while the other one is moving from p_2 to p_1.*
- *there are two positions p_1 and p_2, and four times $t_1 < t_2 < t_3 < t_4$ such that a first mobile agent starts moving from p_1 to p_2 at t_1 and finishes at t_4, and the second mobile agent starts moving from p_1 to p_2 too but at t_2 and finishes at t_3.*

Definition 4.6 (Model variant $\mathcal{AG}$)**.** *This model variant (cf. Definition 2.12) is the set of all environments (s, T, L, i, w, g) such that T is the continuous timeline (cf. Definition 2.5) and s is the infinite grid (cf. Definition 2.9).*

4.1.2.3 Reduction from Approach in $\mathcal{AP}$ to Weak Rendezvous in $\mathcal{AG}$

Theorem 4.1 ([49]). *If there exists a deterministic algorithm achieving weak rendezvous in model variant $\mathcal{AG}$, whose cost is polynomial in the Manhattan distance between the two starting nodes of the agents D and the length of the binary representation of the shortest of their labels $|\ell_{\min}|$, then there exists a deterministic algorithm achieving approach in model variant $\mathcal{AP}$, whose cost is polynomial in the Euclidean distance between the two initial positions of the agents Δ and $|\ell_{\min}|$.*

Proof. First of all, the plane is discretized. A partial bijection from some points of the plane to the nodes of the infinite grid is defined. More precisely, for any point v of the plane, it is possible to define M_v, such that v has an image $M_v(v)$ by M_v, and for every point u mapped by M_v, the points located North, East, South, and West at Euclidean distance 1 from u are mapped to the neighbours of $M_v(u)$ reachable by port N, E, S and W, respectively.

Let $e_1 = (s_1, T, L, i_1, w, g_1, f)$ be any element of $\mathcal{AP}$. Assume that $|L| = 2$. Denote by ℓ_1 and ℓ_2 its two elements, by A and B the agents with these labels, and by v and w the positions $i_1(\ell_1)$ and $i_1(\ell_2)$, respectively. Among the set of points which have an image by M_v, let X be the set of the points which are the closest to w. Let x be a node in X, arbitrarily chosen. Notice that the points of X are at distance at most $\frac{\sqrt{2}}{2} < 1$ from w. Let α be the vector $\vec{xw}$.

Let $e_2 = (s_2, T, L, i_2, w, g_2)$ be an environment constructed from e_1 as follows. Space s_2 is the infinite grid. Moreover, $i_2(\ell_1) = M_v(v)$ and $i_2(\ell_2) = M_v(x)$. For every $t \in T$, every position p of the Euclidean plane with an image by M_v, and every label ℓ of L, if $g_1(t, (p, (1, z_1)), \ell) = x$ with $z_1 = \frac{-\pi}{2}$, 0, $\frac{\pi}{2}$, or π, then $g_2(t, (M_v(p), z_2), \ell) = x$ with $z_2 = S$, E, N or W, respectively. Environment e_2 belongs to $\mathcal{AG}$.

Let R be any weak rendezvous algorithm in $\mathcal{AG}$ whose cost is polynomial in D and $|\ell_{\min}|$. In the execution (e_2, R), there is a time t at which the agents meet. Let u_1 and u_2 be the two nodes such that at t, the latest node of A is u_1, and the next would be u_2 (if it did not meet B on the way). The transformed algorithm R^* for approach in $\mathcal{AP}$ works as follows: Execute algorithm R replacing each instruction "take port N (resp. E, S, or W)" by "go North (resp. East, South, or West) at distance 1".

In view of the construction of e_2, at t, agent A starting at v, is at some point p, traveling from the point whose image by M_v is u_1, to that whose image is u_2. In the same way, agent B, starting at w, at time t, is at some point q such that $q = p + \alpha$. Hence both agents are at distance at most 1 from each other at time t, which means that approach is achieved. □

4.1.3 Contribution

This chapter presents an algorithm for asynchronous rendezvous in the infinite grid whose cost is polynomial in D and $|\ell_{\min}|$.

In view of the reduction from approach to rendezvous in the infinite grid presented in the previous section, the algorithm presented in this chapter implies the existence of an asynchronous approach algorithm whose cost is polynomial in D and $|\ell_{\min}|$.

4.1.4 Roadmap

The next section is dedicated to basic definitions. The solution, procedure `AsyncGridRV`, is sketched in Section 4.3, formally described in Sections 4.4 and 4.5. Section 4.6 presents the correctness proof and cost analysis of the procedure. Finally, some concluding remarks are made in Section 4.7.

4.2 Preliminaries

First of all, this chapter, like Chapter 3, makes use of the transformed label described in Section 3.2.

For any integer k, the *reverse path* of the path $e_1, \ldots, e_k$ is defined as the path $e_k, e_{k-1}, \ldots, e_1 = \overline{e_1, \ldots, e_{k-1}, e_k}$. The number of edge traversals performed by an agent during the execution of any procedure p is denoted by $C(p)$.

Moreover, consider two distinct nodes u and v. A specific path from u to v, denoted $P(u, v)$, is defined as follows. If there exists a unique shortest path from u to v, this shortest path is $P(u, v)$. Otherwise, consider the smallest rectangle $R_{(u,v)}$ such that u and v are two of its corners. $P(u, v)$ is the unique path among the shortest paths from u to v that traverses all the edges on the northern side of $R_{(u,v)}$. Note that $P(u, v) = \overline{P(v, u)}$.

An illustration of $P(u, v)$ is given in Figure 4.1.

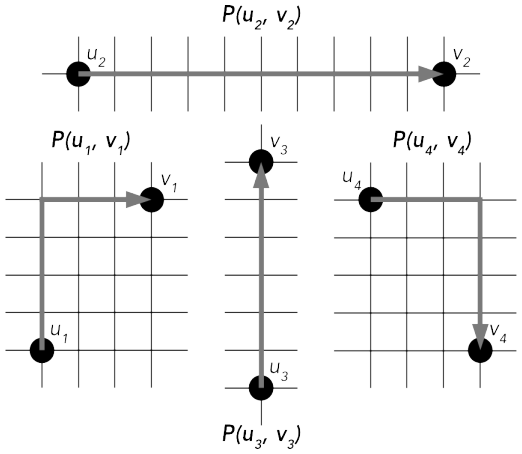


Figure 4.1: Some different cases for $P(u, v)$

4.3 Idea of the Algorithm

4.3.1 Informal Description in a Nutshell

This chapter aims at achieving rendezvous of two asynchronous mobile agents in the infinite grid and in a deterministic way. It is well known that solving rendezvous deterministically is impossible in some symmetric graphs (like the infinite grid) unless both agents are given distinct identifiers called labels. They are used to break the symmetry, i.e., when addressing asynchronous rendezvous, to make the agents follow different routes. The idea is to make each agent "read" its label binary representation, one bit at a time from the most to the least significant bits, and for each bit it reads, follow a route depending on the read bit. Procedure `AsyncGridRV` ensures rendezvous during some of the periods when they follow different routes i.e., when the two agents process two different bits.

Furthermore, to design the routes that both agents will follow, an upper bound on two parameters would be necessary: namely the initial distance between the agents and the length (of the binary representation) of the shortest label. As it is supposed that the agents have no knowledge of these parameters, they both perform successive "assumptions", in the sequel called *phases*, in order to find out such an upper bound. Roughly speaking, each agent attempts to estimate such an upper bound by successive tests, and for each of these tests, acts as if the

upper bound estimation was correct. Both agents first perform Phase 0. When Phase i does not lead to rendezvous, they perform Phase $i+1$, and so on. More precisely, within Phase i, the route of each agent is built in such a way that it ensures rendezvous if 2^i is a good upper bound on the parameters of the problem. Hence, this approach has two requirements: both agents are assumed (1) to process two different bits (i.e., 0 and 1) almost concurrently and (2) to perform Phase $i = \alpha$ almost at the same time—where α is the smallest integer such that the two aforementioned parameters are upper bounded by 2^α.

However, to meet these requirements, two major issues have to be faced. First, since the adversary can vary both agent speeds, the idea described above does not prevent the adversary from making the agents always process the same type of bit at the same time. Moreover, the route cost depends on the phase number, and thus, if an agent were performing some Phase i with i exponential in the initial distance and in the length of the binary representation of the smallest label, then the resulting procedure would not be polynomial. To tackle these two issues, the idea is to use a mechanism that prevents the adversary from making an agent execute the algorithm arbitrarily faster than the other without meeting. Each of these two issues is circumvented via a specific "synchronization mechanism". Roughly speaking, the first one makes the agents read and process the bits of the binary representation of their labels at nearly the same speed, while the second ensures that they start Phase α at almost the same time. This is particularly where the feat of strength is: orchestrating in a subtle manner these synchronizations in a fully asynchronous context while ensuring a polynomial cost. This completes the very high level idea of procedure `AsyncGridRV`. The next section gives more details.

4.3.2 Under the Hood

The approach described above allows to solve rendezvous when there exists an index for which the binary representations of both labels differ. However, this is not always the case especially when a binary representation is a prefix of the other one (e.g., 100 and 1000). Hence, instead of considering its own label, each agent will consider a transformed label: The transformation described in Section 3.2 guarantees the existence of the desired difference over the new labels. In the rest of this description, assume for convenience that the initial Manhattan distance D separating the agents is at least the length of the shortest binary representation of the two transformed labels (the complementary case adds an unnecessary level of complexity to understand the intuition).

As mentioned previously, procedure `AsyncGridRV` (refer to Algorithm 4.5 in Section 4.5) works in phases numbered $0, 1, 2, 3, 4, \ldots$ During Phase i refer to procedure `Assumption` called at line 3 in Algorithm 4.5), the agent supposes that the initial distance D is at most 2^i and processes one by one the first 2^i bits of its transformed label: In the case where 2^i is greater than the binary representation of its transformed label, the agent will consider that each of the last "missing" bits is 0. When processing a bit, the agent executes a particular route which depends on the bit value and the phase number. The route related to bit 0 (relying in particular on procedure `Berry` called at line 9 in Algorithm 4.6) and the route related to bit 1 (relying in particular on procedure `CloudBerry` called at line 11 in Algorithm 4.6) are obviously different and designed in such a way that if both these routes are executed almost simultaneously by two agents within a phase corresponding to a correct upper bound, then rendezvous occurs by the time any of them has been completed.

In the light of this, it turns out that an ideal situation would be that the agents concurrently start phase α and process the bits at quite the same rate within this phase where α denotes the smallest integer such that $2^\alpha \geq D$. Indeed, rendezvous would occur by the time the agents complete the process of the λ-th bit of their transformed label in phase α, where λ is the smallest index for which the binary representations of their transformed labels differ. However, getting such an ideal situation in presence of a fully asynchronous adversary appears to be really challenging. This is where the two synchronization mechanisms briefly mentioned above come

into the picture.

If the agents start Phase α approximately at the same time, the first synchronization mechanism (refer to procedure `RepeatSeed` called at line 15 in Algorithm 4.6) permits to force the adversary to make the agents process their respective bits at similar speed within Phase α, as otherwise rendezvous would occur prematurely during this phase before the process by any agent of the λth bit. This constraint is imposed on the adversary by dividing each bit process into some predefined steps and by ensuring that after each step s of the k-th bit process, for any $k \leq 2^\alpha$, an agent follows a specific route that forces the other agent to complete the step s of its k-th bit process. This route, on which the first synchronization is based, is constructed by relying on a simple principle that enables an agent to "push" the other. The principle is as follows: if an agent performs a given route X included in a given area $\mathcal{S}$ of the infinite grid, then the other agent can force it to finish route X by covering $\mathcal{S}$ as many times as there are edge traversals in X. More precisely, each covering of $\mathcal{S}$ allows to traverse all the edges of X at least once: so, in each covering the agent executing X must complete at least one edge traversal or rendezvous occurs. Hence, one of the major difficulties lies in the setting up of the second synchronization mechanism guaranteeing that the agents start Phase α around the same time. At first glance, it might be tempting to use an analogous principle to the one used for dealing with the first synchronization. Indeed, if an agent a_1 follows a route covering r times an area $\mathcal{Y}$ of the grid, such that $\mathcal{Y}$ is where the first $\alpha - 1$ phases of an agent a_2 take place and r is the maximal number of edge traversals an agent can make during these phases, then agent a_1 pushes agent a_2 to complete its first $\alpha - 1$ phases and to start Phase α. Nevertheless, a strict application of this principle to the case of the second synchronization directly leads to an algorithm having a cost that is super-polynomial in D and the length of the smallest label, due to a cumulative effect that does not appear for the case of the first synchronization. As a consequence, to force an agent to start its Phase α, the second synchronization mechanism does not depend on the kind of route described above, but on a much more complicated route that permits an agent to "*push*" the second one. This works by considering the "pattern" that is drawn on the grid by the second agent rather than just the number of edges that are traversed (refer to procedure `Harvest` called at line 1 in Algorithm 4.6). This is the most tricky part of procedure `AsyncGridRV`, one of the main idea of which relies in particular on the fact that some routes made of an arbitrarily large sequence of edge traversals can be pushed at a relative low cost by some other routes that are of comparatively small length, provided they are judiciously chosen. The following example illustrates this point. Consider an agent a_1 following from a node v_1 an arbitrarily large sequence of X_i, in which each X_i corresponds either to $A\overline{A}$ or $B\overline{B}$ where A and B are any routes ($\overline{A}$ and $\overline{B}$ corresponding to their respective backtrack i.e., the sequence of edge traversals followed in the reverse order). An agent a_2 starting from an initial node v_2 located at a distance at most d from v_1 can force agent a_1 to finish its sequence of X_i (or otherwise rendezvous occurs), regardless of the number of X_i, simply by executing $A\overline{A}B\overline{B}$ from each node at distance at most d from v_2. To support this claim, suppose by contradiction that it does not hold. At some point, agent a_2 necessarily follows $A\overline{A}B\overline{B}$ from v_1. However, note that if either agent starts following $A\overline{A}$ (resp. $B\overline{B}$) from node v_1 while the other is following $A\overline{A}$ (resp. $B\overline{B}$) from node v_1, then the agents meet. Indeed, this implies that the more ahead agent eventually follows $\overline{A}$ (resp. $\overline{B}$) from a node v_3 to v_1 while the other is following A (resp. B) from v_1 to v_3, which leads to rendezvous. Hence, when agent a_2 starts following $B\overline{B}$ from node v_1, agent a_1 is following $A\overline{A}$, and is not in v_1, so that it has at least started the first edge traversal of $A\overline{A}$. This means that when agent a_2 finishes following $A\overline{A}$ from v_1, a_1 is following $A\overline{A}$, which implies, using the same arguments as before, that they meet before either of them completes this route. Hence, in this example, agent a_2 can force a_1 to complete an arbitrarily large sequence of edge traversals with a single and simple route. Actually, the second synchronization mechanism implements this idea (this point is refined in Section 4.5). This was the most complicated thing to set up, as each part of route in every phase had to be orchestrated very carefully to permit,

in the end, this low cost synchronization while still ensuring rendezvous. However, it is through this original and novel way of moving that the polynomial cost is reached.

4.4 Basic Patterns

This section defines some sequences of moving instructions, i.e., patterns of moves, that will serve in turn as building blocks in the construction of the rendezvous algorithm. The main roles of these patterns are given in the next section when presenting the general solution.

4.4.1 Pattern `Seed`

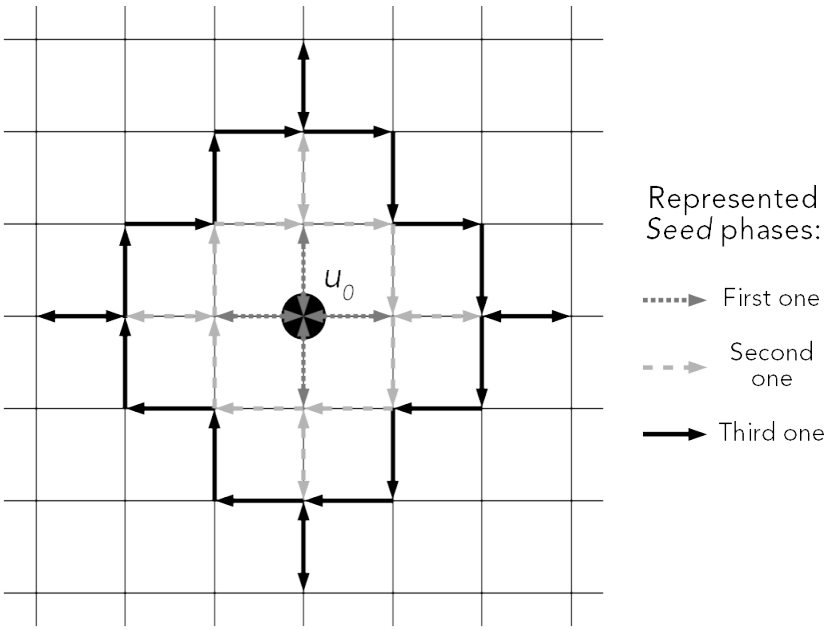


Figure 4.2: An illustration of the movements executed by an agent during the first period of `Seed`(3) from a node u_0. An arrow from a node x to a node y represents an edge traversal from x to y. Depending on the shape of the arrow, the represented movement is performed in a different phase.

Pattern `Seed` is involved as a sub-pattern in the design of all the other patterns presented in this section.

The description of pattern `Seed` is given in Algorithm 4.1. It is made of two periods. For a given non-negative integer x, the first period of pattern `Seed`(x) corresponds to the execution of x phases, while the second period is a complete backtrack of the path traveled during the first period. Pattern `Seed` is designed in such a way that it offers some properties that are shown in Section 4.6.1.2 and that are necessary to conduct the proof of correctness. One of the main purpose of this pattern is the following: starting from a node v, pattern `Seed`(x) allows to visit all nodes of the grid at distance at most x from v and to traverse all edges of the grid linking two nodes at distance at most x from v (informally, the procedure permits to cover an area of radius x). An illustration of pattern `Seed` is given in Figure 4.2.

Algorithm 4.1 Pattern `Seed`(x)

```
1: /* First period */
2: for i ← 1; i ≤ x; i ← i + 1 do
3:     /* Phase i */
4:     perform (N(SE)^i(WS)^i(NW)^i(EN)^i)
5: end for
6: /* Second period */
7: L ← the path followed by the agent during the first period
8: backtrack by following the reverse path L̄
```

4.4.2 Pattern `RepeatSeed`

Following the high level description of procedure `AsyncGridRV` (Section 4.3), `RepeatSeed` is the basic primitive procedure that implements the first synchronization mechanism (between two consecutive steps of a bit process). An agent a_1 executing pattern `RepeatSeed`(x, n) from a node u processes n times pattern `Seed`(x) from node u. All along this execution, a_1 stays at distance at most x from u. Moreover, once the execution is over, the agent is back at u.

The description of pattern `RepeatSeed` is given in Algorithm 4.2.

Algorithm 4.2 Pattern `RepeatSeed`(x, n)

```
execute n times pattern Seed(x)
```

4.4.3 Pattern `Berry`

According to Section 4.3, pattern `Berry` is used in particular to design the specific route that an agent follows when processing bit 0. The description of pattern `Berry` is given in Algorithm 4.3. It is made of two periods, the second of which is a backtrack of the first one. Pattern `Berry` offers several properties that are proved in Section 4.6.1.4 and used in the proof of correctness. Note that, Pattern `Berry`(x, y) executed from a node u for any two integers x and y allows, in particular, an agent to perform Pattern `Seed`(x) from each node at distance at most y from u. An illustration of pattern `Berry` is given in Figure 4.3.

Algorithm 4.3 Pattern `Berry`(x, y)

```
1: /* First period */
2: let u be the current node
3: for i ← 1; i ≤ x + y; i ← i + 1 do
4:     for j ← 0; j ≤ i; j ← j + 1 do
5:         for each node v at distance j from u ordered in the clockwise direction from the
   North do
6:             follow P(u, v)
7:             execute Seed(i − j)
8:             follow P(v, u)
9:         end for
10:    end for
11: end for
12: /* Second period */
13: L ← the path followed by the agent during the first period
14: backtrack by following the reverse path L̄
```

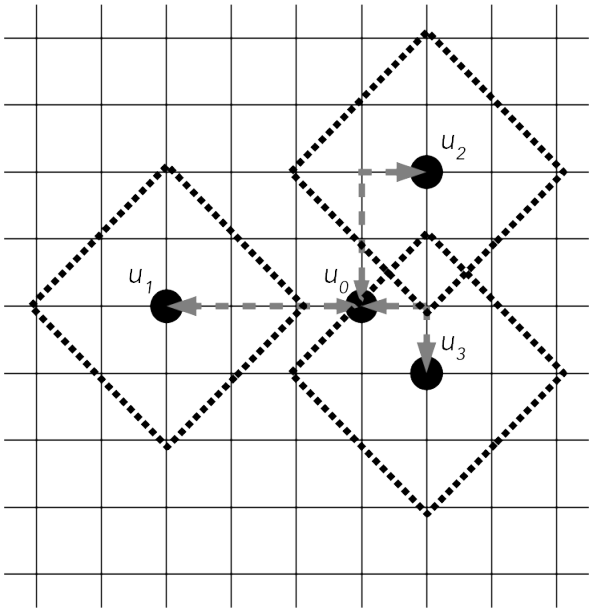


Figure 4.3: Illustration of a part of the route followed by an agent executing pattern `Berry`$(2,3)$ from a node u_0. When executing this pattern the agent has to execute many patterns `Seed` interleaved with executions of paths P from all nodes at distance at most 3 from u_0. Some of these patterns and paths are depicted in the figure. It is particularly the case of the dotted square centered at u_1 (resp. u_2 and u_3) that delimits the set of nodes that are visited when executing a pattern `Seed`(2) from node u_1 (resp. u_2 and u_3). Before executing `Seed`(2) from node u_1 (resp. u_2 or u_3), the agent follows $P(u_0, u_1)$ (resp. $P(u_0, u_2)$ or $P(u_0, u_3)$), and after executing `Seed`(2) from node u_1 (resp. u_2 or u_3), the agent follows the path $P(u_1, u_0)$ (resp. $P(u_2, u_0)$ or $P(u_3, u_0)$). These different paths P are represented by arrows.

4.4.4 Pattern `CloudBerry`

According to Section 4.3, pattern `CloudBerry` is used in particular to design the specific route that an agent follows when processing bit 1. The description of pattern `CloudBerry` is given in Algorithm 4.4. As for patterns `Seed` and `Berry`, the pattern is made of two periods, the second of which corresponds to a backtrack of the first one. Properties related to this pattern are given in Section 4.6.1.4. Note that, Pattern `CloudBerry`(x, y, z, h) executed from a node u for any integers x, y, z and h allows an agent to perform Patterns `Berry`(x, y) and `Seed`(x) from each node at distance at most z from u. Parameter h is an integer input that indicates in which order the agent has to visit each node at distance at most z from u (to execute Patterns `Berry`(x, y) and `Seed`(x) from each of these nodes). Playing on this order is used for technical reasons that are detailed in the proof of Theorem 4.2. An illustration of pattern `CloudBerry` is given in Figure 4.4.

Algorithm 4.4 Pattern `CloudBerry`(x, y, z, h)

1: /* First period */
2: let u be the current node
3: let U be the list of nodes at distance at most z from u ordered in the order of the first visit when applying `Seed`(z) from node u
4: **for** $i \leftarrow 0$; $i \leq 2z(z+1)$; $i \leftarrow i+1$ **do**
5: let v be the node with index $h+i \pmod{2z(z+1)+1}$ in U
6: follow $P(u, v)$
7: execute `Seed`(x)
8: execute `Berry`(x, y)
9: follow $P(v, u)$
10: **end for**
11: /* Second period */
12: $L \leftarrow$ the path followed by the agent during the first period
13: backtrack by following the reverse path $\overline{L}$

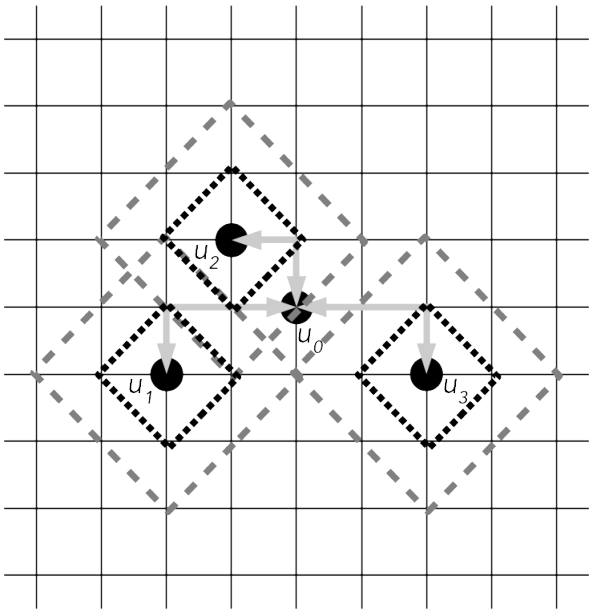


Figure 4.4: Illustration of a part of the route followed by an agent executing Pattern `CloudBerry`$(1, 2, 3, 0)$ from a node u_0. When executing this pattern the agent has to execute paths P as well as patterns `Seed` and `Berry` from all nodes at distance at most 3 from u_0 and in particular from nodes u_1, u_2 and u_3. To go to these nodes from u_0, the agent respectively follows $P(u_0, u_1)$, $P(u_0, u_2)$ and $P(u_0, u_3)$. Once in node u_1 (resp. u_2 and u_3) the agent executes `Seed`(1), which is represented by the smallest dotted square centered at u_1 (resp. u_2 and u_3) and then executes `Berry`$(1, 2)$, which is represented by the largest dotted square centered at u_1 (resp. u_2 and u_3), followed by $P(u_1, u_0)$ (resp. $P(u_2, u_0)$ and $P(u_3, u_0)$). All paths P are represented by arrows.

4.5 Main Algorithm

The asynchronous rendezvous in the infinite grid is provided by procedure `AsyncGridRV` whose formal description is provided by this section. For each subroutine involved, a description

of its main objectives and a high level explanation of how its works are also explained. The main algorithm that solves the rendezvous in the infinite grid is procedure `AsyncGridRV` (whose pseudo-code is given by Algorithm 4.5).

Algorithm 4.5 Procedure `AsyncGridRV`

```
1: d ← 1
2: while agents have not met yet do
3:     execute Assumption(d)
4:     d ← 2d
5: end while
```

Procedure `AsyncGridRV` makes use of a subroutine, i.e., procedure `Assumption`. When an agent executes this procedure with a parameter α that is a "good" assumption i.e., that upper-bounds the initial distance D and the value λ of the smallest bit position for which both transformed labels differ, rendezvous is guaranteed to occur by the end of this execution. In the rest of this section, α denotes the smallest assumption which upper-bounds D and λ.

The code of procedure `Assumption` is given in Algorithm 4.6. It can be divided into two parts. The first part consists of the execution of procedure `Harvest` (line 1 of Algorithm 4.6) and corresponds to the second synchronization mechanism mentioned in Section 4.3. The main feature of this procedure is the following: when the earlier agent finishes the execution of `Harvest`(α) within the execution of `Assumption`(α), the later agent is guaranteed to have at least started to execute `Assumption` with parameter α (actually, as explained below, it is even guaranteed that most of `Harvest`(α) has been executed by the later agent). Procedure `Harvest` is presented below. The second part of procedure `Assumption` (refer to lines $2-19$ of Algorithm 4.6) consists in processing the bits of the transformed label one by one. More precisely when processing a given bit in a call to procedure `Assumption`(d), the agent acts in steps $0, 1, \ldots, 2d(d+1)$: After each of these steps, the agent executes Pattern `RepeatSeed` whose role is described below. In each of these steps, the agent executes `Berry` (resp. `CloudBerry`) if the bit it is processing is 0 (resp. 1). These patterns of moves (refer to Algorithms 4.3 and 4.4 in Section 4.4) are made in such a way that rendezvous occurs by the time any agent finishes the process of its λ-th bit in `Assumption`(α) if the following synchronization property is verified. Each time any of the agents starts executing a step j during the process of its i-th bit in `Assumption`(α), the other agent has finished the execution of either step $j-1$ in the i-th bit process of `Assumption`(α) if $j > 0$, or the last step of the $(i-1)$-th bit process of `Assumption`(α) if $j = 0$ and $i > 0$. To obtain such a synchronization, an agent executes what is called the first synchronization mechanism in the previous section (refer to line 15 in Algorithm 4.6) after each step of a bit process. Actually, this mechanism relies on procedure `RepeatSeed`, the code of which is given in Algorithm 4.1. Note that the total number of steps, and thus of executions of `RepeatSeed`, in `Assumption`(α) is $2\alpha^2(\alpha+1)+\alpha$. For every $0 \leq k \leq 2\alpha^2(\alpha+1)+\alpha$, the k-th execution of `RepeatSeed` in `Assumption`(α) by an agent permits to force the other agent to finish the execution of its k-th step in `Assumption`(α) by repeating a pattern `Seed` (its main purpose is described just above its code given by Algorithm 4.2): with the appropriate parameters, this pattern `Seed` covers any pattern (`Berry` or `CloudBerry`) made in the k-th step of `Assumption`(α) and the number of times it is repeated is at least the maximum number of edge traversals made in the k-th step of `Assumption`(α).

Algorithm 4.7 gives the code of procedure `Harvest`. Procedure `Harvest` is made of two parts: the executions of procedure `PushPattern` (lines $1-3$ of Algorithm 4.7), and the calls to the patterns `CloudBerry` and `RepeatSeed` (lines $4-5$ of Algorithm 4.7). When `Harvest` is executed with parameter α (which is a good assumption), the first part ensures that the later agent has at least completed every execution of `Assumption` with a parameter that is smaller than α, while the second part ensures that the later agent has completed almost the entire execution

of `Harvest`(α) (more precisely, when the earlier agent finishes the second part, it is guaranteed that it remains for the later agent to execute at most the last line before completing its own execution of `Harvest`(α)).

Algorithm 4.6 Procedure `Assumption`(d)

```
1: execute Harvest(d)
2: radius ← 2d^4 + 3d
3: i ← 1
4: while i ≤ d do
5:     j ← 0
6:     while j ≤ 2d(d + 1) do
7:         // Begin of step j
8:         if the length of the transformed label is strictly greater than i, or its i-th bit is 0
   then
9:             execute Berry(radius, d)
10:        else
11:            execute CloudBerry(radius, d, d, j)
12:        end if
13:        // End of step j
14:        radius ← radius + 3d
15:        execute RepeatSeed(radius, C(CloudBerry(radius − 3d, d, d, j)))
16:        j ← j + 1
17:    end while
18:    i ← i + 1
19: end while
```

To give further details on `Harvest`, let us first describe procedure `PushPattern` (its code is given in Algorithm 4.8). When the earlier agent completes the execution of `PushPattern`($2i, d$) with i some power of two, assuming that the later agent had already completed `Assumption`(i), the later agent is guaranteed to have completed its execution of `Assumption`($2i$). To ensure this, the execution of `Assumption`($2i$) is regarded as a sequence of calls to basic patterns (namely `RepeatSeed`, `Berry` and `CloudBerry`), which is formally defined in Definition 4.7. This sequence is what lies behind "the pattern drawn on the grid" mentioned in Subsection 4.3.2. The sequence of calls to basic patterns of the earlier agent in `Assumption`($2i$) is quite similar to the one of the later agent: they have the same length and the s-th pattern of one sequence is `RepeatSeed` if and only if the s-th pattern of the other sequence is `RepeatSeed`. In fact, the only difference, due to distinct transformed labels, is that if the s-th pattern of one sequence is `Berry` (resp. `CloudBerry`), the s-th pattern of the other sequence may be either `Berry` or `CloudBerry`.

For each basic pattern p_s in its sequence, the earlier agent executes another pattern p'_s at the end of which the later agent is ensured to have completed the execution of the s-th basic pattern of its own sequence. Whether p_s is `Berry` or `CloudBerry`, p'_s is the same so that the earlier agent does not need to know the type of the s-th basic pattern in the sequence of the later agent in order to push it (and by extension, does not require the knowledge of the label of the later agent). More precisely, p'_s is chosen as follows.

If p_s is either pattern `Berry` or pattern `CloudBerry`, then p'_s is pattern `RepeatSeed`: the same idea as for the first synchronization mechanism it used once more. If p_s is pattern `RepeatSeed`, then p'_s is pattern `Berry`, relying on a property of the route $X\overline{X}$ (with X any non-empty route) introduced in the last paragraph of Subsection 4.3.2: if both agents follow this route concurrently from the same node, then they meet. Pattern `Seed` can be seen as such a route, and procedure `Berry` (whose code is shown in Algorithm 4.3) consists in executing pattern `Seed` from each node at distance at most α. Hence, unless they meet, the later agent completes its execution

of pattern `RepeatSeed` before the earlier one starts executing `Seed` from the same node. Note that `PushPattern` uses as many patterns as the number of basic patterns in the sequence it is supposed to push: this and the fact of doubling the value of the input parameter of procedure `Assumption` in Algorithm 4.5 contribute in particular to keep the polynomiality of procedure `AsyncGridRV`.

Thus, once the earlier agent completes the first part of `Harvest`(α), the later one has at least started the execution of `Assumption`(α) (and thus of the first part of `Harvest`(α)). At this point, at first glance, it might seem that the problem has just been shifted. Indeed, the number of edge traversals that has to be made to complete all the executions of `Assumption` prior to `Assumption`(α) is quite the same, if not higher, than the number of edge traversals that has to be made when executing the first part of `Harvest`(α). Hence the difference between both agents in terms of edge traversals has not been improved here. However, a crucial and decisive progress has nonetheless been done: contrary *a priori* to the series of `Assumption` executed before `Assumption`(α), the first part of `Harvest`(α) can be pushed at low cost via the execution of pattern `CloudBerry` (line 4 of Algorithm 4.7) by the earlier agent. Actually this pattern corresponds to the kind of route, described at the end of Subsection 4.3.2 for the second synchronization mechanism, which is of small length compared to the sequence of patterns it can push. Indeed, the first part of `Harvest`(α) can be viewed as a "large" sequence of patterns `Seed` and `Berry`: however `Seed` and `Berry` can be seen (by analogy with Subsection 4.3.2) as routes of the form $A\overline{A}$ and $B\overline{B}$ respectively, while pattern `CloudBerry` executes `Seed` and `Berry` (i.e., $A\overline{A}B\overline{B}$) once from at least each node at distance at most α.

Note that when the earlier agent has completed the execution of `CloudBerry` in `Harvest`(α), the later agent has at least started the execution of pattern `CloudBerry` in `Harvest`(α). Hence, there is still a difference between both agents, but it has been considerably reduced: it is now relatively small so that it can be handled pretty easily afterwards.

Algorithm 4.7 Procedure `Harvest`(d)

1: **for** $i \leftarrow 1$; $i < d$; $i \leftarrow 2i$ **do**
2: execute `PushPattern`(i, d)
3: **end for**
4: execute `CloudBerry`$(2d^4, d, d, 0)$
5: execute `RepeatSeed`$(2d^4 + 3d, C(\texttt{CloudBerry}(2d^4, d, d, 0)))$

Definition 4.7 (Basic and Perfect Decomposition)**.** *Given a call P to an algorithm, the basic decomposition of P, denoted by $\mathcal{BD}(P)$, is P itself if P corresponds to a basic pattern, the type of which belongs to* {`RepeatSeed`, `Berry`, `CloudBerry`}*. Otherwise, if P contains no call or contains a moving instruction outside of every call then $\mathcal{BD}(P) = \perp$, else $\mathcal{BD}(P) = \mathcal{BD}(x_1), \mathcal{BD}(x_2), \ldots, \mathcal{BD}(x_n)$ where $x_1, x_2, \ldots, x_n$ is the sequence (in the order of execution) of all the calls in P that are children of P. Moreover, $\mathcal{BD}(P)$ is a perfect decomposition if it does not contain any $\perp$.*

Remark 4.1. *The basic decomposition of every call to procedure* `Assumption` *is perfect.*

Algorithm 4.8 Procedure `PushPattern`(i, d)

1: **for** each p in $\mathcal{BD}($`Assumption`$(i))$ **do**
2: **if** p is a call to pattern `RepeatSeed` with value x as first parameter **then**
3: Execute `Berry`(x, d)
4: **else**
5: /* pattern p is either a call to pattern `Berry` or a call to pattern `CloudBerry` (in view of the above remark) and has at least two parameters */
6: Let x (resp. y) be the first (resp. the second) parameter of p
7: execute `RepeatSeed`$(d + x + 2y, C($`CloudBerry`$(x, y, y, 0)))$
8: **end if**
9: **end for**

4.6 Proof of Correctness and Cost Analysis

The purpose of this section is to prove that procedure `AsyncGridRV` ensures asynchronous rendezvous in the infinite grid at cost $\in \mathrm{O}((D + l)^{33})$ with D the initial distance between the agents and l, the length of the shortest label. To this end, the section is made of four subsections. The first two subsections are dedicated to technical results about the basic patterns presented in Section 4.4 and synchronization properties of procedure `AsyncGridRV`, which are used in turn to carry out the proof of correctness and the cost analysis of `AsyncGridRV` that are presented in the last two subsections.

4.6.1 Properties of the Basic Patterns

This subsection is dedicated to the presentation of some technical results about the basic patterns described in Section 4.4. They are used in the following subsections to prove the correctness of Algorithm 4.5.

4.6.1.1 Vocabulary

Before going any further, some extra vocabulary is introduced in order to facilitate the presentation of the next properties and lemmas.

Definition 4.8. *A pattern execution A precedes another pattern execution B iff the beginning of A occurs by the beginning of B.*

Definition 4.9. *Two pattern executions A and B are concurrent iff:*

- *pattern execution A does not finish before pattern execution B starts*
- *pattern execution B does not finish before pattern execution A starts*

By misuse of language, in the rest of this chapter, "a pattern execution" is sometimes referred to as "a pattern".

Hereafter a pattern A is said to *concurrently precede* a pattern B, iff A and B are concurrent, and A precedes B.

Definition 4.10. *A pattern A pushes a pattern B if for every execution in which B precedes A, agents meet before the end of the execution of A or B finishes before A.*

In the sequel, given two sequences of moving instructions X and Y, X is said to be a prefix of Y if Y can be viewed as the execution of the sequence X followed by another (possibly empty) sequence.

4.6.1.2 Pattern `Seed`

This section shows some properties related to pattern `Seed`. Proposition 4.1 follows by induction on the input parameter of pattern `Seed` and Proposition 4.2 follows from Algorithm 4.1.

Proposition 4.1. *Let x be any positive integer. Starting from a node v, pattern* `Seed`(x) *guarantees the following properties:*

1. *it allows to visit all nodes of the grid at distance at most x from v*
2. *it allows to traverse all edges of the grid linking two nodes at distance at most x from v*

Proposition 4.2. *Given two integers $x_1 \leq x_2$, the first period of pattern* `Seed`(x_1) *is a prefix of the first period of pattern* `Seed`(x_2).

Lemma 4.1. *Let x_1 and x_2 be two positive integers such that $x_1 \leq x_2$. Let a_1 and a_2 be two agents executing respectively patterns* `Seed`(x_1) *and* `Seed`(x_2) *both from the same node such that the execution of pattern* `Seed`(x_1) *concurrently precedes the execution of pattern* `Seed`(x_2). *Let t_1 (resp. t_2) be the time when agent a_1 (resp. a_2) completes the execution of pattern* `Seed`(x_1) *(resp.* `Seed`(x_2)*). Agents a_1 and a_2 meet by time $min(t_1, t_2)$.*

Proof. In view of Proposition 4.2, the first period of `Seed`(x_1) is a prefix of the first period of pattern `Seed`(x_2). If the path followed by agent a_1 during its execution of `Seed`(x_1) is $e_1, e_2, \ldots, e_n$,
$\overline{e_1, e_2, \ldots, e_n}$ (the over-lined part of the path corresponds to the backtrack), then the path followed by agent a_2 during the execution of pattern `Seed`(x_2) is $e_1, e_2, \ldots, e_n, s, \overline{e_1, e_2, \ldots, e_n, s}$ where s corresponds to the edges traversed at a distance $\in \{x_1 + 1; \ldots; x_2\}$.

There are two cases to consider. If agent a_2 completes $e_1, e_2, \ldots, e_n$ by the time a_1 completes $e_1, e_2, \ldots, e_n$, then agents a_1 and a_2 meet while they are following $e_1, e_2, \ldots, e_n$ as agent a_1 is the first agent that starts following $e_1, e_2, \ldots, e_n$. Otherwise, agent a_1 starts following $\overline{e_1, e_2, \ldots, e_n}$ while a_2 is still following $e_1, e_2, \ldots, e_n$: this implies that the agents meet by the time a_1 (resp. a_2) finishes $\overline{e_1, e_2, \ldots, e_n}$ (resp. $e_1, e_2, \ldots, e_n$). So, in both cases the agents meet by time $min(t_1, t_2)$, which concludes the proof of this lemma. □

4.6.1.3 Pattern `RepeatSeed`

This section is dedicated to some properties of pattern `RepeatSeed`. Informally speaking, Lemmas 4.2 and 4.3 describe the fact that pattern `RepeatSeed` pushes respectively pattern `Berry` and `CloudBerry` when it is given appropriate parameters.

Lemma 4.2. *Consider two nodes v_1 and v_2 separated by a distance δ. Let* `Berry`(x_1, y) *and* `RepeatSeed`(x_2, n) *be two patterns respectively executed from v_1 and v_2 with x_1, x_2, y and n positive integers. If $x_2 \geq x_1 + y + \delta$ and $n \geq C($*`Berry`$(x_1, y))$ *then pattern* `RepeatSeed`(x_2, n) *pushes pattern* `Berry`(x_1, y).

Proof. Denote by a_1 and a_2 the agents executing respectively `Berry`(x_1, y) and `RepeatSeed`(x_2, n). Suppose by contradiction that `RepeatSeed`(x_2, n) does not push `Berry`(x_1, y), which means, by Definition 4.10 that there exists an execution in which pattern `Berry`(x_1, y) precedes pattern `RepeatSeed`(x_2, n) such that a_1 neither meets a_2 nor completes `Berry`(x_1, y) before a_2 completes `RepeatSeed`(x_2, n). Remark that this implies in particular that these patterns are concurrent.

When executing its `Berry`(x_1, y) agent a_1 cannot be at a distance greater than $x_1 + y$ from its initial position v_1 and thus cannot be at a distance greater than $\delta + x_1 + y$ from node v_2. Also, in view of Proposition 4.1, each pattern `Seed`(x_2) executed from node v_2 which composes pattern `RepeatSeed`(x_2, n) allows to visit all nodes and to traverse all edges at distance at most x_2 from node v_2. Thus, each pattern `Seed`(x_2) executed from node v_2 allows to visit all nodes

and to traverse all edges (although not necessarily in the same order) that are traversed during the execution of pattern $\texttt{Berry}(x_1, y)$ from node v_1.

Consider the number of edge traversals completed by agent a_1 between the moment when a_2 starts executing any of the $\texttt{Seed}(x_2)$ which compose $\texttt{RepeatSeed}(x_2, n)$ and the moment when a_2 completes this $\texttt{Seed}(x_2)$. If a_1 has not completed a single edge traversal, then whether it was in a node or traversing an edge, it has met a_2 which traverses every edge a_1 traverses during its execution of $\texttt{Berry}(x_1, y)$. This is a contradiction, which implies that each time a_2 completes one of its executions of pattern $\texttt{Seed}(x_2)$, a_1 has completed at least one edge traversal. Since agent a_2 executes $n \geq C(\texttt{Berry}(x_1, y))$ times pattern $\texttt{Seed}(x_2)$, a_1 traverses at least $C(\texttt{Berry}(x_1, y))$ edges before a_2 finishes executing its $\texttt{RepeatSeed}(x_2, n)$. As $C(\texttt{Berry}(x_1, y))$ is the number of edge traversals in $\texttt{Berry}(x_1, y)$, when a_2 finishes executing pattern $\texttt{RepeatSeed}(x_2, n)$, a_1 has finished executing its pattern $\texttt{Berry}(x_1, y)$, which is a contradiction and proves the lemma. □

The following lemma can be proved using similar arguments to those used in the proof of Lemma 4.2.

Lemma 4.3. *Consider two nodes v_1 and v_2 separated by a distance δ. Let $\texttt{CloudBerry}(x_1, y, z, h)$ and $\texttt{RepeatSeed}(x_2, n)$ be two patterns respectively executed from v_1 and v_2 with x_1, x_2, y, z, h and n positive integers. If $x_2 \geq x_1 + y + z + \delta$ and $n \geq C(\texttt{CloudBerry}(x_1, y, z, h))$ then pattern $\texttt{RepeatSeed}(x_2, n)$ pushes pattern $\texttt{CloudBerry}(x_1, y, z, h)$.*

4.6.1.4 Pattern `Berry`

This section is dedicated to the properties of pattern `Berry`. Informally speaking, Lemma 4.4 describes the fact that pattern `Berry` permits to push pattern `RepeatSeed` when it is given appropriate parameters. Proposition 4.3 and Lemma 4.5 are respectively analogous to Proposition 4.2 and Lemma 4.1.

The following proposition follows from Algorithm 4.3.

Proposition 4.3. *Given four positive integers $x_1+y_1 \leq x_2+y_2$, the first period of $\texttt{Berry}(x_1, y_1)$ is a prefix of the first period of $\texttt{Berry}(x_2, y_2)$.*

Lemma 4.4. *Consider two nodes v_1 and v_2 separated by a distance δ. Let $\texttt{RepeatSeed}(x_1, n)$ and $\texttt{Berry}(x_2, y)$ be two patterns respectively executed from v_1 and v_2 with x_1, x_2, y and n positive integers. If $y \geq \delta$ and $x_1 \leq x_2$ then pattern $\texttt{Berry}(x_2, y)$ pushes pattern $\texttt{RepeatSeed}(x_1, n)$.*

Proof. Denote by a_1 and a_2 the agents executing respectively $\texttt{RepeatSeed}(x_1, n)$ and $\texttt{Berry}(x_2, y)$. Suppose by contradiction that $\texttt{Berry}(x_2, y)$ does not push $\texttt{RepeatSeed}(x_1, n)$ which means by Definition 4.10 that there exists an execution in which pattern $\texttt{RepeatSeed}(x_1, n)$ precedes pattern $\texttt{Berry}(x_2, y)$ such that a_1 neither meets a_2 nor completes $\texttt{RepeatSeed}(x_1, n)$ before a_2 completes $\texttt{Berry}(x_2, y)$. When executing $\texttt{Berry}(x_2, y)$, agent a_2 performs $\texttt{Seed}(x_2)$ from each node at distance at most y from v_2 with $y \geq \delta$. Thus, at some point, a_2 executes $\texttt{Seed}(x_2)$ from node v_1. In view of Lemma 4.1, since $x_2 \geq x_1$ and since by assumption, a_1 has not finished executing its $\texttt{RepeatSeed}(x_1, n)$ when a_2 starts executing pattern $\texttt{Seed}(x_2)$ from v_1, agents meet by the end of the latter and thus before the end of $\texttt{Berry}(x_2, y)$ which is a contradiction and proves the lemma. □

Lemma 4.5. *Consider two agents a_1 and a_2 executing respectively patterns $\texttt{Berry}(x_1, y_1)$ and $\texttt{Berry}(x_2, y_2)$ both from node v with x_1, x_2, y_1 and y_2 positive integers such that $x_2 + y_2 \geq x_1 + y_1$. Suppose that the execution of $\texttt{Berry}(x_1, y_1)$ by a_1 concurrently precedes the execution of $\texttt{Berry}(x_2, y_2)$ by a_2. Let t_1 (resp. t_2) be the time when agent a_1 (resp. a_2) completes its execution of pattern $\texttt{Berry}(x_1, y_1)$ (resp. $\texttt{Berry}(x_2, y_2)$). Agents a_1 and a_2 meet by time $min(t_1, t_2)$.*

Proof. This proof is similar to the proof of Lemma 4.1. In view of Proposition 4.3, if the path followed by agent a_1 during its execution of $\texttt{Berry}(x_1, y_1)$ is $e_1, e_2, \ldots, e_n, \overline{e_1, e_2, \ldots, e_n}$ (the overlined part of the path corresponds to the backtrack), then the path followed by agent a_2 during the execution of pattern $\texttt{Berry}(x_2, y_2)$ is $e_1, e_2, \ldots, e_n, s, \overline{e_1, e_2, \ldots, e_n, s}$ where s corresponds to the edges traversed from the $(x_1 + y_1 + 1)$-th iteration of the main loop of pattern `Berry` to its $(x_2 + y_2)$-th iteration.

There are two cases to consider. If agent a_2 completes $e_1, e_2, \ldots, e_n$ by the time a_1 completes $e_1, e_2, \ldots, e_n$, then agents a_1 and a_2 meet while they are following $e_1, e_2, \ldots, e_n$ as agent a_1 is the first agent that starts following $e_1, e_2, \ldots, e_n$. Otherwise, agent a_1 starts following $\overline{e_1, e_2, \ldots, e_n}$ while a_2 is still following $e_1, e_2, \ldots, e_n$: this implies that the agents meet by the time a_1 (resp. a_2) finishes $\overline{e_1, e_2, \ldots, e_n}$ (resp. $e_1, e_2, \ldots, e_n$). So, in both cases the agents meet by time $min(t_1, t_2)$, which concludes the proof of this lemma. □

4.6.1.5 Pattern `CloudBerry`

Informally speaking, the following lemma highlights the fact that pattern `CloudBerry` can push "a lot of basic patterns" under some conditions. In other words, an agent can be obliged to make a lot of edge traversals "at relative low cost".

Lemma 4.6. *Consider two nodes v_1 and v_2 separated by a distance δ. Assume that pattern $\texttt{CloudBerry}(x_1, y_1, z, h)$ is executed from v_2 with x_1, y_1, z and h four positive integers. Also assume that S is a sequence of patterns* `RepeatSeed` *and* `Berry` *executed from v_1. If $z \geq \delta$ and for each pattern* `RepeatSeed` *R and pattern* `Berry` *B belonging to S, $x_1 + y_1$ is greater than or equal to the sum of the parameters of B, and x_1 is greater than or equal to the first parameter of R, then pattern $\texttt{CloudBerry}(x_1, y_1, z, h)$ pushes S.*

Proof. Denote by a_1 and a_2 the agents executing respectively S and $\texttt{CloudBerry}(x_1, y_1, z, h)$. In order to prove that the execution of pattern $\texttt{CloudBerry}(x_1, y_1, z, h)$ by a_2 pushes the sequence of patterns S, suppose by contradiction that there exists an execution in which S precedes pattern $\texttt{CloudBerry}(x_1, y_1, z, h)$ such that a_1 neither meets a_2 nor completes its whole sequence of patterns before a_2 completes $\texttt{CloudBerry}(x_1, y_1, z, h)$.

In view of Algorithm 4.4, when executing $\texttt{CloudBerry}(x_1, y_1, z, h)$, a_2 executes pattern $\texttt{Seed}(x_1)$ followed by pattern $\texttt{Berry}(x_1, y_1)$ on each node at distance at most z from v_2. Since $z \geq \delta$, during its execution of $\texttt{CloudBerry}(x_1, y_1, z, h)$, a_2 follows $P(v_2, v_1)$, executes pattern $\texttt{Seed}(x_1)$ (denoted by p_1) and then pattern $\texttt{Berry}(x_1, y_1)$ (denoted by p_2) both from node v_1. The proof that the execution of $\texttt{CloudBerry}(x_1, y_1, z, h)$ by a_2 pushes the execution of S by a_1 consists in showing that the agents meet by the time a_2 completes its executions of p_1 and p_2.

By assumption, a_1 has not finished executing S when a_2 arrives on v_1 to execute p_1 and p_2. Consider what it can be executing at this moment. If it is executing pattern $\texttt{Seed}(x_2)$ with $x_2 \leq x_1$ a positive integer, then in view of Lemma 4.1, the agents meet by the end of the execution of p_1, which contradicts the assumption that the agents do not meet before the end of $\texttt{CloudBerry}(x_1, y_1, z, h)$. This means that when a_2 starts executing p_1, a_1 is executing pattern $\texttt{Berry}(x_2, y_2)$ for some positive integers x_2 and y_2 such that $x_2 + y_2 \leq x_1 + y_1$. After p_1, a_2 executes p_2. By Lemma 4.5, if a_1 is still executing pattern $\texttt{Berry}(x_2, y_2)$ for some positive integers x_2 and y_2 such that $x_2 + y_2 \leq x_1 + y_1$ (the same as above, or another) then the agents meet by the end of the execution of p_2 which is once again a contradiction. As a consequence, when a_2 starts executing p_2, a_1 is executing pattern $\texttt{Seed}(x_3)$ for some positive integer $x_3 \leq x_1$. Denote by p_3 this pattern, and remember that a_1 starts it after a_2 starts p_1. Moreover, when a_2 starts executing p_2, a_1 can not be in v_1 as it is the node where a_2 starts p_2, thus it has at least started the first edge traversal of p_3. Hence, p_1 concurrently precedes p_3, and a_2 completes the execution of p_1 before a_1 completes the execution of p_3.

In view of Algorithm 4.1, like in the proof of Lemma 4.1, the route followed by a_1 when executing p_3 can be denoted by $e_1, \ldots, e_n, \overline{e_1, \ldots, e_n}$ and the route followed by a_2 when executing

p_1 can be denoted by $e_1, \ldots, e_n, s, \overline{e_1, \ldots, e_n, s}$ where s corresponds to edges traversed at a distance belonging to $\{x_3 + 1; \ldots; x_1\}$. Remark that in view of the definition of a backtrack, $\overline{e_1, \ldots, e_n, s} = \overline{s}, \overline{e_1, \ldots, e_n}$. Consider the moment t_1 when a_1 completes the first period of p_3 and begins the second one. It has just traversed $e_1, \ldots, e_n$, and is about to follow $\overline{e_1, \ldots, e_n}$. At this moment, a_2 can not have started the edge traversals $\overline{e_1, \ldots, e_n}$, or else agents have met by t_1, which would be a contradiction. However, as p_1 is completed before p_3, a_2 must finish executing some non-empty part of $s\overline{s}$ followed by $\overline{e_1, \ldots, e_n}$ before a_1 finishes executing $\overline{e_1, \ldots, e_n}$ which implies that the agents meet by the end of the execution of p_1 and contradicts once again the hypothesis that they do not meet by the end of p_2.

So, in every case, the assumption that a_1 neither meets a_2 nor finishes executing S before the end of the execution of $\texttt{CloudBerry}(x_1, y_1, z, h)$, is contradicted. Hence, the execution of pattern $\texttt{CloudBerry}(x_1, y_1, z, h)$ by a_2 pushes the execution of S by a_1, and the lemma holds. □

4.6.2 Agents Synchronizations

This subsection aims at introducing and proving several synchronization properties offered by procedure `AsyncGridRV` (refer to Lemmas 4.10 and 4.11). The following statements and proofs make use of the notation D which denotes the initial distance separating the two agents in the infinite grid. The expression "synchronization" means that if one agent has completed some part of its rendezvous algorithm, then either it must have met the other agent or this other agent has also completed some part (not necessarily the same one) of its algorithm i.e., it must have made progress.

To prove Lemmas 4.10 and 4.11, some more technical results are necessary—Lemmas 4.7, 4.8, and 4.9.

Lemma 4.7. *Let v_1 and v_2 be the two nodes separated by a distance D that are initially occupied by the agents a_1 and a_2 respectively. Let c_1 and d_1 be two non-negative integers such that $d_1 \geq D$. Assume the prefix of the execution of agent a_1 is the sequence $S_1 = \texttt{Assumption}(1), \ldots, \texttt{Assumption}(2^{c_1})$. Assume that a part of the execution of agent a_2 is the sequence $S_2 = \texttt{PushPattern}(1, d_1), \ldots, \texttt{PushPattern}(2^{c_1}, d_1)$. Either the agents meet before the end of the execution of S_2 or S_1 finishes before S_2.*

Proof. Assume by contradiction that there exists some scenario E_1 in which neither the agents meet before the end of the execution of S_2 by agent a_2 nor the execution of S_1 by a_1 finishes before the execution of S_2 by a_2.

In view of Algorithm 4.8, and since there are as many occurrences of procedure `Assumption` in S_1 as of procedure `PushPattern` in S_2, there are as many basic patterns (among `RepeatSeed`, `Berry`, and `CloudBerry`) in $\mathcal{BD}(S_1)$ as in $\mathcal{BD}(S_2)$. Each basic pattern inside $\mathcal{BD}(S_1)$ and $\mathcal{BD}(S_2)$ is given an index between 1 and n according to its order of appearance. In view of Remark 4.1, for any integer d_2, $\mathcal{BD}(\texttt{Assumption}(d_2))$ is perfect, which implies that $\mathcal{BD}(S_1)$ is perfect too. This has the following consequences. When agent a_1 starts the execution of S_1, this agent starts the execution of the first basic pattern in $\mathcal{BD}(S_1)$. Moreover, when agent a_1 completes the execution of S_1, it completes the execution of the n-th basic pattern in $\mathcal{BD}(S_1)$. Lastly, for any integer i between 1 and $n-1$, agent a_1 does not make any edge traversal between the i-th and the $(i+1)$-th basic pattern in $\mathcal{BD}(S_1)$. In other words, every edge traversal agent a_1 makes during the execution of S_1 is performed during one of the basic patterns inside $\mathcal{BD}(S_1)$. Remark that $\mathcal{BD}(S_2)$ is perfect too.

The next step of this proof consists in showing by induction on i that for every integer i between 1 and n, a_1 either meets a_2 or completes the execution of the i-th pattern inside $\mathcal{BD}(S_1)$ before a_2 completes the execution of the i-th pattern inside $\mathcal{BD}(S_2)$. If $i = 1$, two cases are distinguished. In the first case, the first pattern of $\mathcal{BD}(S_2)$ starts before the first pattern of $\mathcal{BD}(S_1)$, while in the second case it does not i.e.,, in view of Definition 4.8, the first pattern of $\mathcal{BD}(S_1)$ precedes the first pattern of $\mathcal{BD}(S_2)$.

In the first case, since it does not make any edge traversal before the moment t_1 when it starts executing the first pattern of $\mathcal{BD}(S_1)$, a_1 is assumed to be in v_1 from the moment t_2 when a_2 starts executing the first pattern in $\mathcal{BD}(S_2)$ to t_1. Another scenario E_2 can be built in which a_1 (resp. a_2) executes S_1 (resp. S_2) from v_1 (resp. v_2) as in E_1, at every moment of E_2 both a_1 and a_2 are at the exact same place as in E_1, but in which the first pattern of $\mathcal{BD}(S_1)$ precedes the first pattern of $\mathcal{BD}(S_2)$. This is achieved by designing the behavior of the adversary in E_2 as follows. The adversary handles a_2 in the same way in E_2 as in E_1. From the moment t_3 at which a_1 starts executing S_1 in E_2 to the moment t_2 at which a_2 starts executing S_2 (both in E_1 and E_2), as well as from t_2 to the moment t_1 at which a_1 starts executing S_1 in E_1, in E_2, the adversary prevents a_1 from moving from v_1. Moreover, from t_1 on, in E_2, the adversary handles a_1 as in E_1. Since at every moment both a_1 and a_2 are at the same place in E_1 and E_2, it is enough to prove, in E_2, that a_1 either meets a_2 or completes the execution of the first pattern inside $\mathcal{BD}(S_1)$ before a_2 completes the execution of the first pattern inside $\mathcal{BD}(S_2)$, to show that this also holds in E_1. Also, in E_2, the first pattern of $\mathcal{BD}(S_1)$ precedes the first pattern of $\mathcal{BD}(S_2)$, this is the second of the two cases. Hence, when $i = 1$ it is enough to consider the second case only.

If the first pattern of $\mathcal{BD}(S_1)$ precedes the first pattern of $\mathcal{BD}(S_2)$, then in view of Lemmas 4.2, 4.3 and 4.4, Algorithm 4.8 and the fact that $d_1 \geq D$, whatever the type of the first pattern inside $\mathcal{BD}(S_1)$ (`Berry`, `CloudBerry` or `RepeatSeed`), a_1 either meets a_2 or completes the first pattern inside $\mathcal{BD}(S_1)$ before a_2 completes the first pattern inside $\mathcal{BD}(S_2)$.

Assume that there exists an integer j in $\{1, \ldots, (n-1)\}$ such that a_1 either meets a_2 or completes the j-th pattern inside $\mathcal{BD}(S_1)$ before a_2 completes the j-th pattern inside $\mathcal{BD}(S_2)$. This paragraph aims at showing that a_1 either meets a_2 or completes the $(j+1)$-th pattern inside $\mathcal{BD}(S_1)$ before a_2 completes the $(j+1)$-th pattern inside $\mathcal{BD}(S_2)$. In order to achieve this, suppose that the agents do not meet before the end of the execution of the $(j+1)$-th pattern inside $\mathcal{BD}(S_2)$ and show that the execution of the $(j+1)$-th pattern inside $\mathcal{BD}(S_1)$ finishes before the execution of the $(j+1)$-th pattern inside $\mathcal{BD}(S_2)$. In view of the induction hypothesis, and the assumption that the agents do not meet before the end of the execution of the $(j+1)$-th pattern inside $\mathcal{BD}(S_2)$, the j-th pattern inside $\mathcal{BD}(S_1)$ finishes before the j-th pattern inside $\mathcal{BD}(S_2)$ which means that the $(j+1)$-th pattern inside $\mathcal{BD}(S_1)$ precedes the $(j+1)$-th pattern inside $\mathcal{BD}(S_2)$. Again, in view of Lemmas 4.2, 4.3 and 4.4, Algorithm 4.8 and the fact that $d_1 \geq D$, whatever the type of the $(j+1)$-th pattern inside $\mathcal{BD}(S_1)$, it finishes before the $(j+1)$-th pattern inside $\mathcal{BD}(S_2)$. In particular, if the $(j+1)$-th pattern inside $\mathcal{BD}(S_1)$ is a `Berry` or a `CloudBerry` called after the test at line 8, at line 9 or 11, (refer to Algorithm 4.6), regardless of which of the two patterns it is, a_1 completes its execution before the end of the $(j+1)$-th pattern inside $\mathcal{BD}(S_2)$. Indeed, for any positive integers x, y, z and h, $\texttt{CloudBerry}(x, y, z, h)$ can be viewed as composed of several $\texttt{Berry}(x, y)$ so that $C(\texttt{CloudBerry}(x, y, z, h)) \geq C(\texttt{Berry}(x, y))$.

This means in particular that before the end of the n-th pattern inside $\mathcal{BD}(S_2)$ and thus before the end of S_2, a_1 either meets a_2 or completes the n-th pattern inside $\mathcal{BD}(S_1)$ and thus S_1 itself, which completes the proof. □

Lemma 4.8. *Let d_1 and x_1 be some integers such that the first parameter of each basic pattern inside $\mathcal{BD}(\texttt{Assumption}(d_1))$ is assigned a value which is at most x_1. For every integer $d_2 \geq d_1$, the first parameter of each basic pattern inside $\mathcal{BD}(\texttt{PushPattern}(d_1, d_2))$ is less than or equal to $x_1 + 3d_2$.*

Proof. In view of Algorithm 4.8, each basic pattern inside the basic decompositions $\mathcal{BD}(\texttt{Assumption}(d_1))$ and $\mathcal{BD}(\texttt{PushPattern}(d_1, d_2))$ (with $d_2 \geq d_1$ some integer) is given an index from 1 to n according to its order of appearance, with n the number of basic patterns in either of these decompositions. Thus, for any integer i from 1 to n, there is a pair of patterns (p_1, p_2) such that p_1 is the i-th basic pattern inside $\mathcal{BD}(\texttt{Assumption}(d_1))$, and p_2 is the i-th pattern inside $\mathcal{BD}(\texttt{PushPattern}(d_1, d_2))$. Thus, this proof consists in showing that there is no

such pair (p_1, p_2) such that the first parameter of p_2 is given a value greater than $x_1 + 3d_2$. To this end, three cases depending on the type of pattern p_1 are analyzed.

First consider the case in which p_1 is pattern $\texttt{RepeatSeed}(x_2, n_1)$ with $x_2 \leq x_1$ and n_1 two positive integers. In view of Algorithm 4.8, since p_1 is pattern $\texttt{RepeatSeed}(x_2, n_1)$, p_2 is $\texttt{Berry}(x_2, d_2)$, which means that its first parameter is at most x_1 and thus at most $x_1 + 3d_2$.

Consider the cases in which p_1 is either pattern `Berry` or pattern `CloudBerry`. First remark the following. In $\mathcal{BD}(\texttt{Assumption}(d_1))$, whether it is called directly by procedure $\texttt{Assumption}(d_1)$, or inside its call to $\texttt{Harvest}(d_1)$, or inside the call of the latter to $\texttt{PushPattern}(d_3, d_1)$ with some integer $d_3 < d_1$, the second parameter of pattern `Berry` is always d_1, and the second and third parameters of pattern `CloudBerry` are always d_1 as well.

In view of Algorithm 4.8, whether p_1 is pattern $\texttt{Berry}(x_2, d_1)$ or pattern $\texttt{CloudBerry}(x_2, d_1, d_1, h)$ with two positive integers h and $x_2 \leq x_1$, p_2 is $\texttt{RepeatSeed}(d_2 + x_2 + 2d_1, C(\texttt{CloudBerry}(x_2, d_1, d_1, h)))$. Its first parameter is $d_2 + x_2 + 2d_1$ which is at most $x_1 + 3d_2$.

Hence, within $\mathcal{BD}(\texttt{PushPattern}(d_1, d_2))$, there cannot be any call to a basic pattern in which the first parameter is assigned a value greater than $x_1 + 3d_2$, which proves the lemma. □

Lemma 4.9. *The first parameter of each basic pattern inside* $\mathcal{BD}(\texttt{Assumption}(d_1))$ *(with* d_1 *any power of two) is at most* $32d_1^4 - 6d_1$.

Proof. This lemma is proved by induction on d_1.

First consider that $d_1 = 1$. The basic patterns inside $\mathcal{BD}(\texttt{Assumption}(1))$ are enumerated and for each of them the first parameter is proved to be given a value which is less than or equal to $32d_1^4 - 6d_1 = 26$. Procedure $\texttt{Assumption}(1)$ begins with $\texttt{Harvest}(1)$ which is composed of calls to $\texttt{CloudBerry}(2, 1, 1, 0)$ and $\texttt{RepeatSeed}(5, C(\texttt{CloudBerry}(2, 1, 1, 0)))$, with both first parameters lower than 26. After $\texttt{Harvest}(1)$ too, the first parameter that is given to the patterns called in procedure $\texttt{Assumption}(1)$ is always at most 26. Indeed, the first parameter is assigned its maximum value when $j = 2d_1(d_1 + 1) = 4$ and $i = d_1 = 1$ i.e., when $3d_1 = 3$ has been added $i(j + 1) = 5$ times to the initial value of *radius* i.e., 5, which gives a maxiuml value equal to $5 + 15 = 20 < 26$. This concludes the analysis of the case when $d_1 = 1$.

Assume that there exists a power of two d_2 such that for each power of two $d_3 \leq d_2$, the first parameter of each basic pattern inside $\mathcal{BD}(\texttt{Assumption}(d_3))$ is at most $32d_2^4 - 6d_2$. The same method is used. The basic patterns inside $\mathcal{BD}(\texttt{Assumption}(2d_2))$ are enumerated and each of them is proved to be given a value for the first parameter which is at most $512d_2^4 - 12d_2$. Procedure $\texttt{Assumption}(2d_2)$ begins with $\texttt{Harvest}(2d_2)$ which in turn, begins with $\texttt{PushPattern}(1, 2d_2)$, ..., $\texttt{PushPattern}(d_2, 2d_2)$. By induction hypothesis, inside $\mathcal{BD}(\texttt{Assumption}(1))$, ..., $\mathcal{BD}(\texttt{Assumption}(d_2))$, the first parameter of each basic pattern is at most $32d_2^4 - 6d_2$. In view of Lemma 4.8, inside $\mathcal{BD}(\texttt{PushPattern}(1, 2d_2))$, ..., $\mathcal{BD}(\texttt{PushPattern}(d_2, 2d_2))$, the first parameter of each basic pattern is at most $32d_2^4 - 6d_2 + 6d_2 = 32d_2^4 < 512d_2^4 - 12d_2$. Moreover, after $\texttt{PushPattern}(1, 2d_2), \ldots, \texttt{PushPattern}(d_2, 2d_2)$, procedure $\texttt{Harvest}(2d_2)$ calls pattern $\texttt{CloudBerry}(32d_2^4, 2d_2, 2d_2, 0)$ followed by pattern $\texttt{RepeatSeed}(32d_2^4 + 6d_2, C(\texttt{CloudBerry}(32d_2^4, 2d_2, 2d_2, 0)))$. Inside these calls, the first parameter is respectively given the values $32d_2^4$ and $32d_2^4 + 6d_2$ which are both lower than $512d_2^4 - 12d_2$. Moreover, after $\texttt{Harvest}(2d_2)$, in the same way as when $d_1 = 1$, the first parameter can be proved to keep increasing and reach a maximum value equal to $32d_2^4 + 6d_2 + 12d_2^2(4d_2(2d_2 + 1) + 1) = 128d_2^4 + 48d_2^3 + 12d_2^2 + 6d_2 < 512d_2^4 - 12d_2$ which completes the proof of the lemma.

□

Before presenting the next lemma, it is necessary to introduce the following notions. The first four lines of procedure `Harvest` are referred to as its first part and the last line is referred to as the second one. Procedure `Assumption` begins with a call to procedure `Harvest`: the first part of procedure `Assumption` is considered to be the first part of this call, and that the second part of procedure `Assumption` is the second part of this call. After these two parts, there is a third part in procedure `Assumption` which consists of calls to basic patterns.

Moreover, note that the execution of procedure `AsyncGridRV` can be viewed as a sequence of consecutive calls to procedure `Assumption` with an increasing parameter. The $(i+1)$-th call to procedure `Assumption` (i.e., the call to procedure `Assumption`(2^i)) by an agent executing procedure `AsyncGridRV` is referred to as Phase i.

Lemma 4.10. *Consider two agents a_1 and a_2 executing procedure* `AsyncGridRV`*. Let i_1 and d_1 be two integers such that $2_1^i = d_1 \geq D$. Agent a_1 either meets a_2 or completes the execution of the first part of Phase i_1 before agent a_2 completes the execution of the second part of Phase i.*

Proof. Assume by contradiction that the lemma is false. This implies in particular that when a_2 finishes executing the second part of Phase i_1, a_1 is either executing Phase i_2 for an integer $i_2 < i_1$, or the first part of Phase i_1.

First of all, in view of Lemma 4.7 and since $d_1 \geq D$, a_1 either meets a_2 or finishes executing the sequence `Assumption`$(1), \ldots,$ `Assumption`(2^{i_1-1}) before a_2 completes the sequence `PushPattern`$(1, d_1)$, $\ldots$, `PushPattern`$(2^{i_1-1}, d_1)$ (i.e., the loop at the beginning of procedure `Harvest`(d_1)). Given that by assumption, agents do not meet before a_2 completes its execution of the second part of Phase i_1, a_1 starts executing the first part of Phase i_1 before a_2 finishes executing the loop at the beginning of procedure `Harvest`(d_1), which means that the execution of the loop at the beginning of procedure `Harvest`(d_1) by a_1 precedes the execution of `CloudBerry`$(2d_1^4, d_1, d_1, 0)$ by a_2.

This is at the root of the proof that when a_2 finishes executing `CloudBerry`$(2d_1^4, d_1, d_1, 0)$, a_1 has finished executing the loop at the beginning of procedure `Harvest`(d_1). In view of Lemmas 4.8 and 4.9, while executing this loop, a_1 executes a sequence of patterns `RepeatSeed` and `Berry` called by procedure `PushPattern` whose the first parameter is at most $2d_1^4$. Since $d_1 \geq D$, in view of Lemma 4.6 and the assumption that the agents do not meet before the end of the execution of the second part of Phase i_1 by a_2, when a_2 finishes executing `CloudBerry`$(2d_1^4, d_1, d_1, 0)$, a_1 has finished executing the loop.

After executing pattern `CloudBerry`$(2d_1^4, d_1, d_1, 0)$ but before completing procedure `Harvest`(d_1), a_2 performs `RepeatSeed`$(2d_1^4 + 3d_1, C($`CloudBerry`$(2d_1^4, d_1, d_1, 0)))$. In view of the previous paragraph, the execution of this pattern by a_2 is preceded by the execution of `CloudBerry`$(2d_1^4, d_1, d_1, 0)$ by a_1. When a_2 finishes executing `RepeatSeed`$(2d_1^4 + 3d_1, C($`CloudBerry`$(2d_1^4, d_1, d_1, 0)))$, in view of Lemma 4.3 and since by assumption the agents have not met, a_1 has finished executing pattern `CloudBerry`$(2d_1^4, d_1, d_1, 0)$. This means that when a_2 finishes executing `Harvest`(d_1) and thus the second part of Phase i_1, a_1 has completed the execution of the first part of Phase i_1, which proves the lemma.

□

The following lemma addresses the calls to pattern `RepeatSeed` in the second and in the third part of procedure `Assumption`(d_1) for any power of two d_1. In the statement and proof of this lemma, they are called "synchronization `RepeatSeed`", and indexed from 1 to $(d_1(2d_1(d_1 + 1) + 1) + 1)$ in their ascending execution order in these two parts of the procedure. During any execution of procedure `Assumption`(d_1) for any power of two d_1, the call to `RepeatSeed` in the second part of procedure `Assumption` is the first (indexed by 1) synchronization `RepeatSeed` of this procedure.

Lemma 4.11. *Let a_1 and a_2 be two agents executing procedure* `AsyncGridRV`*. Let v_1 and v_2 be their respective initial nodes separated by a distance D. For any power of two $d_1 \geq D$ and any positive integer $i \leq d_1(2d_1(d_1 + 1) + 1) + 1$, if the agents have not met yet, then when any of them completes the execution of the i-th synchronization* `RepeatSeed` *of* `Assumption`(d_1)*, the other agent has at least started it.*

Proof. Suppose that agent a_2 has just finished executing the i-th synchronization `RepeatSeed` inside procedure `Assumption`(d_1) for any power of two $d_1 \geq D$ and any positive integer $i \leq$

$d_1(2d_1(d_1+1)+1)+1$. This proof consists in showing by induction on i that if rendezvous has not occurred yet then a_1 has at least started executing this i-th synchronization `RepeatSeed`.

First consider the case in which $i = 1$. The synchronization `RepeatSeed` a_2 has just finished executing is called at the end of the execution of procedure `Harvest`(d_1) called at line 1 of procedure `Assumption`(d_1). Since $d_1 \geq D$, in view of Lemma 4.10, when a_2 completes the execution of the first synchronization `RepeatSeed` and thus the execution of `Harvest`(d_1), either the agents have met or a_1 has completed the execution of the first part of procedure `Assumption`(d_1) i.e., begun the execution of the first synchronization `RepeatSeed`.

Make the assumption that for any power of two $d_1 \geq D$, during any execution of procedure `Assumption`(d_1), there exists an integer j from 1 to $d_1(2d_1(d_1+1)+1)+1$ such that when agent a_2 completes the execution of the j-th synchronization `RepeatSeed`, either the agents have met or a_1 has at least started the execution of the j-th synchronization `RepeatSeed`, and prove that when a_2 completes the execution of the $(j+1)$-th synchronization `RepeatSeed`, either the agents have met or a_1 has at least started the execution of the same synchronization `RepeatSeed`. Assume by contradiction that when a_2 finishes executing the $(j+1)$-th synchronization `RepeatSeed`, a_1 has neither met a_2 nor started executing the $(j+1)$-th synchronization `RepeatSeed`.

After executing the j-th synchronization `RepeatSeed`, a_2 executes line 9 or line 11 of Algorithm `Assumption`(d_1) and thus either pattern `Berry` or pattern `CloudBerry`, depending on the bits of its transformed label. The induction hypothesis implies that the execution of the j-th synchronization `RepeatSeed` by a_1 precedes the execution by a_2 of either `Berry` or `CloudBerry` between the j-th and the $(j+1)$-th synchronization `RepeatSeed`. In view of Lemmas 4.4 and 4.6, as $d_1 \geq D$, whichever pattern a_2 executes, it pushes the execution of the j-th synchronization `RepeatSeed` by a_1. By assumption, when a_2 finishes executing line 9 or line 11 of Algorithm `Assumption`(d_1) after the j-th synchronization `RepeatSeed`, the agents have not met which implies that a_1 has finished executing the j-th synchronization `RepeatSeed`.

The next pattern that a_2 executes is the $(j+1)$-th synchronization `RepeatSeed`. Given the above assumptions and statements, when a_2 starts executing this synchronization `RepeatSeed`, a_1 has finished executing the j-th synchronization `RepeatSeed` and has started executing line 9 or line 11 of Algorithm `Assumption`(d_1). In view of Lemmas 4.2 and 4.3, since $d_1 \geq D$, whichever pattern a_1 executes, it is pushed by the execution of the $(j+1)$-th synchronization `RepeatSeed` by a_2. Given that, still by assumption, the agents do not meet before a_2 completes the execution of the $(j+1)$-th synchronization `RepeatSeed`, when this occurs, a_1 has completed the execution of line 9 or 11 of Algorithm `Assumption`(d_1), just after the j-th, and just before the $(j+1)$-th synchronization `RepeatSeed`. Hence, when a_2 completes the execution of the $(j+1)$-th synchronization `RepeatSeed`, a_1 has at least started executing the $(j+1)$-th synchronization `RepeatSeed`, which contradicts the hypothesis that when a_2 completes the execution of the $(j+1)$-th synchronization `RepeatSeed`, a_1 has neither met a_2 nor started executing the $(j+1)$-th synchronization `RepeatSeed`, and proves the lemma. □

4.6.3 Correctness of Procedure `AsyncGridRV`

Theorem 4.2. *Procedure* `AsyncGridRV` *solves the problem of asynchronous rendezvous in the infinite grid.*

Proof. To prove this theorem, it is enough to prove the following claim.

Claim 4.1. *Let d_1 be the smallest power of two such that $d_1 \geq max(D, l)$ with l the index of the first bit which differs in the transformed labels of the agents. Algorithm* `AsyncGridRV` *ensures rendezvous by the time any agent completes an execution of procedure* `Assumption`(d_1).

Proof of the claim: First, in view of Proposition 3.1, l exists. Respectively denote by v_1 and v_2, the initial nodes of two agents denoted by a_1 and a_2. This proof is made by contradiction. Suppose that the agents a_1 and a_2 execute procedure `AsyncGridRV` but do not meet by the time

any agent completes an execution of procedure `Assumption`(d_1) where d_1 is the smallest power of two such that $d_1 \geq max(D, l)$.

This in particular means that one of the agents eventually starts executing `Assumption`(d_1). Since $d_1 \geq D$, in view of Lemma 4.10, as soon as this agent completes the execution of procedure `Harvest`(d_1), both agents have started executing `Assumption`(d_1). Otherwise, agents have met which is a contradiction. Without loss of generality, suppose that the bits in the transformed labels of agents a_1 and a_2 with the index l are respectively 1 and 0.

In order to prove this claim, the first step consists in showing that there exists an iteration of the loop at line 6 of Algorithm 4.6 during which the two following properties are satisfied:

1. the value of variable i is equal to l

2. the value of variable j is such that when executing pattern `CloudBerry` at line 11, the first pair of patterns `Seed` and `Berry` executed inside this `CloudBerry` by a_1 starts from v_2

The first property follows from the fact that $d_1 \geq l$.

This paragraph aims at showing that the second property is verified too. Let U be a list of all the nodes at distance at most d_1 from v_1 and ordered in the order of the first visit when executing `Seed`(d_1) from node v_1. The same list is considered in the algorithm of pattern `CloudBerry`(x, d_1, d_1, h) for any positive integers x and h. First of all, there are $2d_1(d_1+1)+1$ nodes at distance at most d_1 from v_1, and thus in U. Since the distance between v_1 and v_2 is $D \leq d_1$, v_2 belongs to U. Denote by j_1 its index (between 0 and $2d_1(d_1+1)$) in U. According to procedure `Assumption`, the value of variable j is incremented at each iteration of the loop at line 6 and takes one after another each integer value between 0 and $2d_1(d_1+1)$. Consider the iteration when it is equal to j_1. According to Algorithm 4.4, the first node from which a_1 executes `Seed` and `Berry` is the node which has index $j_1 + 0 \pmod{2d_1(d_1+1)+1} = j_1$. This node is v_2, which proves that there exists an iteration of the loop at line 6 during which the second property is verified too. Denote it by I. It is the iteration after the $(1 + (l-1)(2d_1(d_1+1)+1) + j_1)$-th synchronization `RepeatSeed` inside Phase d_1.

In view of Lemma 4.11, when an agent completes its execution of the i-th synchronization `RepeatSeed` inside the second and the third part of any execution of procedure `Assumption`(d_1) (for any positive integer i less than or equal to $(d_1(2d_1(d_1+1)+1)+1)$, the other agent has at least begun the execution of this synchronization `RepeatSeed`. Thus, when an agent is the first one which starts executing I, it has just finished executing the $(1+(l-1)(2d_1(d_1+1)+1)+j_1)$-th synchronization `RepeatSeed` and the other agent is executing (or finishing executing) the same `RepeatSeed`. What follows proves that rendezvous occurs before any of the agents starts the next synchronization `RepeatSeed`.

Consider the patterns the agents execute between the beginning of the $(1+(l-1)(2d_1(d_1+1)+1)+j_1)$-th synchronization `RepeatSeed`, and the beginning of the next one. Agent a_1 executes pattern `RepeatSeed`(x, n) with x and n two positive integers (call this pattern p_1) and pattern `CloudBerry`(x, d_1, d_1, j_1) from node v_1 while a_2 executes `RepeatSeed`(x, n) (call it p_2) and `Berry`(x, d_1) (this is p_3) from node v_2. During its execution of pattern `CloudBerry`(x, d_1, d_1, j_1) from node v_1, a_1 first follows $P(v_1, v_2)$, and then executes pattern `Seed`(x) followed by pattern `Berry`(x, d_1) both from node v_2 (call them respectively p_4 and p_5). Recall that during any execution of pattern `Berry`(x, d_1) from node v_2, there are two periods, the second one consisting in backtracking every edge traversal made during the first one. During the first period, in particular, an agent executes a pattern `Seed`(x) from every node at distance at most d_1, among which there are node v_1 and node v_2. Since backtracking `Seed`(x) allows to perform exactly the same edge traversals as `Seed`(x), during the second period of pattern `Berry`(x, d_1), there is also an execution of pattern `Seed`(x) from node v_1 and another from v_2.

Consider two different cases. In the first one, when a_1 starts executing p_4 from v_2, inside p_3, a_2 has not yet started following $P(v_2, v_1)$ to go executing `Seed`(x) from v_1. In the second one,

when a_1 starts executing p_4 from v_2, a_2 has at least started following $P(v_2, v_1)$ to go executing $\texttt{Seed}(x)$ from v_1. In the following, these two cases are analyzed.

In the first case, consider what a_2 can be executing when a_1 starts executing p_4 from node v_2 after following $P(v_1, v_2)$. First, it can still be executing the synchronization $\texttt{RepeatSeed}$ p_2 from node v_2. Then, in view of Lemma 4.1, rendezvous occurs. The only other pattern that a_2 can be executing at this moment is p_3. However, in this case, a_2 will have finished its execution of p_3 before a_1 starts p_5, just after p_4. Otherwise, in view of Lemma 4.5, rendezvous occurs.

The reader has just been reminded that during any execution of pattern $\texttt{Berry}(x, d_1)$ from v_2, agent a_2 performs, among the patterns $\texttt{Seed}(x)$ from every node at distance at most d_1 from v_2, pattern $\texttt{Seed}(x)$ from v_2. If it executes one of these patterns $\texttt{Seed}(x)$ while a_1 is executing its p_4 from node v_2 after following $P(v_1, v_2)$, in view of Lemma 4.1, rendezvous occurs. This implies that before a_1 finishes following $P(v_1, v_2)$, a_2 has completed each execution of pattern $\texttt{Seed}(x)$ from v_2 inside its execution of $\texttt{Berry}(x, d_1)$.

This means that, each execution of pattern $\texttt{Seed}(x)$ from node v_2 during the second period of p_3 has already been completed by a_2 when a_1 starts executing its own $\texttt{Seed}(x)$ from v_2. Since inside the second period of p_3, a_2 executes pattern $\texttt{Seed}(x)$ from node v_2, a_2 has already executed the whole first period of p_3 when a_1 starts executing p_4 from v_2 including pattern $\texttt{Seed}(x)$ performed from node v_1, since v_1 is at distance at most d_1 from v_2. This contradicts the definition of this first case: according to this definition, when a_1 starts executing p_4 from v_2, inside p_3, a_2 has not followed $P(v_2, v_1)$ yet, and thus has not executed $\texttt{Seed}(x)$ from v_1.

In the second case, rendezvous is also proved to occur, which is a contradiction. Recall that in this case, when a_1 starts executing p_4 from v_2, a_2 has at least started following $P(v_2, v_1)$ to go executing $\texttt{Seed}(x)$ from v_1. If a_2 has not finished following $P(v_2, v_1)$ when a_1 starts following $P(v_1, v_2)$, then agents meet by time $min(t_1, t_2)$ since $P(v_1, v_2) = \overline{P(v_2, v_1)}$, where t_1 (resp. t_2) denotes the time when a_1 (resp. a_2) finishes following $P(v_1, v_2)$ (resp. $P(v_2, v_1)$). If a_2 has finished following $P(v_2, v_1)$ before a_1 starts executing $P(v_1, v_2)$, then it has started executing $\texttt{Seed}(x)$ from v_1 before a_1 finishes executing p_1 (before it executes $\texttt{CloudBerry}(x, d_1, d_1, j_1)$), which means in view of Lemma 4.1 that the agents achieve rendezvous.

So, whatever the execution chosen by the adversary, rendezvous occurs in the worst case by the time any agent completes $\texttt{Assumption}(d_1)$, which proves the claim, and by extension the theorem. ⋆ □

4.6.4 Cost Analysis

Theorem 4.3. *The cost of procedure* $\texttt{AsyncGridRV}$ *belongs to* $\mathrm{O}((D + |\ell_{\min}|)^{33})$.

Proof. In order to prove this theorem, the following two claims are required.

Claim 4.2. *The cost of each basic pattern inside* $\mathcal{BD}(\texttt{Assumption}(d_1))$ *(with* d_1 *any power of two) is in* $\mathrm{O}(d_1^{30})$.

Proof of the claim: To prove this claim, the most costly basic pattern which could belong to $\mathcal{BD}(\texttt{Assumption}(d_1))$ is exhibited and its cost is shown to belong to $\mathrm{O}(d_1^{30})$.

Examining their algorithms permits to state the following upper bounds on the costs of the basic patterns: $C(\texttt{Seed}(x)) \in \mathrm{O}(x^2)$, $C(\texttt{RepeatSeed}(x, n)) \in \mathrm{O}(n \times C(\texttt{Seed}(x)))$, $C(\texttt{Berry}(x, y)) \in \mathrm{O}((x + y)^5)$, and $C(\texttt{CloudBerry}(x, y, z, h)) \in \mathrm{O}(z^2 \times (z + C(\texttt{Seed}(x)) + C(\texttt{Berry}(x, y))))$. Remark that the higher the values of their parameters are, the higher the costs of the patterns are (except for the fourth parameter of $\texttt{CloudBerry}$ which does not impact its cost).

Also notice that pattern $\texttt{Seed}$ does not belong to $\mathcal{BD}(\texttt{Assumption}(d_1))$. It is called when executing the other basic patterns. Moreover, if they are given the same values for their two first parameters, pattern $\texttt{CloudBerry}$ is more costly than $\texttt{Berry}$, which makes it a good candidate for being the most costly pattern. But, when called with a second parameter which is the cost of

`CloudBerry`, pattern `RepeatSeed` is much more costly than the latter which makes it the most costly pattern inside $\mathcal{BD}(\texttt{Assumption}(d_1))$. Remark that in procedure `AsyncGridRV`, the second parameter of pattern `RepeatSeed` is the cost of either `Berry` or `CloudBerry`. In particular, it cannot be the cost of another `RepeatSeed`.

In view of Lemma 4.9, for any power of two d_1, inside $\mathcal{BD}(\texttt{Assumption}(d_1))$, the value of the first parameter given to the patterns is at most $32d_1^4 - 6d_1$. In addition, for each basic pattern `Berry` or `CloudBerry` inside $\mathcal{BD}(\texttt{Assumption}(d_1))$, the value given to its second parameter is always d_1. This gives upper bounds on the values of the parameters the most costly pattern can be given. Hence, the cost of each pattern called inside $\mathcal{BD}(\texttt{Assumption}(d_1))$ is at most $C(\texttt{RepeatSeed}(32d_1^4 - 6d_1, C(\texttt{CloudBerry}(32d_1^4 - 6d_1, d_1, d_1, h))))$ with h any positive integer. This cost belongs to $\mathrm{O}(C(\texttt{RepeatSeed}(d_1^4, d_1^2(d_1 + (d_1^4)^2 + (d_1^4)^5))))$ and thus to $\mathrm{O}((d_1^4)^2 d_1^{22})$ i.e., to $\mathrm{O}(d_1^{30})$. ⋆

Claim 4.3. *The cost of procedure* $\texttt{Assumption}(d_1)$ *(with* d_1 *any power of two) belongs to* $\mathrm{O}(d_1^{33})$.

Proof of the claim: In view of Definition 4.7 and Remark 4.1, for any power of two d_1, the cost of procedure $\texttt{Assumption}(d_1)$ is the same as the sum of the costs of all the basic patterns inside $\mathcal{BD}(\texttt{Assumption}(d_1))$. In view of Claim 4.2, for any power of two d_1, inside $\mathcal{BD}(\texttt{Assumption}(d_1))$, the cost of each basic pattern is in $\mathrm{O}(d_1^{30})$. Thus, to prove this claim it is enough to show that $\mathcal{BD}(\texttt{Assumption}(d_1))$ contains a number of basic patterns which is in $\mathrm{O}(d_1^3)$.

For any power of two d_1, procedure $\texttt{Assumption}(d_1)$ is composed of a call to $\texttt{Harvest}(d_1)$ and the nested loops. These loops consist in $2d_1(2d_1(d_1 + 1) + 1)$ calls to basic patterns. Half of them are made to `RepeatSeed` and the others either to `Berry` or to `CloudBerry`. In its turn, $\texttt{Harvest}(d_1)$ is composed of two parts: a loop calling procedure `PushPattern` and two basic patterns. For any power of two d_2, in view of Algorithm 4.8, and since they are both perfect, the number of basic patterns inside $\mathcal{BD}(\texttt{PushPattern}(d_2, d_1))$ or $\mathcal{BD}(\texttt{Assumption}(d_2))$ is the same. As a consequence, if $d_1 \geq 2$, $\mathcal{BD}(\texttt{PushPattern}(1, d_1))$, …, $\mathcal{BD}(\texttt{PushPattern}(\frac{d_1}{2}, d_1))$ is composed of as many basic patterns as there are in $\mathcal{BD}(\texttt{Assumption}(1))$, …, $\mathcal{BD}(\texttt{Assumption}(\frac{d_1}{2}))$.

For any power of two i, denote by $L_1(i)$ (resp. $L_2(i)$) the number of calls to basic patterns inside $\mathcal{BD}(\texttt{Assumption}(i))$ (resp. $\mathcal{BD}(\texttt{Harvest}(i))$). These numbers verify the following equations:

$$L_1(i) = L_2(i) + 2i(2i(i+1)+1)$$
$$L_2(i) = \sum_{j=0}^{\log_2(i)-1} (L_1(2^j)) + 2$$

They imply the following:

$$L_2(1) = 2 \quad \text{and}$$
$$\text{if} \quad i \geq 2 \quad \text{then} \quad L_2(i) = L_2(\frac{i}{2}) + L_1(\frac{i}{2}) = 2L_2(\frac{i}{2}) + i(i(\frac{i}{2}+1)+1)$$

which can also be written

$$L_2(i) = 2i + \sum_{j=1}^{\log_2(i)} (2^{\log_2(i)-j} \cdot 2^j(2^j(2^{j-1}+1)+1))$$
$$L_2(i) = 2i + i \sum_{j=1}^{\log_2(i)} (2^j(2^{j-1}+1)+1)$$
$$L_2(i) = 2i + i \sum_{j=1}^{\log_2(i)} (2^{2j-1} + 2^j + 1)$$
$$L_2(i) = 2i + i(\log_2(i) + \frac{2(i^2-1)}{3} + 2(i-1))$$

Hence, both $L_2(i)$ and $L_1(i)$ belong to $\mathrm{O}(i^3)$. This means that for any power of two d_1, $\mathcal{BD}(\texttt{Assumption}(d_1))$ is composed of a number of basic patterns which is in $\mathrm{O}(d_1^3)$. Hence, in view of Claim 4.2, the cost of $\texttt{Assumption}(d_1)$ indeed belongs to $\mathrm{O}(d_1^{33})$, which proves the claim. ⋆

Now, it remains to conclude the proof of the theorem. In view of Claim 4.1, rendezvous is achieved by the end of the execution of $\texttt{Assumption}(\delta)$ by any of the agents, where δ is the smallest power of two such that $\delta \geq max(D, l)$ and l is the index of the first bit which differs in the transformed labels of the agents. Moreover, in view of Claim 4.3, the cost of each call to $\texttt{Assumption}(d_1)$ for some power of two $d_1 \leq \delta$ belongs to $\mathrm{O}(d_1^{33})$. Since $\sum_{i=0}^{\log \delta}(2^{i^{33}}) \leq 2\delta^{33}$, the sum of the costs of these calls to procedure `Assumption` and thus the cost of procedure `AsyncGridRV` until rendezvous is achieved belongs to $\mathrm{O}(\delta^{33})$. Moreover, by construction, $l \leq 2|\ell_{\min}| + 2$. This means that the cost of `AsyncGridRV` belongs to $\mathrm{O}((D + |\ell_{\min}|)^{33})$. □

4.7 Conclusion

From Theorems 4.1, 4.2 and 4.3, we obtain the following result concerning the task of approach in the plane.

Theorem 4.4. *The task of approach can be solved at cost polynomial in the unknown initial distance Δ separating the agents and in the length of (the binary representation) of the shortest of their labels.*

Throughout the paper, we made no attempt at optimizing the cost. Actually, as the attentive reader will have noticed, our main concern was only to prove the polynomiality. Hence, a natural open problem is to find out the optimal cost to solve the task of approach. This would be all the more important as in turn we could compare this optimal cost with the cost of solving the same task with agents that can position themselves in a global system of coordinates (the almost optimal cost for this case is given in [40]) in order to determine whether the use of such a system (e.g., GPS) is finally relevant to minimize the traveled distance.

Chapter 5

Byzantine Gathering in Finite Graphs

Contents

5.1 Introduction

5.1.1 Introduction and Related Work

The scale-up when considering numerous agents is inevitably tied to the occurrence of faults among them, the most emblematic of which is the Byzantine one. Byzantine faults are very interesting under multiple aspects, especially because the Byzantine case is the most general one, as it subsumes all the other kinds of faults. In the field of fault tolerance they are considered as the worst faults that can occur. This chapter investigates the problem of gathering in the presence of agents subject to Byzantine faults called Byzantine agents.

As explained in Section 1.1.3.6, the behavior of the Byzantine agents is arbitrary and unpredictable. They can be viewed as malicious, controlled by some adversary trying to make the task fail, or at least to decrease the efficiency of its achievement. In particular, the gathering of all mobile agents as stated by Definition 2.26 cannot be ensured since the Byzantine agents may refuse to stay with the other (good) agents, or declare that the gathering is achieved at any time. This motivates the introduction of the task called Byzantine gathering or Byzantine gathering. It is very similar to gathering. It consists in gathering all good agents and having them all declare that the gathering is achieved. Nothing is expected from the Byzantine agents.

Gathering in finite graphs in presence of many Byzantine agents is considered in a similar model in earlier articles [23, 51] detailed in Section 1.1.3.6. They address the question of the number of good agents which is necessary and sufficient to achieve Byzantine gathering. To answer it, their authors propose deterministic algorithms having two requirements. First of all,

each of these algorithms requires that all good agents know the values of the some parameters of the problem namely the number of nodes of the graph n, and the number of Byzantine agents f. Then, these algorithms suffer from a time complexity which is super-exponential in n and the labels of the agents. Despite the requirements of their algorithms, the authors provide the number of good agents which is necessary and sufficient to achieve Byzantine gathering, in several cases.

This chapter is a continuation of the existing articles related to Byzantine gathering in finite graphs. Attention is in particular paid to the two aforementioned requirements, by aiming at presenting an algorithm for Byzantine gathering which is polynomial in n and $|\ell_{\min}|$, with $|\ell_{\min}|$ the length of the shortest label among those of the good agents.

The assumption that the mobile agents a priori know some information about the environment or their initial positions does not only appear in these articles about Byzantine gathering [11]. The following question naturally arises: Does assuming that they know this information makes the task easier than if they had that other information? Under the paradigm of algorithms with advice [1, 38, 61, 62, 72, 90, 101], all entities are given a same binary string as input. A particular attention is paid to the length of this string. It allows to compare the amount of information required by two algorithms. An algorithm requires more information than another one if the input string it requires is longer. This idea can also be used to compare the amount of information required by the task, and thus somehow their difficulty. However, it is worth noticing than not every pair of lengths can be compared: they may depend on different and independent parameters of the task. This is the second aim of this chapter, using this paradigm of algorithms with advice, and present an algorithm which requires an advice (called global knowledge in this chapter) of asymptotically optimal length.

5.1.2 Model

The introduction of Byzantine agents, and information initially provided to the good agents requires to derive the model presented in Chapter 2. The definitions presented below are only used in this chapter.

This section starts by introducing the Byzantine agents, and introducing a new kind of environment to take them into account.

Definition 5.1 (Byzantine and good agents)**.** *There are two kinds of mobile agents (cf. Definition 2.1): the Byzantine ones and the good ones.*

Definition 5.2 (Byzantine environment)**.** *Each Byzantine environment is an environment (s, T, L, i, w, g) (cf. Definition 2.11) extended by the addition of two elements:*

- *a subset F of L.*
- *an application b from F to the set of the mobile agent algorithms (cf. Definition 2.2).*

The next definitions introduce the global knowledge initially provided to the good agents. The idea behind them is to represent the information by a binary string computed from a part of the Byzantine environment by some oracle which, along with the mobile agent algorithm, is part of an algorithm with advice.

Definition 5.3 (Algorithm with advice)**.** *Every algorithm with advice is a mobile agent algorithm (cf. Definition 2.2) extended by the addition of an oracle. The latter is an application whose domain is the set of 6-tuples (s, T, L, i, w, g, F) such that there exists b such that (s, T, L, i, w, g, F, b) is a Byzantine environment. Its codomain is the set of the binary strings.*

Definition 5.4 (Global knowledge initially learnt from the oracle)**.** *Let $(e, \mathcal{A})$ be any execution (cf. Definition 2.13) with $e = (s, T, L, i, w, g, F, b)$ a Byzantine environment and $\mathcal{A}$ an algorithm*

with advice. Denote by o the oracle of $\mathcal{A}$. Upon waking up, every mobile agent learns $o((s, T, L, i, w, g, F))$. This information is called global knowledge.

In order to complete the derivation of the model from Chapter 2, it is necessary to describe what instructions the Byzantine agents execute. This is explained by Definition 5.5 which is a rewriting of Definition 2.14 for Byzantine environments.

Definition 5.5 (Algorithms executed by the mobile agents, labels and initial positions)**.** *Let $(e, \mathcal{A})$ be any execution (cf. Definition 2.13) with $e = (s, T, L, i, w, g, F, b)$ a Byzantine environment. There are $|L|$ mobile agents spread in s. Each of them is assigned a distinct label ℓ of L. If $\ell \in F$, the corresponding mobile agent is Byzantine. It executes $b(\ell)$ with ℓ as input. Otherwise, it executes $\mathcal{A}$, also with ℓ as input. For each mobile agent with label ℓ, the first position (cf. Definition 2.6) it occupies is $i(\ell)$ and is called its initial position.*

Definition 5.1 distinguishes two kinds of mobile agents: the good ones and the Byzantine ones. The former all execute the same mobile agent algorithm while the others are assumed to be controlled by the adversary, each with a specific algorithm. Remark that the Byzantine agents have the same abilities as the good ones, and that the good agents have no mean to distinguish Byzantine agents from good ones a priori. The cardinality of F i.e., the number of Byzantine agents is often denoted by f.

To fully understand the interactions between good and Byzantines agents, it is particularly relevant to take a look at Definitions 2.23, 2.24, and 2.25 (kept unchanged in this chapter). At any time t, any mobile agent (and thus Byzantine agent) can update the information it sends at most once. Moreover, it cannot choose to transmit its message to just a subset of the mobile agents within the range of its current position. Hence, at any time t, all good agents at a same position receive the same set of transmissions.

Remark that the global knowledge is the same for all good agents. It is a binary string which encodes some information about the whole environment but the behavior of the Byzantine agents. The latter is not included in the input of the oracle, which leaves it arbitrary and unpredictable by the good agents. In this chapter, an algorithm refers to a couple of an actual algorithm (sequence of instructions) and the oracle which computes the global knowledge. When describing the algorithm, it is necessary to explain what is the global knowledge.

This chapter addresses the task called Byzantine gathering more formally stated by Definition 5.6. The formal definition of the model variant studied is delayed to the next section.

Definition 5.6 (Byzantine gathering)**.** *Let M be any model variant (cf. Definition 2.12), $\mathcal{A}$ be any algorithm with advice. Algorithm $\mathcal{A}$ achieves Byzantine gathering in model variant M, if for every Byzantine environment e of M, in the execution $(e, \mathcal{A})$, there exists a time at which, all good agents are gathered at a same position and declare that the gathering is achieved.*

5.1.3 Contribution

As mentioned above, the existing deterministic algorithms dedicated to Byzantine gathering in finite graphs all have the major disadvantage of having a time complexity that is super-exponential in the number of nodes and the labels. Actually, these solutions are all based on a common strategy that consists in enumerating the possible initial configurations, and successively testing them one by one. Once the testing reaches the correct initial configuration, the gathering can be achieved. However, in order to get a significantly more efficient algorithm, such a costly strategy must be abandoned in favor of a completely new one.

This chapter addresses the design a deterministic solution for Byzantine gathering that makes a concession on the proportion of Byzantine agents within the team, but that offers a significantly lower duration. Another concern is using a global knowledge of short length. In this respect, assuming that the agents are in a *strong team* i.e., a team in which the number of good agents is

at least the quadratic value $5f^2+6f+2$, positive and negative results are given. On the positive side, an algorithm that solves Byzantine gathering with all strong teams in all graphs of size at most n, for any integers n and f, in a time polynomial in n and $|\ell_{\min}|$ is shown. The algorithm works using a global knowledge of size $\mathrm{O}(\log\log\log n)$, which is of optimal order of magnitude in this context to reach a time complexity that is polynomial in n and $|\ell_{\min}|$. Indeed, on the negative side, a proof that there is no deterministic algorithm solving Byzantine gathering with all strong teams in all graphs of size at most n, for any integers n and f, in a time polynomial in n and $|\ell_{\min}|$ and using a global knowledge of size $o(\log\log\log n)$ is provided.

The next definition states the model variant studied, in finite graphs, with synchronous settings, and in which the good agents are numerous enough to be a strong team.

Definition 5.7 (Model variant $\mathcal{BF}$)**.** *This model variant (cf. Definition 2.12) is the set of all Byzantine environments (s, T, L, i, w, g, F, b) such that $s = (P, Z, Q, d, c, r)$ is a finite graph (cf. Definition 2.8), T is the discrete timeline (cf. Definition 2.4), $|L| - |F| \geq 5|F|^2 + 6|F| + 2$, and g verifies the following property. Let t, q, and ℓ be any elements of T, Q, and L, respectively. The image by g of (t, q, ℓ) is* 1*.*

5.1.4 Roadmap

The next section is dedicated to the presentation of some basic definitions and routines needed in the rest of this chapter. Section 5.3 describes two building blocks that are used in turn in Section 5.4 to establish the positive result. In Section 5.5, the negative result is proved. Finally, some concluding remarks are made in Section 5.6.

5.2 Preliminaries

This chapter relies on several tools similar to those used in Chapter 3 and introduced in Section 3.2.

First of all, procedure `Explo`(i) is used in this chapter as well. Its time complexity is denoted X_i.

Besides this exploration procedure, a label transformation similar to that of Chapter 3 is used. Let ℓ_A be the label of an agent A and $(b_1 \ \dots \ b_{|\ell_A|})$ its binary representation with c its length. The binary representation of the corresponding doubled label $D(\ell_A)$ is $(1\ 0\ b_1\ b_1\ \dots\ b_{|\ell_A|}\ b_{|\ell_A|}\ 0\ 1\ 1\ 0\ b_1\ b_1\ \dots\ b_{|\ell_A|}\ b_{|\ell_A|}\ 0\ 1)$. This transformation is made to ensure the following property that is used in the proof of correctness the algorithm in Section 5.4.

Proposition 5.1. *Let ℓ_A and ℓ_B be two labels such that $\ell_A < \ell_B$. There exist two positive integers $i \leq \frac{|D(\ell_A)|}{2}$ and $\frac{|D(\ell_A)|}{2} < j \leq |D(\ell_A)|$ such that $D(\ell_A)[i] \neq D(\ell_B)[i]$ and $D(\ell_A)[j] \neq D(\ell_B)[j]$.*

Proof. There are two cases to consider: either $|\ell_A| = |\ell_B|$ or $|\ell_A| < |\ell_B|$. In the first case, since $\ell_A \neq \ell_B$, there exists a positive integer $i \leq |\ell_A|$ such that $\ell_A[i] \neq \ell_B[j]$. This implies in particular that $D(\ell_A)[2i+1] \neq D(\ell_B)[2i+1]$ and $D(\ell_A)[2|\ell_A| + 2i + 5] \neq D(\ell_B)[2|\ell_A| + 2i + 5]$. In the second case, either $D(\ell_A)[2|\ell_A| + 3] \neq D(\ell_B)[2|\ell_A| + 3]$ or $D(\ell_A)[2|\ell_A| + 4] \neq D(\ell_B)[2|\ell_A| + 4]$, and if $D(\ell_A)[2|\ell_A| + 5] = D(\ell_B)[2|\ell_A| + 5]$ then $D(\ell_A)[2|\ell_A| + 6] \neq D(\ell_B)[2|\ell_A| + 6]$. Hence, in each case the proposition holds. □

Throughout the chapter, some routines are designed in the form of a description of several states, where an agent has to apply specific rules, along with how to transit among them. In each round spent executing such a routine, each good agent tells its current state to the other agents sharing the same node. Sometimes, an agent is also required to tell extra information other than only its state: when such a situation arises, this point is obviously precised. Moreover, in the description of the states, the following expressions are used. The expression "agent A enters

state W" precisely means that at the previous round, agent A was in some state U $\neq$ W and at the current round, it is in state W. The expression "agent A exits state X" means that agent A remains in state X until the end of the current round and is in some state V $\neq$ X at the following round. Lastly, the expression "agent A transits from state Y to state Z" means that agent A exits state Y at the current round and enters state Z at the following one. Thus, in each round, agent A is always exactly in at most one state.

5.3 Building Blocks

To design the solution that is given in Section 5.4, the description of two prior subroutines which will be used as building blocks is required.

In the rest of this section, for each of the two building blocks, its high level idea, its detailed description, as well as its proof of correctness and cost analysis are given.

5.3.1 Procedure `Group`

The first building block called `Group` takes as input three integers $\mathcal{T}$, n and bin such that $bin \in \{0;1\}$. Let x be an integer that is at least $f+2$. Roughly speaking, subroutine `Group`$(\mathcal{T}, n, bin)$ ensures that $(x-f)$ good agents finish the execution of the subroutine at the same round and in the same node in a graph of size at most n provided the following two conditions are verified: the number of agents is at least $(x-1)(f+1)+1$, and all good agents start executing the subroutine in some interval lasting at most $\mathcal{T}$ rounds, with the same parameters except for the last one that has to be 0 (resp. 1) for at least one good agent. The time complexity of the procedure is polynomial in the first two parameters $\mathcal{T}$ and n.

5.3.1.1 High level idea

As mentioned previously, subroutine `Group` aims at ensuring that $x-f$ good agents finish the execution of the subroutine at the same round and in the same node. To achieve this, several difficulties have to be faced, especially the fact that the agents know neither x nor f, and also the fact that agents have a priori no mean to detect whether an agent is good or not. Indeed, instructions like "If there are at least $x-f$ good agents in my current node, then ..." cannot be used. Neither can "If there are at least x or f agents in my current node, then ..." no matter whether there are some Byzantine agents or not in the current node. So, to circumvent these problems, procedure `Group` is made of two phases. The first phase aims at ensuring that at least x agents executing the first phase meet in the same node (even though the involved agents do not detect this event). This phase lasts exactly the same time for each good agent and when it finishes it, a good agent is at the node from which it started executing it. The second phase consists, for a good agent, in replaying in the same order the same edge traversals and waiting periods made during the first i rounds of its first phase started at round t, such that $t+i$ is the round when the agent was with the maximum number of agents executing the first phase (if there are several such rounds, the latest one is chosen). Once this is done, the agent stops executing `Group`. By doing so, it is guaranteed that $x-f$ good agents (those involved in the last maximal meeting of the first phase) will stop executing the second phase (and thus procedure `Group`) in the same node and at the same round, as all the meetings involving the maximal number of agents in the first phase, necessarily involve at least x agents. Hence, the key of the procedure is to make x agents meet in the first phase.

During the execution of the first phase, the agents are partitioned into two distinct groups, namely *followers* and *searchers*. The first group corresponds to agents executing the subroutine with $bin=0$ and the second group corresponds to those executing it with $bin=1$.

The first phase works in steps $1, 2, \ldots, \mathcal{S}$ where $\mathcal{S}$ is some polynomial in $\mathcal{T}$ and n. At a very high level, in each step, the main role of followers is to remain idle in their initial starting

nodes in order to "mark" possible positions on which x agents could meet, while the main role of searchers is to look for these positions. To this end, each searcher will make use of a kind of map that it initially computes during the first step by making an entire traversal of the graph, using procedure `Explo`(n). Actually, this map corresponds to a sequence P of objects symbolizing every visited node v along with the list of labels of the agents that are (or pretend to be) followers present in node v at the time of the visit by the searcher. More precisely, the length of P is equal to the number of visited nodes in `Explo`(n), and the i-th object of P contains, among other information, the set of all followers' labels present in the i-th visited node of the traversal. Note that such a map will be called *imperfect map* as some nodes can be represented several times in the sequence P. Indeed `Explo`(n) guarantees that each node is visited at least once but some nodes may be visited more than once. The use of the qualifying term "imperfect" also stems from the fact that the list of followers' labels that are stored in P may be plagued by artificial ones created by Byzantine agents. In all the other steps, the searchers never recompute a new imperfect map, but always use the one computed in the first step, along with some possible updates on the lists of labels. How and when these updates are applied is explained below: they are obviously related to "bad behaviors" coming from Byzantine agents.

For the convenience of the explanation, let us first consider an ideal situation in which there is a unique follower among the good agents. If there is no Byzantine agent, during the first step, by moving to the node that hosts the unique follower, all searchers meet in the same node: using their maps, they are all able to determine a path to this follower. Thus, if the number of good agents is at least x, there is necessarily a round in which x agents meet in the same node. However, when Byzantine agents come into the picture, the problem becomes a tricky one, as these malicious agents can also pretend to have the status of followers (with the same label or not). Hence, all the searchers may not necessarily choose to move towards the same follower, which may prevent *in fine* the meeting of x agents. To deal with this issue and limit the confusion caused by Byzantine agents, in each step every good searcher A proceeds as follows. Let ℓ be the smallest label in sequence P (corresponding to the imperfect map of A) and let i be the first object of P in which ℓ appears. Agent A moves to the node u of the graph corresponding to the i-th object of P in order to meet again the follower with label ℓ and waits some prescribed amount of rounds with it at node u. If A does not see a follower with label ℓ when reaching u, or at some point during its waiting period at u it does not see anymore any follower with label ℓ, then agent A updates its imperfect map by removing ℓ from the list of the i-th object. Then, in all cases, the agent will end up starting the following step (if any) with its possibly updated map. The total number $\mathcal{S}$ of steps has been carefully chosen so that it is larger than the total number of map updates that can be made by all good agents in the network. Hence, the existence of a step in which there is no map update can be ensured: in such a step it can be proved that the number of different locations that are reached by searchers is at most $f+1$. Thus, if the number of good agents is at least $(x-1)(f+1)+1$, using arguments relying on the *pigeonhole principle*, the meeting of x agents can be proved. Keep in mind that all the above explanations are made under the assumption there is a single good follower in the team. When there are more than one good follower, things get more complicated. For instance, observe in this case that even though the number of good agents is at least $(x-1)(f+1)+1$, our approach, without additional precautions, may fail to make at least x agents meet on the same node as the number of good searchers may not be enough to ensure the meeting. Indeed, for a given number of agents, the more followers, the less searchers to distribute. However, through extra technical actions requiring sometimes some followers to end up behaving as a searcher, it is possible to overcome this issue and still ensure the meeting of x agents provided the cardinality of the set of good agents is at least $(x-1)(f+1)+1$.

5.3.1.2 Detailed description

To describe subroutine `Group`, a function called IM is used. It takes as input two integers n and $q \in \{0,1\}$, and returns an ordered sequence P of lists of labels: $P =< L_1, \ldots, L_{X_n} >$. The returned sequence P is called an *imperfect map*. When a given agent A performs $IM(q,n)$, it actually executes `Explo`(n) with some additional actions. At each step of `Explo`(n), depending on the value of q, A checks the presence of a given agent or a group of agents to compute P. During the first step, the agent is at the node from which it starts $IM(q,n)$. Let us consider the j^{th} step of `Explo`(n) ($j \in \{1, \ldots, X_n\}$) and let u be the node on which A is at this step. If $q = 0$, L_j is a list of pairwise distinct labels such that $\ell_B \in L_j$ if and only if there is on u an agent B with label ℓ_B being or pretending to be a follower. If $q = 1$, L_j is a list of pairwise distinct labels such that $\ell_B \in L_j$ if and only if there is on u an agent B with label ℓ_B being or pretending to be a follower in state `Wait-for-attendees`. State `Wait-for-attendees` is defined in the description of the algorithm. When ℓ_B is added to a list of P by A, A is said to record B. At the end of `Explo`(n), the agent traverses all the edges traversed in `Explo`(n) in the reverse order, and then it exits $IM(q,n)$.

To facilitate the presentation of the formal description of procedure `Group`, the following two definitions are needed.

Definition 5.8 (Useful map)**.** *An imperfect map P is said to be useful if and only if P contains a non-empty list.*

Definition 5.9 (Index of a map)**.** *Let $P =< L_1, \ldots, L_{X_n} >$ be a useful map. Let S be the set of every label that appears in at least one list of P. Let j be the smallest integer such that L_j contains the smallest label of S: j is the index of P.*

All requirements to give the formal description of the subroutine are met. This is the goal of the next paragraphs. Subroutine `Group`$(\mathcal{T}, n, bin)$ comprises two phases: **Process** and **Build-up**. Let us consider a given agent A executing `Group`$(\mathcal{T}, n, bin)$ from an initial node v. When $bin = 0$, the agent is said to be a *follower*. Otherwise, it is said to be a *searcher*. The description is in the form of several states along with rules to transit among them. At the beginning of each state, the agent is in its initial node v.

- **Phase Process**. Agent A proceeds in steps $1, 2, \ldots, \mathcal{S}$ where $\mathcal{S} = n^2.\mathcal{T}.X_n + 1$. Assume without loss of generality that A is at step $s \in \{1, \ldots, \mathcal{S}\}$. Unless stated explicitly, all the transitions between states which are presented below are performed within the same step. In all what follows $\mathcal{H} = (n+1)[\mathcal{T} + 4X_n + (X_n.n)(\mathcal{T}n + n)(2X_n + \mathcal{T})] + 3$. in the following, A's behavior is described depending on the value of bin.

  - $bin = 0$ (A is a follower). In this case, A can be in one of the following states: `Invite`, `Wait-for-attendees`, `Search-for-a-group` and `Follow-Up`. At the beginning of each step s, agent A is in state `Invite`. The actions to be performed in each state are presented in what follows.

    State `Invite` Agent A waits $2\mathcal{T} + 3X_n$ rounds. At the end of this waiting time, if A is on the same node as at least one searcher, A transits to state `Wait-for-attendees`. Otherwise, it transits to state `Search-for-a-group`.

    State `Wait-for-attendees` Agent A waits $2\mathcal{T} + X_n + \mathcal{H}$ rounds. If at each round of this waiting period, there is at least one searcher at node v, then at the end of the waiting period agent A transits to state `Follow-Up`. Otherwise, as soon as there is a round of the waiting period when there is no searcher at node v, agent A transits to state `Search-for-a-group` (hence the waiting period may be prematurely stopped).

State `Search-for-a-group` Let k be the number of rounds spent by agent A in state `Wait-for-attendees` of step s.

Note that $k = 0$ if A transited directly to state `Search-for-a-group` from state `Invite` in step s. Let w be a counter, the initial value of which is 0. The way this counter is incremented and decremented is explained below.

While agent A does not reach round $t + 2\mathcal{T} + X_n + \mathcal{H} - k$ where t is the round when it entered this state in step s, it proceeds as follows (thus, what follows is then interrupted when reaching round $t+2\mathcal{T}+X_n+\mathcal{H}-k$). Agent A first waits $\mathcal{T}$ rounds and then executes $IM(1, n)$. Once this is done, the agent has a map P. Each time P is useful (refer to Definition 5.8) and $w = 0$, the agent performs the first $i - 1$ edge traversals of `Explo`(n) from its initial node v where i is the index of P: just before each edge traversal, counter w is incremented by one. Let us refer to the node reached at the end of these $i-1$ edge traversals by u. As long as there is a follower B in state `Wait-for-attendees` on u such that ℓ_B is the smallest label in the i-th list L_i of P, A remains idle. By contrast, if there is no such follower on u in some round, agent A updates P by removing from L_i its smallest element and then goes back to its initial node v by performing the $(i-1)$ edge traversals executed above in the reverse order: just before each edge traversal of this backtrack, counter w is decremented by one.

As soon as agent reaches round $t+2\mathcal{T}+X_n+\mathcal{H}-k$, the agents proceeds as follows: if $w = 0$, it transits to state `Follow-Up`. Otherwise, if $w > 0$, A goes back to its initial node v by traversing in the reverse order the sequence of w edges $e_1, e_2, \ldots, e_k$ corresponding to the w first edge traversals of `Explo`(n) from node v: once this is done, it transits to state `Follow-Up`.

State `Follow-Up` Let x be the number of rounds elapsed from the beginning of the current step. Agent A waits $5\mathcal{T} + 5X_n + \mathcal{H} - x$ rounds. At the end of the waiting time, if $s < \mathcal{S}$, A transits to state `Invite` of step $s+1$. Otherwise, A transits to state `Restart` of phase **Build-up**.

– $bin = 1$ (A is a searcher). Agent A can be in one of the following states: `Search-for-an-invitation`, `Accept-an-invitation` and `Follow-Up`.

At the beginning of each step s, agent A is in state `Search-for-an-invitation`. What follows presents the set of actions to be performed for each state.

State `Search-for-an-invitation` Agent A first waits $\mathcal{T}$ rounds. Next, if $s = 1$ (first step of phase **Process**), A executes $IM(0, n)$ and then transits to state `Accept-an-invitation`. The output of the execution of $IM(0, n)$ is stored in variable Z. This variable may be updated in the current step as well as the following ones: each time this variable is mentioned, consider its up-to-date value. If $s > 1$, A waits $2X_n$ rounds and then transits to state `Accept-an-invitation`.

State `Accept-an-invitation` In the case where Z is not a useful map, A transits to state `Follow-Up`. Otherwise, let j and ℓ be the index of Z and the smallest label in the j-th list of Z respectively. Agent A performs the first $j - 1$ edge traversals of `Explo`(n). Let t be the round when agent A finishes these first $j - 1$ edge traversals, and let u be the node reached by A in round t. As soon as there is a round in $\{t + 1, t + 2, \ldots, t + 2\mathcal{T} + X_n + \mathcal{H}\}$ for which there is no follower B at node u such that label $\ell_B = \ell$, agent A updates P by removing ℓ from L_j and goes back to its initial node v by performing the $(j-1)$ edge traversals executed above in the reverse order. Once this backtrack is done, agent A transits to state `Follow-Up`.

If agent A is still in state `Accept-an-invitation` in round $t + 2\mathcal{T} + X_n + \mathcal{H}$, it goes back to its initial node v by performing the $(j-1)$ edge traversals executed above in the reverse order, and then it transits to state `Follow-Up` (note that in this latter case, Z remains unchanged).

State `Follow-Up` Let x be the number of rounds elapsed from the beginning of the current step. Agent A waits $5\mathcal{T} + 5X_n + \mathcal{H} - x$ rounds. At the end of the waiting time, if $s < \mathcal{S}$, then A transits to state `Search-for-an-invitation` of step $s + 1$. Otherwise, it transits to state `Restart` of phase **Build-up**.

- **Phase Build-up**. Agent A can only be in state `Restart`. *//At the beginning of this phase, the agent is at the node from which it started procedure* `Group` *i.e., node* v

  State `Restart` Let r be the round in which A initiated `Group` and let $r+i$ be the round in phase **Process** in which A is on a node containing the largest number of agents (including A itself) that are not in state `Restart`. If there are several such rounds, it chooses the one with the largest value i. Denote by r' the round in which the agent enters this state. From round r' to $r' + i - 1$, agent A replays exactly the same waiting periods and edges traversals from round r to $r + i - 1$. More precisely, for each integer y in $\{0, 1, \ldots, i-1\}$, if agent A remains idle (resp. leaves the current node via a port o) from round $r + y$ to round $r + y + 1$, then agent A remains idle (resp. leaves the current node via port o) from round $r' + y$ to $r' + y + 1$. In round $r' + i$, the agent stops the execution of `Group`.

5.3.1.3 Correctness and complexity analysis

Let $\mathcal{E}$ and Δ be respectively the set of all good agents in the network and the first round in which an agent of $\mathcal{E}$ starts executing `Group`$(\mathcal{T}, n, bin)$. Let x be an integer that is at least $f + 2$. To conduct the proof of correctness as well as the complexity analysis, several assumptions are made in the rest of this subsection: $|\mathcal{E}| \geq (x-1)(f+1) + 1$, every agent of $\mathcal{E}$ starts executing `Group`$(\mathcal{T}, n, bin)$ at round $\Delta + \mathcal{T} - 1$ at the latest, and at least one agent of A starts executing the procedure with $bin = 0$ (resp. $bin = 1$).

The following lemma is related to the duration of each step and the duration of phase Process. Recall that $\mathcal{S}$ and $\mathcal{H}$ are polynomials in n and $\mathcal{T}$ given in the detailed description of procedure `Group`.

Lemma 5.1. *Let A be an agent of $\mathcal{E}$. The following two properties are verified.*

1. *Each step of phase Process executed by A lasts exactly $5\mathcal{T} + 5X_n + \mathcal{H}$ rounds.*
2. *The execution of phase Process by agent A lasts exactly $\mathcal{S}(5\mathcal{T} + 5X_n + \mathcal{H})$ rounds.*

Proof. According to the algorithm, $\mathcal{S}$ corresponds to the number of steps in phase Process. So, if the first property holds, the second one also holds. Hence, to prove the lemma, it is enough to prove that the first property is true: this will be the purpose of the rest of this proof.

Let s be a step of phase Process executed by agent A. Let us first prove that A transits to state `Follow-Up` of step s after having spent at most $4\mathcal{T}+5X_n+\mathcal{H}$ rounds in this step. Depending on the value of bin, A can be either a searcher or a follower. Two cases are considered.

- A is a follower. The state of A in the first round of every step s of phase Process during the execution of `Group` is `Invite`. Agent A spends $2\mathcal{T}+3X_n$ rounds in state `Invite` before transiting to either state `Wait-for-attendees` or state `Search-for-a-group` depending on whether there is a searcher on the same node as A at the end of this waiting time. Agent A remains in either state `Wait-for-attendees` or `Search-for-a-group` at most $2\mathcal{T} + 2X_n + \mathcal{H}$ rounds before transiting to state `Follow-Up`. Hence, agent A spends at most $4\mathcal{T} + 5X_n + \mathcal{H}$ in step s before transiting to state `Follow-Up`.

- A is a searcher. The state of A in the first round of `Group` is `Search-for-an-invitation`. First, agent A waits $\mathcal{T}$ rounds. Next, if $s = 1$, A executes $IM(0, n)$ that lasts $2X_n$ rounds before transiting to state `Accept-an-invitation`. Otherwise $s > 1$ and A waits $2X_n$ rounds before transiting to state `Accept-an-invitation`. That is, in both cases, A spends $\mathcal{T} + 2X_n$ in total before transiting to state `Accept-an-invitation`. Once A transits to state `Accept-an-invitation`, if Z, the output of $IM(0, n)$ performed while in state `Search-for-an-invitation` in the first step of phase Process, is not a useful map, A transits to state `Follow-Up` and the lemma holds. If by contrast, Z is useful then A performs the first $(j - 1)$ edge traversals of `Explo`(n) where j is the index of Z. Agent A waits at most $2\mathcal{T} + X_n + \mathcal{H}$ rounds (with a follower) and then performs less than X_n edge traversals to retrieve its initial position before transiting to state `Follow-Up`. So, agent A spends at most $3\mathcal{T} + 5X_n + \mathcal{H}$ in step s before transiting to state `Follow-Up`.

Hence, whether A is a follower or not, it spends at most $x \leq 4\mathcal{T} + 5X_n + \mathcal{H}$ rounds in step s before transiting to state `Follow-Up` of step s. However, according to state `Follow-Up`, the agent waits exactly $5\mathcal{T} + 5X_n + \mathcal{H} - x$ rounds before leaving step s. Hence, the lemma holds. □

The following statement is a corollary of the previous lemma.

Corollary 5.1. *Let A and B be any two good agents of $\mathcal{E}$ such that $t_A - t_B \geq 0$, where t_A (resp. t_B) is the round when A (resp. B) starts executing* `Group`*. For every step s of phase Process, agent A finishes executing s, exactly $t_A - t_B$ rounds after B finishes executing it.*

In the following, "initial node" refers to the node from which the agent starts executing procedure `Group`. Note that the statement of Lemma 5.2 calls for the notion of "recording" that is introduced in the description of function IM.

Lemma 5.2. *Let A be a good searcher of $\mathcal{E}$. For every follower B in $\mathcal{E}$, agent A records B during its execution of $IM(0, n)$ when B is on its initial node.*

Proof. According to the algorithm, agent A executes $IM(0, n)$ while in state `Search-for-an-invitation` of step 1. More precisely, when starting step 1, agent A first waits $\mathcal{T}$ rounds and then executes $IM(0, n)$ that lasts $2X_n$ rounds. On the other hand, agent B waits $2\mathcal{T} + 3X_n$ rounds in its initial node at the beginning of step 1. Hence, in view of the initial delay $\mathcal{T}$, the lemma follows. □

To continue, the definition of *target node* is introduced. A node u is said to be a target node of a good searcher A in a step $s > 1$, if u is the node that is reached after performing the first $(j-1)$ edge traversals of `Explo`(n) from the initial node of A and j is the index of the imperfect map of A at the beginning of its execution of step s.

Lemma 5.3. *Let A be a searcher of set $\mathcal{E}$ starting a step s with a useful map P, the index of which is j. Let ℓ be the smallest label in the j-th list of P. If the target node of A in step s is the initial node of a good follower B such that $\ell_B = \ell$, then A does not update P in any step $s' \geq s$.*

Proof. The proof of this lemma consists in proving by induction on $i \geq 0$ that A does not update P in step $s + i$. First consider the initial step in which $i = 0$. Assume by contradiction that the target node of A in step s is the initial node u of a good follower B such that $\ell_B = \ell$, but A updates P in step s. According to procedure `Group`, agent A reaches node u while in state `Accept-an-invitation`. Then, agent A updates P in step s only if agent A does not meet agent B when reaching target node u or A notices the absence of B on u within $2\mathcal{T} + X_n + \mathcal{H}$ rounds after its meeting with B. However, when agent B starts step s, it first waits $2\mathcal{T} + 3X_n$ in state `Invite`. Hence in view of Corollary 5.1 and the definition of $\mathcal{T}$, agent A meets B when reaching target node u while B is in state `Invite`: indeed agent A spends $\mathcal{T} + 2X_n$ rounds

in state `Search-for-an-invitation` and at most X_n rounds in state `Accept-an-invitation` before reaching its target node u at some round t. Moreover, since agent A and B are good, according to states `Wait-for-attendees` and `Accept-an-invitation` agent A remains with B at node u at least $2\mathcal{T} + X_n + \mathcal{H}$ rounds after round t. As a result, agent A does not update P in step s, which is a contradiction and proves the first step of the induction. Now consider there exits a positive integer i' such that the property holds for all $i \leq i'$. If the last step of procedure `Group` is step $s + i'$, the lemma directly follows. Otherwise, note that agent A begins step $s + i' + 1$ with the exact same map as in step $s + i'$. Hence using the same arguments as in step $s + i'$, agent A does not update P in step $s + i' + 1$. This closes the induction and proves the lemma. □

Note that in view of Lemmas 5.2 and 5.3, the imperfect map of every searcher of $\mathcal{E}$ remains always useful. In other terms, each of them always have a target node in every step of procedure `Group`. This is stated in the following proposition.

Proposition 5.2. *The map of every searcher of $\mathcal{E}$ is always useful.*

In order to prove the main result of this section, i.e., Theorem 5.1, the next three lemmas are necessary.

Lemma 5.4. *If $f < n$, then there exists an integer s in $\{1, \ldots, \mathcal{S}\}$ such that no searcher in set $\mathcal{E}$ updates its imperfect map P during its execution of step s.*

Proof. Assume for the sake of a contradiction that for each s in $\{1, \ldots, \mathcal{S}\}$ there is at least one searcher of $\mathcal{E}$ that updates the output of its imperfect map P during its execution of step s. According to procedure `Group`, every searcher A in $\mathcal{E}$ executes $IM(0, n)$ to compute P. When A performs $IM(0, n)$, A records all the followers it meets during the execution of `Explo`(n) in $IM(0, n)$. In particular, for each visited node A can record at most f Byzantine agents. This leads to at most f "wrong" labels in each list of P. Since $f < n$, in view of Lemma 5.3, each searcher of $\mathcal{E}$ performs at most $n.X_n$ updates of P. Note that, two distinct searchers of $\mathcal{E}$ which start executing procedure `Group` from the same node and at the same round act exactly in the same manner: in particular, they traverse the same edges synchronously, compute the same imperfect map and make the same updates at the same time. Hence, taking into account the maximum delay $\mathcal{T}$, the number of rounds in which there is a searcher of $\mathcal{E}$ making an update of its imperfect map is upper-bounded by $\mathcal{U} = \mathcal{T}n^2X_n$. However, according to the algorithm $S = \mathcal{U} + 1$. This is a contradiction, which proves the lemma. □

In view of Lemma 5.4, s_{min} is defined as being the first step for which there is no updates made by a searcher of $\mathcal{E}$.

Lemma 5.5. *If $f < n$, then there exist a round α and a node v such that x agents meet on node v at round α, and a searcher of $\mathcal{E}$ is in state* `Accept-an-invitation` *at round α.*

Proof. In order to prove the lemma, a series of 4 claims are proved first. The following notations will facilitate the conduct of this proof.

Let $\mathcal{Q}$ be the set of nodes verifying the following condition: a node u is in $\mathcal{Q}$ if u is a target node of a searcher of $\mathcal{E}$ in step s_{min}. In view of Proposition 5.2, $\mathcal{Q} \neq \emptyset$. Let $\mathcal{F}_\mathcal{Q}$ be the set of followers of $\mathcal{E}$ being on a node of $\mathcal{Q}$ at the beginning of their execution of step s_{min}. Let ρ be the last round in which a follower of $\mathcal{E}$ is in state `Invite` before entering either state `Search-for-a-group` or state `Wait-for-attendees` (at round $\rho + 1$) during its execution of step s_{min}.

Claim 5.1. *At round ρ, every searcher A of $\mathcal{E}$ is on its target node. Moreover, A remains on its target node for at least $\mathcal{H}$ rounds after round ρ.*

Proof of the claim: According to procedure `Group` and the maximal delay $\mathcal{T}$, at round ρ every searcher has spent in step s_{min} at least $\mathcal{T} + 3X_n$ rounds and at most $3\mathcal{T} + 3X_n$ rounds. Moreover, in view of the definition of step s_{min} and Proposition 5.2, every searcher remains in its target node at least $2\mathcal{T} + X_n + \mathcal{H}$ rounds while in state `Accept-an-invitation` of step s_{min}. However, before entering state `Accept-an-invitation` of step s_{min}, each searcher spends at least $\mathcal{T} + 2X_n$ rounds and at most $\mathcal{T} + 3X_n$ in step s_{min}. Hence the claim follows. ⋆

Claim 5.2. *Let B be a follower of $\mathcal{F}_{\mathcal{Q}}$. Agent B remains idle in state* `Wait-for-attendees` *on its initial node from round $\rho + 1$ to round $\rho + \mathcal{H}$.*

Proof of the claim: Let u be the initial node of B and ρ' the last round in which it is in state `Invite` of step s_{min}. Since u is a target node of a searcher A of $\mathcal{E}$, agent A reaches u after having spent at least $\mathcal{T} + 2X_n$ rounds and at most $\mathcal{T} + 3X_n$ rounds in step s_{min}. Since B waits $2\mathcal{T} + 3X_n$ in state `Invite` at the beginning of step s_{min}, in view of the maximum delay between any pair of agents of $\mathcal{E}$, A reaches node u while B is still in state `Invite`. Moreover, by definition of step s_{min}, A remains on u during $2\mathcal{T} + X_n + \mathcal{H}$ rounds (in state `Accept-an-invitation`). Hence according to procedure `Group`, at round ρ' agent B has shared its initial node with agent A for at most $2\mathcal{T} + X_n$ rounds and it enters state `Wait-for-attendees` at round $\rho' + 1$. So, after ρ', agent A stays idle with B for at least $\mathcal{H}$ rounds. This means in particular that B is in state `Wait-for-attendees` from round $\rho' + 1$ to $\rho' + \mathcal{H}$.

Let $diff = \rho - \rho'$. Note that $0 \leq diff \leq \mathcal{T}$. According to the description of state `Wait-for-attendees`, from round $\rho' + \mathcal{H}$ to $\rho' + \mathcal{H} + diff$, agent B leaves state `Wait-for-attendees` only if A leaves u at some round in $\{\rho' + \mathcal{H}, \ldots, \rho' + \mathcal{H} + diff\}$. However, this is impossible according to Claim 1 and the fact that $\rho' \in \{\rho - \mathcal{T} + 1; \rho - \mathcal{T} + 2, \ldots, \rho\}$: indeed $\rho' + \mathcal{H} > \rho + 1$ and $\rho' + \mathcal{H} + diff = \rho + \mathcal{H}$. Hence agent B remains in `Wait-for-attendees` from round $\rho' + 1$ to round $\rho' + \mathcal{H} + diff$, which proves the claim. ⋆

Claim 5.3. *Among the nodes of $\mathcal{Q}$, at least $|\mathcal{Q}| - 1$ of them host a Byzantine agent in every round from round $\rho + 1$ to round $\rho + \mathcal{H}$.*

Proof of the claim: Let $\mathcal{I}$ be the time interval between round $\rho + 1$ and round $\rho + \mathcal{H}$. This proof consists in showing that during $\mathcal{I}$, at least $|\mathcal{Q}| - 1$ nodes of $\mathcal{Q}$ host a Byzantine agent. Let B be the first follower of $\mathcal{E}$ that starts the execution of `Group`: if there are several agents satisfying the condition, the one with the smallest label is chosen. Let us denote by Δ_B the round in which B starts the execution of `Group`. From Claim 1, during time interval $\mathcal{I}$, every searcher is on its target node. That is, there are $|\mathcal{Q}|$ distinct target nodes for the searchers of $\mathcal{E}$. Hence, from procedure `Group` and Lemmas 5.2 and 5.3, it follows that on each node of Q there is at least one agent B' being (or pretending to be) a follower such that its label is at most ℓ_B. According to the definition of B, and in particular its unicity, at least $|\mathcal{Q}| - 1$ target nodes host a Byzantine agent from round $\rho + 1$ to $\rho + \mathcal{H}$. Hence the claim holds. ⋆

Let $\mathcal{X} = \mathcal{T} + 4X_n + (nX_n)(\mathcal{T}n + n)(2X_n + \mathcal{T})$. Note that $\mathcal{H} = (n + 1)\mathcal{X} + 3$.

Claim 5.4. *Let $\rho + 1 \leq \nu \leq \rho + \mathcal{H} - \mathcal{X}$ be a round, if any, such that no good follower of $\mathcal{E}$ enters state* `Search-for-a-group` *from round ν to round $\nu + \mathcal{X} - 1$. At least x agents meet at some round in $\{\nu + 1, \ldots, \nu + \mathcal{X}\}$.*

Proof of the claim: Let $\mathcal{F}_{\mathcal{Q}'}$ be the set of followers of $\mathcal{E}$ that do not belong to $\mathcal{F}_{\mathcal{Q}}$ and do not enter state `Search-for-a-group` of step s_{min} by round $v - 1$. Let $\mathcal{F}_{\mathcal{Q}''}$ be the set of followers of $\mathcal{E}$ that do not belong to $\mathcal{F}_{\mathcal{Q}}$ and enter state `Search-for-a-group` of step s_{min} by round $v - 1$. Let $\mathcal{Q}'$ be the set of initial nodes of agents in $\mathcal{F}_{\mathcal{Q}'}$. Note that every good follower belongs to $\mathcal{F}_{\mathcal{Q}} \cup \mathcal{F}_{\mathcal{Q}'} \cup \mathcal{F}_{\mathcal{Q}''}$.

Let B be a follower of $\mathcal{F}_{\mathcal{Q}''}$. The first step of this proof consists in showing that the map of B, when it is computed, is always useful in step s_{min} till round $\rho + \mathcal{H}$ included. Note that in view of Claim 1, it is enough to prove that agent B starts and finishes the execution of $IM(1, n)$ in $\{\rho + 1, \ldots, \rho + \mathcal{H}\}$. In view of the definition of ρ and Corollary 5.1, B enters state

`Search-for-a-group` at some round in $\{\rho-\mathcal{T}+1,\ldots,\nu-1\}$. Moreover, when a follower enters this state, it first waits $\mathcal{T}$ before executing $IM(1,n)$ that lasts $2X_n$ rounds. Hence B starts and finishes $IM(1,n)$ in $\{\rho+1,\ldots,\nu+\mathcal{T}+2X_n\}$. However, $\nu+\mathcal{T}+2X_n \leq \rho+\mathcal{H}$, which proves that the map of B, when it is computed, remains always useful in $\{\rho+1,\ldots,\rho+\mathcal{H}\}$.

As mentioned above, at round $\nu+\mathcal{T}+2X_n$, every follower of $\mathcal{F}_{\mathcal{Q}''}$ has completed its execution of $IM(1,n)$. Observe that when a good follower B transits to state `Search-for-a-group` from state `Wait-for-attendees` on a node u at some given round w between round $\rho+1$ and round $\rho+\mathcal{H}$, every good follower on u also transits to state `Search-for-a-group` from state `Wait-for-attendees` at round w: moreover, these good followers behave in a same synchronous manner i.e., they execute the same actions in each round between w to round $\rho+\mathcal{H}$. That is, the total number of distinct maps of the agents of $\mathcal{F}_{\mathcal{Q}''}$ at round $\nu+\mathcal{T}+2X_n$ is at most $(\mathcal{T}.n+n)$: there are at most $\mathcal{T}n$ distinct maps of the good followers that transit to state `Search-for-a-group` from either state `Invite` or state `Wait-for-attendees` before round $\rho+1$ and at most n additional distinct maps of the good followers that transit from state `Wait-for-attendees` to state `Search-for-a-group` after round ρ.

Next, assume that there exists a round α' such that $\nu+\mathcal{T}+2X_n \leq \alpha' \leq \nu+\mathcal{X}-2X_n$ and no good follower of $\mathcal{F}_{\mathcal{Q}''}$ updates its imperfect map from round α' to round $\alpha'+2X_n$. In this case, the aim is to show that x agents meet in the same node at some round in $\{\alpha',\ldots,\alpha'+2X_n\}$. Let B, P and j be respectively a follower of $\mathcal{F}_{\mathcal{Q}''}$, the imperfect map of B and its index from round α' to round $\alpha'+2X_n$. The target node of B is the node that is reached after performing the first $(j-1)$ edge traversals of `Explo`(n) from the initial node of B. Agent B updates its imperfect map P only if on its target node, there is no follower B' such that $\ell_{B'}$ is the smallest label in L_j of P. Since there are no updates from round α' to round $\alpha'+2X_n$, at round $\alpha'+2X_n$, every follower B of $\mathcal{F}_{\mathcal{Q}''}$ is on its target node u.

In the case where u is neither in $\mathcal{Q}$ nor $\mathcal{Q}'$, the aim is to show that u hosts at least one Byzantine agent. From procedure `Group`, at round $\alpha'+2X_n$, node u hosts a follower B' such that $\ell_{B'}$ is the smallest label in L_j of P. If B' is a good follower, B' is in state `Wait-for-attendees` with a searcher A (recall that no good follower transits to state `Search-for-a-group` from round ν to round $\nu+\mathcal{X}-1$). However, A cannot be a good searcher of $\mathcal{E}$ since u is not in $\mathcal{Q}$. Hence, u hosts indeed a Byzantine agent at round $\alpha'+2X_n$. Note that in view of the definition of ν and the algorithm, each agent of $\mathcal{F}_{\mathcal{Q}'}$ is on its initial node with a Byzantine agent pretending to be a searcher from round $\rho+1$ to $\nu+\mathcal{X}-1$ (as all the good searchers are in nodes $\notin \mathcal{Q}'$ according to Claim 1). Let $\mathcal{Q}''$ be the target nodes which do not belong to $\mathcal{Q}\cup\mathcal{Q}'$, of the good followers of $\mathcal{F}_{\mathcal{Q}''}$ at round $\alpha'+2X_n$. Then, by Claim 3, $|\mathcal{Q}|+|\mathcal{Q}'|+|\mathcal{Q}''| \leq f+1$. Moreover, at round $\alpha'+2X_n$, every good agent is in a node of $\mathcal{Q}\cup\mathcal{Q}'\cup\mathcal{Q}''$. Hence by the Pigeonhole principle, it follows that x agents share the same node at round $\alpha'+2X_n$. If round α' exists, then the claim holds. So to conclude the proof of this claim, it remains to show the existence of round α'. Recall that each follower of $\mathcal{F}_{\mathcal{Q}''}$ performs at most $X_n.n$ updates of its imperfect map $\mathcal{P}$ (since it can record at most f Byzantine agents that pretend to be followers in state `Wait-for-attendees` on each node during the execution of $IM(1,n)$). Besides, as argued earlier, the total number of distinct maps of the agents of $\mathcal{F}_{\mathcal{Q}''}$ at round $\nu+\mathcal{T}+2X_n$ is at most $(\mathcal{T}.n+n)$. So, after at most $(\mathcal{T}.n+n).(X_n.n)(\mathcal{T}+2X_n = \mathcal{X}-\mathcal{T}-4X_n$ rounds from $\nu+\mathcal{T}+2X_n$, no good follower of $\mathcal{F}_{\mathcal{Q}''}$ updates its imperfect map. Moreover, every good follower of $\mathcal{F}_{\mathcal{Q}''}$ spends at most $2X_n$ rounds before reaching its target node. This proves the existence of round α' and by extension the claim. ⋆

What follows builds on the claims to prove the lemma. Assume by contradiction that the lemma does not hold. This means either there is no round when x agents meet, or in every round z when x agent meet, no searcher of $\mathcal{E}$ is in state `Accept-an-invitation` at round z. Let us first consider the former case. Let $\mathcal{F}'$ be the set of good followers that enter state `Search-for-a-group` from state `Wait-for-attendees` at some round in $\{\rho+1,\ldots,\rho+\mathcal{H}\}$. From Claim 4, there is no consecutive $\mathcal{X}$ rounds in $\{\rho+1,\ldots,\rho+\mathcal{H}-\mathcal{X}\}$ in which no good follower of $\mathcal{E}$ transits to

state `Search-for-a-group` (otherwise, round α, which is defined in the statement of this lemma, exits). From round $\rho+2$ to $\rho+\mathcal{H}$, only the followers of $\mathcal{F}'$ may enter state `Search-for-a-group`. From round $\rho+2$ all the agents of $\mathcal{F}'$ have already entered state `Wait-for-attendees` in view of the definition of ρ. Note that $|\mathcal{Q}_{\mathcal{F}'}| \leq n$ where $|\mathcal{Q}_{\mathcal{F}'}|$ is the set of initial nodes of at least one follower of $\mathcal{F}'$. Moreover, let C be an agent of $\mathcal{F}'$ that enters state `Search-for-a-group` from state `Wait-for-attendees` at a round $t \in \{\rho+2, \ldots, \rho+\mathcal{H}\}$: before round t, agent C does not move in step s_{min}, and all the agents of $\mathcal{F}'$ that are in state `Wait-for-attendees` and share the same node as C in round $t-1$ also enter state `Search-for-a-group` at round t. Hence, after at most $n\mathcal{X}$ rounds from round $\rho+2$, there is no agent that can enter state `Search-for-a-group` till round $\rho+\mathcal{H}$ included. However round $\rho+3+n\mathcal{X} \leq \rho+\mathcal{H}-\mathcal{X}$. Hence there exists a round v satisfying the statement of Claim 4 and there is a meeting of at least x agents at some round in $\{\nu+1, \ldots, \nu+\mathcal{X}\}$: this is a contradiction with the fact that α does not exist. Concerning the latter case, note that there is a round α in $\{\nu+1, \ldots, \nu+\mathcal{X}\}$ in which x agents meet. In view of Claim 1 and procedure `Group`, every searcher of $\mathcal{E}$ is in state `Accept-an-invitation` in every round belonging to $\{\nu+1, \ldots, \nu+\mathcal{X}\}$: this contradicts the fact that no searcher of $\mathcal{E}$ is in state `Accept-an-invitation` at round α. □

Lemma 5.6. *If there exists a round r at which at least $x \geq f+2$ agents meet on the same node and among them all the good ones are executing phase Process at round r, then at least $(x-f)$ good agents exit their execution of* `Group` *at the same round and on the same node.*

Proof. Assume there exists such a round. Let us show that $(x-f)$ good agents exit their execution of `Group` at the same round and on the same node. Let x' be the largest number of agents executing `Group` but not in state `Restart` which met in the same node u in some round $\Delta+w$. If there are several such rounds, consider the one with the largest value of w. The good agents executing `Group` but in another state than `Restart` are precisely those executing phase Process, which implies that $x' \geq x$. Let $\mathcal{Y}$ be the set of good agents executing phase Process on u at round $\Delta+w$. Remark that at round $\Delta+w$ on u there are at most f Byzantine agents. Hence, $|\mathcal{Y}| \geq x'-f$.

When in state `Restart`, every agent A of $\mathcal{Y}$ repeats exactly the same waiting periods and edge traversals as in its execution phase Process in order to reconstruct the group of agents that was at node u in round $\Delta+w$. More precisely, let r and r' be the round when A initiated `Group` and the round when A enters state `Restart` respectively. Let i be an integer such that $r+i = \Delta+w$. From round r' to $r'+i-1$, agent A replays exactly the same waiting periods and edges traversals from round r to $r+i-1$: for each integer y in $\{0, 1, \ldots, i-1\}$, if agent A remains idle (resp. leaves the current node via a port o) from round $r+y$ to round $r+y+1$, then agent A remains idle (resp. leaves the current node via port o) from round $r'+y$ to $r'+y+1$. In round $r'+i$, agent A is in node u and stops the execution of `Group`. Besides, in view of Lemma 5.1, every good agent spends the same number of rounds executing phase Process: let us denote this number by $\mathcal{W}$. So, $r'+i = r+\mathcal{W}+i = \Delta+w+\mathcal{W}$. Hence, every agent of $\mathcal{Y}$ is in node u and stops the execution of `Group` at round $\Delta+w+\mathcal{W}$. □

The next statement is the main theorem related to procedure `Group` which ends this subsection. In order to use the theorem outside of this subsection, the assumptions made in the beginning of this subsection are recalled in the statement.

Theorem 5.1. *Consider a team made of at least $(x-1)(f+1)+1$ good agents in a graph of size at most n, where $x \geq f+2$. Let Δ be the first round when a good agent starts executing* `Group`$(\mathcal{T}, n, bin)$*. If all good agents start executing* `Group`$(\mathcal{T}, n, bin)$ *by round $\Delta+\mathcal{T}-1$, and parameter bin is 0 (resp. 1) for at least one good agent, then the following property is verified. After at most a time polynomial in n and $\mathcal{T}$ from Δ, at least $(x-f)$ good agents finish the execution of* `Group` *at the same round and in the same node.*

Proof. When in state `Restart`, an agent only replays all or part of the waiting periods and edge traversals made in phase Process. Hence, according to Lemma 5.1 and the initial delay that is at most $\mathcal{T}$, every good agent finishes the execution of `Group` after at most a time polynomial in n and $\mathcal{T}$ from Δ.

So to prove the theorem it remains just to show that there is a group of at least $(x - f)$ good agents that exit `Group` on the same node and at the same time. This follows directly from Lemma 5.6 and the claim that is proven below.

Claim 1. At least x agents meet on the same node at some round t, and among them all the good ones are executing phase Process of procedure `Group` at round t.

Proof of Claim 1. If $f \geq n$, there are always x agents sharing the same node as the number of good agent is at least $(f + 1)x$. Moreover, at round $\Delta + \mathcal{T}$ every good agent is executing phase process of procedure `Group`. Hence, the claim holds if $f \geq n$.

So let us focus on the case where $f < n$. From Lemma 5.5, there is a round α when x agents meet in some node v and there is a good searcher A in state `Accept-an-invitation` of some step s in round α. At round α, it remains for agent A at least $\mathcal{T}$ rounds to spend in step s. Indeed, in state `Follow-Up` of step s, an agent has to wait $5\mathcal{T} + 5X_n + \mathcal{H} - x$ rounds and x is upper-bounded by $4\mathcal{T} + 5X_n + \mathcal{H}$ (this is shown in the proof of Lemma 5.1). Hence, in view of Corollary 5.1, no good agent has finished step s of phase Process at round α. Moreover, agent A has necessarily spent more than $\mathcal{T}$ rounds in step s when in round α. So, every good agent is executing phase Process of procedure `Group` at round α, which proves the claim. □

5.3.2 Procedure `Merge`

The second building block called `Merge` takes as input two integers n and $\mathcal{T}$. Subroutine `Merge`$(\mathcal{T}, n)$ allows all the good agents to finish their executions of the subroutine in the same node and at the same round, provided the following two conditions are satisfied. The first condition is that all good agents are in a graph of size at most n and start executing `Merge`$(\mathcal{T}, n)$ in an interval lasting at most $\mathcal{T}$ rounds. The second condition is that at least $4f+2$ good agents start executing `Merge`$(\mathcal{T}, n)$ at the same round and in the same node. The time complexity of the procedure is polynomial in $\mathcal{T}$ and n.

5.3.2.1 High level idea

For the sake of convenience, consider in this section that a group of agents is a set of all agents, at least one of which is good, that start executing procedure `Merge` in the same node and at the same round. In the sequel, assume there is a group of at least $4f + 2$ good agents. The reasons why such an assumption is necessary will appear at the end of the explanations. Let G_{max} and v_{max} be respectively the group with the largest initial number of agents and its starting node. In case there are several possible groups G_{max}, the one having the largest lexicographically ordered list of pairwise distinct labels denoted by L_{max} is chosen: this guarantees the unicity of G_{max} as it contains at least $4f + 2$ good agents. The cardinality of a list L will be denoted by $|L|$.

The idea underlying procedure `Merge` is to make all good agents elect the same node, and then gather in it (if this is ensured, then it can be guaranteed that all good agents finish the execution of `Merge` at a same round using some technicalities). Each node is a candidate, and each good agent supports the node in which it started executing the procedure. Besides supporting its candidate, each good agent is also a voter. When acting as a supporter, a good agent stays idle to promote its candidate and when acting as a voter, it makes a traversal of the graph in order to visit all nodes of the graph (using procedure `Explo`(n)), and then elects one of the nodes using the information provided by the supporters. In order to establish such a strategy, note that all good agents must not act as voters at the same time. Otherwise, there would be no supporter left in its candidate node to promote it. Hence, the election process is divided into two parts, and each group is divided into two subgroups of nearly equal size using

the labels of the agents. During the first (resp. second) part of the election, the first (resp. second) subgroup of each group acts as voters while the second (resp. first) subgroup of each group acts as supporters.

When visiting a node during its traversal of the graph, a voter gets from each supporter of this node a promotional information: for a good supporter, it is simply the lexicographically ordered list of all pairwise distinct labels of the agents that were initially in its group. Once its traversal is done, the voter considers each node v satisfying the property that at least $\lceil \frac{|L|}{4} \rceil$ distinct agents in v have transmitted a lexicographically ordered list L. Among these nodes, the voter elects the one for which the property is true with the list L having the largest cardinality: in case of a tie, the lexicographical order on the labels is used as done to ensure the unicity of G_{max}. By doing so, all good agents elect node v_{max} and then gather in it: the purpose of the last paragraph is to explain why v_{max} is unanimously elected.

By definition, the number of good agents that is initially in G_{max}, and thus $|L_{max}|$ is at least $4f + 2$. Moreover, the number of Byzantine agents is initially at most f in G_{max}. Hence, it can be shown than this strategy permits to always have at least $\lceil \frac{|L_{max}|}{4} \rceil$ distinct agents in v_{max} that transmit list L_{max} to all voters. Note that each good supporter transmits a list L such that $|L| < |L_{max}|$, or $|L| = |L_{max}|$ and L is not lexicographically larger than L_{max}. So, the only way the Byzantine agents could prevent the good agents to elect v_{max} would be that at least $\lceil \frac{|L'|}{4} \rceil$ Byzantine agents transmit a list L' such that $|L'| > |L_{max}|$, or $|L'| = |L_{max}|$ and L' is lexicographically larger than L_{max}. However this situation is impossible because the Byzantine agents are not numerous enough: indeed $\lceil \frac{|L'|}{4} \rceil \geq f + 1$.

5.3.2.2 Formal description of the algorithm

When an agent A executes `Merge`$(\mathcal{T}, n)$, it can transit to different states that are `Census`, `Election` and `Synchronization`. When agent A starts the execution of `Merge`, it is in state `Census`. In the algorithm, the cardinality of a list L will be denoted by $|L|$.

State `Census` Agent A spends a single round in this state. Besides its state, it transmits its label to the agents sharing the same node. Agent A assigns to variable H, the lexicographically ordered list of all pairwise distinct labels of agents that are currently in its node and in state `Census`. Then A transits to state `Election`.

State `Election` When it enters this state, agent A initializes two variables: it assigns an empty list to variable I, and 0 to variable pi. This state is made of five different periods: the first, third and fifth (resp. the second and fourth) ones are waiting periods (resp. moving periods). In each round of the two first waiting periods, agent A transmits the list H built when in state `Census`. If ell_A belongs to the first $\lfloor \frac{|H|}{2} \rfloor$ labels of H, then the durations of the two first waiting periods are respectively $\mathcal{T} - 1$ and $\mathcal{T} + 2X_n - 1$. Otherwise, they respectively last $\mathcal{T} + 2X_n - 1$ and $\mathcal{T} - 1$ rounds. The duration of the third waiting period is given after describing the second moving period.

During the first moving period, agent A executes `Explo`(n) followed by a backtrack in which the agent traverses all edges traversed in `Explo`(n) in the reverse order. Once this backtrack is done, the agent assigns to variable I the largest list I_1, if any, having the following property: there is a round during the execution of `Explo`(n) at which agent A is in a node where at least $\lceil \frac{|I_1|}{4} \rceil$ distinct agents in state `Election` transmit I_1. (A list I_2 is larger than another list I_3 if and only if I_2 contains more elements, or I_2 and I_3 contain the same number of elements and I_2 is lexicographically larger than I_3). If such a list I_1 exists, the agent also assigns to variable pi, the smallest number of edge traversals made by A during the execution of `Explo`(n) to reach a node satisfying the above property with I_1. Otherwise, the agent leaves variables I and pi unchanged.

During the second moving period, agent A performs the first pi edge traversals of `Explo`(n). Once this is done, agent A checks whether $H = I$ or not. If $H = I$, then the third waiting period

lasts $\mathcal{T} + X_n - 1$ rounds, and at its expiration, A transits to state `Synchronization`. Otherwise, the third waiting period lasts $2\mathcal{T} + X_n - 1$ but can be interrupted when agent A notices at least $\lceil \frac{3|I|}{4} \rceil$ agents in state `Synchronization` in its node: as soon as such an event occurs, agent A exits the execution of `Merge`$(\mathcal{T}, n)$. In case such an interruption does not occur, the agent exits the execution of `Merge`$(\mathcal{T}, n)$ at the end of the waiting period.

State `Synchronization` Agent A spends one round in this state and then exits the execution of `Merge`$(\mathcal{T}, n)$.

5.3.2.3 Correctness and complexity analysis

The proof of correctness and cost analysis of procedure `Merge` only consist of the following theorem.

Theorem 5.2. *Consider a team of agents in a graph of size at most n. Let r_0 be the first round when a good agent starts executing* `Merge`$(\mathcal{T}, n)$*. If every good agent starts executing* `Merge`$(\mathcal{T}, n)$ *by round $r_0 + \mathcal{T} - 1$ and among them at least $4f + 2$ start the execution in the same node and at the same round, then all good agents finish their executions of procedure* `Merge` *in the same node and at the same round $r < r_0 + 4\mathcal{T} + 6X_n - 1$.*

Proof. Note that according to procedure `Merge`, every good agent spends at most $4\mathcal{T} + 6X_n - 1$ rounds in any execution of procedure `Merge`$(\mathcal{T}, n)$. Hence, to prove the theorem it is enough to prove that all good agents finish their executions of procedure `Merge` in the same node and at the same round.

Let us denote by H_1 the largest list H built by any good agent in state `Census`, and by A one of the good agents that builds it. By assumption, they are at least $4f + 2$ good agents that start the execution in the same node and at the same round. As a result, in view of the description of state `Census`, H_1 contains at least $4f + 2$ elements, and agent A belongs to the group of at least $3f + 2$ good agents in state `Census` that compute the same list H_1 at a round r_1 in a node v_1. Let us call T_1 the group of all the good agents in state `Census` in node v_1 at round r_1. This proof relies on the following two claims.

Claim 5.5. *The agents of T_1 are the only good agents that build list H_1 while in state* `Census`*.*

Proof of the claim: Let us assume by contradiction that the claim is false. Hence, there is a good agent B in state `Census` which also builds H_1 in a node v_2 at a round r_2 such that $v_2 \neq v_1$ or $r_2 \neq r_1$. In view of the description of state `Census`, there are all the labels of the agents of T_1 in H_1. Thus, for each good agent of T_1, there is an agent in state `Census` with the same label in node v_2 at round r_2. However, there are at least $3f + 2$ agents in T_1, and since they only spend round r_1 in state `Census` in node v_1, none of them is in this state in node v_2 at round r_2. Besides, all the good agents have different labels and the Byzantine agents are not numerous enough to be these $3f + 2$ agents in state `Census` in node v_2 at round r_2. This contradicts the existence of these $3f + 2$ agents and the assumption that B builds H_1 in node v_2 at round r_2. Hence, the claim is proven. ⋆

Claim 5.6. *Each good agent starts its third waiting period in node v_1.*

Proof of the claim: This proof starts with showing the following two facts. The first fact is that in each of the rounds belonging to $\{r_1 + 1, \ldots, r_1 + 2\mathcal{T} + 4X_n - 2\}$, there are at least $\lceil \frac{|H_1|}{4} \rceil$ good agents in state `Election` that transmit the list H_1 in node v_1. The second fact is that each good agent performs entirely its first moving period between round $r_1 + 1$ and round $r_1 + 2\mathcal{T} + 4X_n - 2$.

Let us focus on the first fact. In view of the description of state `Census`, the list H_1 contains at least $3f + 2$ elements corresponding to the labels of the agents of T_1, all of which are good, and at most $|T_1| + f$ elements, with $|T_1|$ the number of agents in T_1. This means that $f < \lceil \frac{|H_1|}{4} \rceil$, $|T_1| > \lceil \frac{3|H_1|}{4} \rceil$, $\lfloor \frac{|H_1|}{2} \rfloor - f \geq \lceil \frac{|H_1|}{4} \rceil$ and $\lceil \frac{|H_1|}{2} \rceil - f \geq \lceil \frac{|H_1|}{4} \rceil$ i.e., in each half of H_1 there

are at least $\lceil \frac{|H_1|}{4} \rceil$ labels of agents of T_1. This implies that in each of the rounds belonging to $\{r_1 + \mathcal{T}, \ldots, r_1 + \mathcal{T} + 4X_n - 1\}$, there are at least $\lceil \frac{|H_1|}{4} \rceil$ good agents in state `Election` transmitting the list H_1 in node v_1. Moreover, in view of the description of state `Election`, all the agents of T_1 wait in v_1 and transmit H_1 in each round from round r_1+1 to round $r_1+\mathcal{T}-1$, and from round $r_1 + \mathcal{T} + 4X_n$ to round $r_1 + 2\mathcal{T} + 4X_n - 2$. Hence, the first fact is true.

Let us go further by considering the second fact. Each good agent starts the execution of procedure `Merge` between rounds r_0 and $r_0 + \mathcal{T} - 1$. Then, it spends a single round in state `Census`, and enters state `Election` between round $r_0 + 1$ and round $r_0 + \mathcal{T}$. Actually, the good agents of T_1 are in state `Census` at round r_1. This means that r_1 belongs to $\{r_0; \ldots; r_0 + \mathcal{T} - 1\}$. Since every good agent spends at least $\mathcal{T} - 1$ rounds and at most $\mathcal{T} + 2X_n - 1$ rounds in the first waiting period, every good agent starts its first moving period between round $r_0 + \mathcal{T}$ and round $r_0 + 2\mathcal{T} + 2X_n - 1$ i.e., between round $r_1 + 1$ and round $r_1 + 2\mathcal{T} + 2X_n - 1$. Since the first moving period lasts $2X_n$ rounds, the second fact is true.

Hence, from the two facts, during its first moving period each good agent visits v_1 and notices at least $\lceil \frac{|H_1|}{4} \rceil$ agents in state `Election` transmitting the same list H_1. As a result, in view of the description of state `Election` each good agent finishes the second moving period at round v_1 except if the following event occurs: there is a list H_2 strictly larger than or identical to H_1 such that at a round r_3, in a node $v_3 \neq v_1$, at least $\lceil \frac{|H_2|}{4} \rceil$ agents in state `Election` transmit H_2 to a good agent while it is performing the `Explo`(n) of its first moving period. However, such an event cannot occur. Let us assume by contradiction it can. Since $|H_2| \geq |H_1| \geq 4f + 2$, among $\lceil \frac{|H_2|}{4} \rceil > f$ agents in state `Election` transmitting H_2, there must be at least one good agent which builds H_2 in state `Census`. Note that either H_2 is identical to H_1 or it is larger than H_1. If H_2 is identical to H_1 Claim 1 is contradicted. If H_2 is larger than H_1, it is the maximality of H_1 which is contradicted. This concludes the proof of the claim. ⋆

In view of Claim 2 and the description of states `Census` and `Election`, every good agent finishes its execution in the same node. Hence, to conclude the proof of the theorem, it is enough to prove now that all good agents finish the execution at the same time. To do this, in view of the fact that $|T_1| \geq \lceil \frac{3|H_1|}{4} \rceil$ and the fact that each good agent assigns to variable I the same list H_1 at the end of its first moving period, it is enough to show that there is a round in which the good agents of T_1 are in state `Synchronization` and all the others good agents are performing their third waiting period. It is the purpose of the following lines.

First assume that no good agent prematurely interrupts its third waiting period before round $r_1 + 3\mathcal{T} + 5X_n - 2$. Since each good agent assigns to variable I the same list H_1, each agent of T_1 performs no edge traversal in the second moving period and enters state `Synchronization` at round $r_1 + 3\mathcal{T} + 5X_n - 2$. Each good agent starts its first waiting period between round $r_1 - \mathcal{T} + 2$ and round $r_1 + \mathcal{T}$. Moreover, it can spend from 0 to X_n rounds in its second moving period. This implies that each good agent completes it between round $r_1 + \mathcal{T} + 4X_n - 1$ and round $r_1 + 3\mathcal{T} + 5X_n - 3$ and starts the third waiting period between round $r_1 + \mathcal{T} + 4X_n$ and round $r_1 + 3\mathcal{T} + 5X_n - 2$. Furthermore, each good agent that does not belong to T_1 assigns to variable I a list that is different from the list it has built when in state `Census`, and thus its third waiting period lasts $2\mathcal{T} + X_n - 1$ rounds. This means that each good agent which does not belong to T_1 completes its third waiting period between round $r_1 + 3\mathcal{T} + 5X_n - 2$ and round $r_1 + 5\mathcal{T} + 6X_n - 4$. Hence, each good agent that does not belong to T_1 is performing its third waiting period at round $r_1 + 3\mathcal{T} + 5X_n - 2$ when the agents of T_1 enter state `Synchronization`. As a result, the theorem is true if no good agent prematurely interrupts its third waiting period before round $r_1 + 3\mathcal{T} + 5X_n - 2$. However, no good agent can interrupt its third waiting period at a round $r < r_1 + 3\mathcal{T} + 5X_n - 2$. Indeed, if it was the case, that would imply that there are at least $\lceil \frac{3|H_1|}{4} \rceil$ agents in state `Synchronization` at round r and among them there is necessarily one good agent of T_1: this contradicts the fact that the agents of T_1 enter state `Synchronization` at round $r_1 + 3\mathcal{T} + 5X_n - 2$. This ends the proof of the theorem. □

5.4 The Positive Result

This section shows a procedure, called `Gather`, that solves Byzantine gathering with strong teams in all graphs of size at most n, assuming that the global knowledge is the binary representation of $\lceil \log \log n \rceil$. Note that the length of such a global knowledge belongs to $O(\log \log \log n)$ bits. In this section, the value of $\lceil \log \log n \rceil$ is denoted by $\mathcal{GK}$. The procedure works in a time polynomial in n and $|\ell_{\min}|$, and it makes use of the building blocks introduced in the previous section.

In the sequel, G_n denotes the maximal time complexity of procedure `Group`(X_n, n, ρ) with $\rho \in \{0; 1\}$ in all graphs of size at most n. Moreover, M_n denotes the maximal time complexity of procedure `Merge`$(X_n + G_n, n)$ in all graphs of size at most n. Note that according to Theorems 5.1 and 5.2, G_n and M_n exist and are polynomials in n.

5.4.1 Intuition

In order to better describe the high level idea of `Gather`, first consider a situation that would be ideal to solve Byzantine gathering with a strong team and that would be as follows. Instead of assigning distinct labels to all agents, the adversary assigns to each of them just one bit $\rho \in \{0; 1\}$, so that there are at least one good agent for which $\rho = 0$ and at least one good agent for which $\rho = 1$. Such a situation would clearly constitute an infringement of the model, but would allow the simple protocol described in Algorithm 5.1 to achieve the task in a time that is polynomial in n when $\mathcal{GK} = \lceil \log \log n \rceil$.

Algorithm 5.1 Algorithm executed by every good agent in the ideal situation.

1: let ρ be the bit assigned to me by the adversary
2: execute $\mathcal{G}(\rho)$
3: declare that gathering is achieved

Algorithm 5.2 $\mathcal{G}(\rho)$ executed by a good agent.

1: $N \leftarrow 2^{(2^{\mathcal{GK}})}$
2: execute `Explo`(N)
3: execute `Group`(X_N, N, ρ)
4: execute `Merge`$(X_N + G_N, N)$

Algorithm 5.1 consists mainly of a call to $\mathcal{G}(\rho)$ that is given by Algorithm 5.2. Since $\mathcal{GK} = \lceil \log \log n \rceil$, at line 1 of Algorithm 5.2, N is a polynomial upper-bound on n, and the execution of `Explo`(N) in a call to $\mathcal{G}(\rho)$ by the first woken-up good agent permits to visit every node of the graph and to wake up all dormant agents. As a result, the delay between the starting times of `Group`(X_N, N, ρ) by any two good agents of the strong team is at most X_N. According to the properties of procedure `Group` (refer to Theorem 5.1) and the fact that the number of good agents is at least $5f^2 + 6f + 2 = ((5f + 2) - 1)(f + 1) + 1$, this guarantees in turn that the delay between the starting times of `Merge`$(X_N + G_N, N)$ by any two good agents is at most $X_N + G_N$, and at least $4f + 2$ good agents start this procedure at the same time in the same node. Hence, in view of the properties of procedure `Merge` (refer to Theorem 5.2), all good agents declare gathering is achieved at the same time in the same node after a polynomial number of rounds (in n) since the wake-up time of the earliest good agent.

Unfortunately, such an ideal situation is not assumed. At first glance, one might argue that it is not really a problem because all agents are assigned distinct labels that are, after all, distinct binary strings. Thus, by ensuring that each good agent applies on its label the transformation given in Section 3.2 (or some extension like the transformation described in Section 5.2, and then processes one by one each bit b_i of its doubled label by executing $\mathcal{G}(b_i)$, it can be guaranteed (with some minor technical adjustments) that the gathering of all good agents

is done in time polynomial in n and $|\ell_{\min}|$. Indeed, in view of Proposition 3.1 the conditions of the ideal situation are recreated when the agents process their j-th bits for some j at most linear in $|\ell_{\min}|$. Unfortunately, this is not enough. In fact, in the ideal situation, there is just one bit to process: thus, *de facto* every good agent knows that every good agent knows that gathering will be done at the end of this single process. However, it is no longer the case when the agents have to deal with sequences of bit processes: the good agents have a priori no mean to detect collectively and simultaneously when they are gathered. It should be noted that if the agents knew f, an existing algorithmic component (refer to [51]) allowing to solve Byzantine gathering if at some point some good agents detect the presence of a group of at least $2f + 1$ agents in the network could be used. Such a group is necessarily constructed during the sequence of bit processes given above, but again, it cannot be a priori detected as the agents do not know f or an upper-bound on it. Hence, to optimize the amount of global knowledge, a new strategy is required to allow the good agents to declare gathering achieved jointly and simultaneously. It is the purpose of the rest of this subsection.

To get all good agents declare simultaneously the gathering achieved, it is necessary to reach a round in which every good agent knows that every good agent knows that gathering is done. The introduction by Section 5.2 of a transformation different from that of Section 3.2 is not fortuitous: the doubled label offers a stronger property (refer to Proposition 5.1). When a good agent has finished to read the first half of its doubled label – call such an agent *experienced* – it has the guarantee that the gathering of all good agents has been done at least once. Hence, when an experienced agent starts to process the second half of its doubled label, it actually knows an approximation of the number of good agents with a margin of error of f at the most. For the sake of convenience, consider that an experienced agent knows the exact number $|\mathcal{A}|$ of good agents: the general case adds a slight level of complexity that is unnecessary to understand the intuition. So, each time an experienced agent completes the process of a bit in the second half of its doubled label, it is in a node containing less than $|\mathcal{A}|$ agents or at least $|\mathcal{A}|$ agents. In the first case, the experienced agent is sure that the gathering is not achieved. In the second case, the experienced agent is in doubt. Procedure `Gather` builds on this doubt. How can it be done? So far, each bit process was just made of one call to procedure $\mathcal{G}$: now at the end of each bit process, a waiting period of some prescribed length is added, followed by an extra step that consists in applying $\mathcal{G}$ again, but this time according to the following rule. If during the waiting period it has just done, an agent A was in a node containing, for a sufficiently long period, an agent pretending to be experienced and in doubt (this agent may be A itself), then agent A is said to be *optimistic* and the second step corresponds to the execution of $\mathcal{G}(0)$. Otherwise, agent A is said to be *pessimistic* and the second step corresponds to the execution of $\mathcal{G}(1)$.

If at least one good agent is optimistic within a given second step, then the gathering of all good agents is done at the end of this step. Indeed, through similar arguments of partition to those used for the ideal situation, it can be shown that it is the case when at least another agent is pessimistic. However, it is also, more curiously, the case when there is no pessimistic agents at all. This is due in part to the fact that two good experienced agents cannot have been in doubt in two distinct nodes during the previous waiting period (otherwise, the definition of $|\mathcal{A}|$ would be contradicted). Thus, all good agents start $\mathcal{G}(0)$ from at most $f + 1$ distinct nodes (as the Byzantine agents can mislead the good agents in at most f distinct nodes during the waiting period), which implies by the pigeonhole principle that at least $4f + 2$ good agents start it from the same node. Combined with some other technical arguments, one can show that the conditions of Theorem 5.2 are fulfilled when the agents execute `Merge` at the end of $\mathcal{G}(0)$, thereby guaranteeing again gathering of all good agents.

As a result, the addition of an extra step to each bit process gives the following interesting property: when a good agent is optimistic at the beginning of a second step, at its end the gathering is done and, more importantly, the optimistic agent knows it because its existence ensures it. Note that, it is a great progress, but unfortunately it is not yet sufficient, particularly

because the pessimistic agents do not have the same kind of guarantee. The way of remedying this is to repeat once more the same kind of algorithmic ingredient as above. More precisely, at the end of each second step, a waiting period of some prescribed length is added again, followed by a third step that consists in applying $\mathcal{G}$ in the following manner. If during the waiting period it has just done, an agent X was in a node containing, for a sufficiently long period, an agent pretending to be optimistic, then the third step of agent X corresponds to the execution of $\mathcal{G}(0)$ and it becomes optimistic if it was not. Otherwise, the third step of agent X corresponds to the execution of $\mathcal{G}(1)$ and the agent stays pessimistic.

By doing so, a significant move forward is made. To understand why, the reader is invited to reconsider the case when there is at least one good agent that is optimistic at the beginning of a second step. As seen earlier, at the end of this second step, all good agents are necessarily gathered and every optimistic agent knows it. In view of the last changes made to the solution, when starting the third step, every good agent is then optimistic. As explained above the absence of pessimistic good agent is very helpful, and using here the same arguments, it is certain that when finishing the third step, all good agents are gathered and every good agent knows it because all of them are optimistic. Actually, it is even a little more subtle: the optimistic agents of the first generation (i.e., those that were already optimistic when starting the second step) know that the gathering is done and know that every good agent knows it. Concerning the optimistic agents of the second generation (i.e., those that became optimistic only when starting the third step), they just know that the gathering is done, but do not know whether the other agents know it or not. Recall that to get all good agents declare simultaneously the gathering achieved, the requirement consists in reaching a round in which every good agent knows that every good agent knows that gathering is done. Such a consensus is close. To reach it, at the end of a third step, the optimistic agents of the first generation make themselves known to all agents. Note that if there were at least $f + 1$ agents declaring to be optimistic agents of the first generation and if f was part of $\mathcal{GK}$, the consensus would be reached. Indeed, among the agents declaring to be optimistic of the first generation, at least one is necessarily good and every agent can notice it: at this point it can be shown that every good agent knows that every good agent knows that gathering is done.

However, the agents do not know f. That being said, at the end of a third step, note that an optimistic agent knowing that the gathering is done can compute an approximation $\tilde{f}$ of the number of Byzantine agents. More precisely, if the number of agents gathered in its node is p, the optimistic agent knows that the number of Byzantine agents cannot exceed $\tilde{f} = \max\{y | (5y + 1)(y + 1) + 1 \leq p\}$ according to the definition of a strong team. Based on this fact, the solution is complete. Indeed, the algorithm is designed in such a way that all good agents correctly declare the gathering is achieved in the same round after having computed the same approximation $\tilde{f}$ and noticed at least $\tilde{f} + 1$ agents that claim being optimistic of the first generation during a third step. Such an event necessarily occurs before any agent finishes the $(4|\ell_{\min}|+8)$-th bit process of its doubled label, which permits to obtain the promised polynomial complexity. This is where the feat of strength of procedure `Gather` is: obtaining such a complexity with a small amount of global knowledge, while ensuring that the Byzantine agents cannot confuse the good agents in any way. Actually, the algorithm is judiciously orchestrated so that the only thing Byzantine agents can really do is just to accelerate the resolution of the problem.

5.4.2 Formal Description

Algorithm 5.3 gives the formal description of procedure `Gather`. As mentioned at the beginning of this section, $\mathcal{GK} = \lceil \log \log n \rceil$. Procedure `Gather` uses the two building blocks `Group` and `Merge` described in the previous section. It also uses two small subroutines, `Learn` and `CheckGathering`, which are described after Algorithm 5.3. Both these subroutines do not have any input parameters, but when executing them, the agent can access to the current value

of every variable defined in Algorithm 5.3. Hence the variables defined in Algorithm 5.3 can be viewed as variables of global scope.

Algorithm 5.3 Procedure `Gather` executed by an agent A with label ℓ_A.

```
1: N ← 2^(2^(G_K))
2: // recall that D(ℓ_A) is the doubled label of agent A (refer to Section 5.2)
3: γ ← 1
4: i ← 1
5: execute Explo(N)
6: while i ≤ 3|D(ℓ_A)| do
7:     if i mod 3 = 1 then
8:         ω ← 0
9:         ρ ← D(ℓ_A)[(i div 3) + 1]
10:    end if
11:    execute Group(X_N, N, ρ)
12:    execute Merge(X_N + G_N, N)
13:    execute Learn
14:    let (ρ, γ) be the value returned by Learn
15:    if ρ = 0 then
16:        ω = ω + 1
17:    end if
18:    if i mod 3 = 0 then
19:        execute CheckGathering
20:        let flag be the boolean value returned by CheckGathering
21:        if flag =true then
22:            declare that gathering is achieved
23:        end if
24:    end if
25:    let r be the time elapsed since the beginning of the execution of this procedure
26:    wait X_N + i(3X_N + 4(G_N + M_N) + 2) − r rounds
27:    i ← i + 1
28: end while
```

In the presentation of the high level idea of procedure `Gather` in the previous subsection, some qualifiers like "experienced and in doubt", "optimistic of the second generation" or "optimistic of the first generation" are used only to ease the understanding but do not appear explicitly in the formal description. However note that these qualifiers are reflected in the values $1, 2$ or 3 of variable ω. For example, an optimistic agent of the first generation corresponds to an agent for which $\omega = 3$.

The following paragraphs formally describe the subroutines `Learn` and `CheckGathering`, starting with `Learn`.

Subroutine `Learn`

When executing this subroutine, an agent A can transit to different states that are `Learning`, `Optimist` and `Pessimist`. The initial state is `Learning`. During an execution of this procedure, A never moves. Denote by v the node occupied by the agent while executing this subroutine and by T_N the value $X_N + G_N + M_N$.

State `Learning` Agent A spends one round in this state. Let x be the maximum number of agents in state `Learning` (including itself) that A notices at this round in node v. Let z be $max(\gamma, x)$. The agent A transits either to state `Optimist` or to state `Pessimist`. It transits to state `Optimist` if $\omega \neq 0$, or $2i > 3|D(\ell_A)|$ and $x \geq z - \max\{y|(5y+1)(y+1)+1 \leq z\}$. Otherwise, it transits to state `Pessimist`.

State `Optimist` Agent A waits $3T_N$ rounds in this state. At the end of this waiting period, the agent exits the execution of `Learn`: the returned value of the subroutine is then the couple $(0, z)$.

State `Pessimist` Agent A waits $3T_N$ rounds in this state. At the end of the waiting period, the agent exits the execution of `Learn` and returns a couple, the value of which is as follows. If during the waiting period, agent A notices $2T_N$ consecutive rounds such that in each of

them there is at least one agent in state `Optimist` in node v, then the returned value is $(0, z)$. Otherwise the returned value is $(1, z)$.

The next paragraph describes the second subroutine `CheckGathering`.

Subroutine `CheckGathering`

Agent A waits a single round and then exists the execution of `CheckGathering`. During this round, agent A transmits the value of its variable ω, and the word "Check-gathering" in order to indicate that it is executing the same named subroutine. Denote by p the number of agents in its current node during the single round of the execution. If the value of the variable ω of A belongs to $\{2; 3\}$, and there are more than $\max\{y|(5y+1)(y+1)+1 \leq p\}$ distinct agents transmitting 3 and "Check-gathering", the subroutine returns `true`. Otherwise, the subroutine returns `false`.

5.4.3 Proof and Analysis

This subsection proves the correctness and the polynomiality of procedure `Gather` to solve Byzantine gathering with strong teams in all graphs of size at most n, assuming that $\mathcal{GK} = \lceil \log \log n \rceil$.

Proposition 5.3. *Let p be a positive integer. If within a team of p agents, there are $g \geq (5f+1)(f+1)+1$ good agents and at most f Byzantine agents, then $f \leq \max\{y|(5y+1)(y+1)+1 \leq p\} < g$.*

Proof. First of all, $f \leq \max\{y|(5y+1)(y+1)+1 \leq p\}$ follows from the fact that $p \geq (5f+1)(f+1)+1$. Then, assume by contradiction that $\max\{y|(5y+1)(y+1)+1 \leq p\} \geq g$. This implies that $(5g+1)(g+1)+1 \leq p$. However, $2g < (5g+1)(g+1)+1$ and $p \leq g+f < 2g$. By transitivity, $2g < 2g$. This is a contradiction, which completes the proof. □

The executions of all the subroutines and building blocks that are mentioned in the following statements and their proofs always occur during an execution of procedure `Gather` with $\mathcal{GK} = \lceil \log \log n \rceil$ by an agent in a graph of size at most n. Hence, for ease of reading, this is omitted. The following notations are used in the statement of the next lemma and its proof. For any good agent A, denote by $r_{A,i}$ the round (if any) at which A starts its i-th execution of procedure `Group`. Also denote by t_i the first round (if any) at which there is at least one good agent that starts its i-th execution of procedure `Group`. Finally, according to line 1 of Algorithm 5.3, N is the value $2^{(2^{\mathcal{GK}})}$.

Lemma 5.7. *Let A be a good agent. For any positive integer i, $r_{A,i+1} = r_{A,i} + 3X_N + 4G_N + 4M_N + 2$, and every good agent that starts its i-th execution of* `Group` *does it at round $t_i + X_N - 1$ at the latest.*

Proof. Since `Explo`(N) allows to visit every node of the graph, once the first awoken good agent has completed its first execution of `Explo`(N) at the beginning of procedure `Gather`, each good agent is awoken and has at least started its first execution of `Explo`(N). Every good agent spends exactly X_N rounds executing it, and then starts its first execution of procedure `Group`. Hence, every good agent starts its first execution of procedure `Group` in some interval of X_N rounds, between rounds t_1 and $t_1 + X_N - 1$.

Now consider the routines a good agent executes between the beginnings of any two consecutive executions of `Group`. To conclude this proof, it is enough to show that their execution lasts at most $3X_N + 4G_N + 4M_N + 2$ rounds. Any good agent spends at most G_N rounds executing `Group`(X_N, N, bin) for any $bin \in \{0; 1\}$, at most M_N rounds executing `Merge`$(X_N + G_N, N)$, exactly $3T_N + 1$ rounds executing `Learn`, and exactly 1 round executing `CheckGathering`. The sum of these amounts of rounds is $2 + 3T_N + M_N + G_N$.

In view of line 26 of Algorithm 5.3 and since each agent spends exactly X_N rounds executing the initial `Explo`(N), for any positive integer i and any good agent A, the above sum is exactly the

amount of rounds between $r_{A,i}$ and $r_{A,i+1}$. Hence, any good agent spends exactly $4M_N + 4G_N + 3X_N + 2$ rounds between the beginnings of any two consecutive executions of procedure `Group`, which completes the proof. □

Proposition 5.4. *Consider a round r at which two good agents A and B execute the same routine R from the set* {`Group`, `Learn`, `Merge`, `CheckGathering`}*. If A is executing its i-th execution of R at round r, then B is also executing its i-th execution of R at round r.*

Proof. Assume by contradiction that there exists some round r_1 at which two good agents A and B are respectively executing their i-th and j-th execution of a same routine R from the set {`Group`, `Learn`, `Merge`, `CheckGathering`} with $i < j$. In view of Lemma 5.7, $(r_{B,j} - r_{B,i}) \geq 3X_N + 4G_N + 4M_N + 2$. Thus $(r_1 - r_{B,i}) \geq 3X_N + 4G_N + 4M_N + 2$. However, if R is `Group` or `Merge`, then $(r_1 - r_{A,i}) \leq G_N + M_N$. Hence, $(r_{A,i} - r_{B,i}) > X_N$, which contradicts Lemma 5.7.

Hence, R must be `Learn` or `CheckGathering`. In either case, in view of Algorithm 5.3, between $r_{B,j}$ and r_1, B must have executed `Group` and `Merge`. Executing these routines requires a minimal number a rounds which is strictly larger than X_N. This means that, $(r_1 - r_{B,i}) \geq 4X_N + 4G_N + 4M_N + 3$. On the other hand, $(r_1 - r_{A,i}) \leq 3X_N + 4G_N + 4M_N + 2$ rounds. Hence, $(r_{A,i} - r_{B,i}) > X_N$, which contradicts again Lemma 5.7. □

By Lemma 5.7 and lines 11-12 of Algorithm 5.3, all good agents that start their i-th execution of `Merge`$(X_N + G_N, N)$ for a given positive integer i, do it in an interval lasting at most the number of rounds given as first parameter of `Merge`. Hence, the following corollary is a consequence of Theorem 5.1, Theorem 5.2, and Lemma 5.7.

Corollary 5.2. *Assume that for a given positive integer i, there is a group of at least $(5f + 1)(f + 1) + 1$ good agents that start (at possibly different nodes or rounds) their i-th executions of* `Group`(X_N, N, ρ) *with $\rho = 0$ for at least one good agent and $\rho = 1$ for at least one other good agent. There exist a node v_1 and a round r_1 such that each good agent in the graph that completes its i-th execution of* `Merge` *does it at round r_1 in node v_1.*

Before proving Theorem 5.3 that is the main result of this section, it is necessary to prove the following series of four lemmas.

Lemma 5.8. *Assume that for a given positive integer i, there are at least $5f+2$ good agents that start (at possibly different rounds) their i-th executions of* `Group`(X_N, N, ρ) *in the same node v_1 with $\rho = 0$. There exist a round r_2 and a node v_2 such that each good agent in the graph that completes its i-th execution of procedure* `Merge` *does it at round r_2 in node v_2.*

Proof. Assume that there exist an integer i and a node v_1 such that a group T_1 of at least $5f+2$ good agents all start their i-th executions of `Group`(X_N, N, ρ) in the same node v_1 with $\rho = 0$. In view of Lemma 5.7, every good agent that starts its i-th execution of `Group`, does it between rounds t_i and $t_i + X_N - 1$. In view of the description of `Group`, at the beginning of its execution of `Group`$(X_N, N, 0)$, every good agent, which is called a follower, first enters state `Invite` and spends strictly more than X_N rounds waiting in this state. Hence, there is at least one round at which each good agent of T_1 is waiting in state `Invite` in node v_1 during the first phase of its i-th execution of `Group`. Thus, by Lemma 5.6, at least $4f + 2$ good agents exit their i-th execution of `Group` at the same round and in the same node. This means that at least $4f + 2$ good agents start their i-th execution of `Merge`$(X_N + G_N, N)$ at the same round and in the same node. Besides, each good agent spends at most G_N rounds in any execution of `Group` which means that each good agent that completes its i-th execution of `Group` does it between round t_i and round $t_i + G_N + X_N - 1$. Hence, in view of Theorem 5.2, there exist a round r_2 and a node v_2 such that each good agent that completes its i-th execution of `Merge`$(X_N + G_N, N)$ does it at r_2 in v_2. □

Lemma 5.9. *Assume that $g \geq (5f+1)(f+1)+1$ good agents start (at possibly different nodes or rounds) their $(3j+1)$-th execution of subroutine* `Group`*, for a given integer j. Let $k \leq 3$ be the smallest positive integer, if any, such that the $(3j+k)$-th execution of procedure* `Learn` *by at least one good agent returns a couple whose the first element is 0. There exists a node v_1 such that each good agent that enters state* `Optimist` *during its $(3j+k)$-th execution of* `Learn`*, does it in v_1.*

Proof. Assume by contradiction that two good agents A and B both enter state `Optimist` during their $(3j+k)$-th executions of `Learn` but from different nodes, respectively v_A and v_B. To conduct this proof, it is necessary to explain what the entrance of these good agents in state `Optimist` implies. Note that during their $(3j+k)$-th executions of `Learn`, both agents have the same value for variable i i.e., $(3j+k)$. In view of lines 7-8 of Algorithm 5.3, when a good agent starts its $(3j+1)$-th execution of `Group`, the value of its variable ω is 0. Since ω is only incremented on line 16, when the first element of the pair returned by `Learn` is 0, by definition of k, at the beginning of their $(3j+k)$-th executions of `Learn`, the value of ω for both A and B is still 0. Thus, in view of the description of procedure `Learn`, $2(3j+k) > 3|D(\ell_A)|$ (resp. $2(3j+k) > 3|D(\ell_B)|$) and while in state `Learning`, A (resp. B) notices at least $z_A - \max\{y_A | (5y_A+1)(y_A+1)+1 \leq z_A\}$ (resp. $z_B - \max\{y_B | (5y_B+1)(y_B+1)+1 \leq z_B\}$) agents in state `Learning` in its node, where $|D(\ell_A)|$ (resp. $|D(\ell_B)|$) denotes the length of the doubled label of A (resp. B) as defined in Algorithm 5.3 and z_A (resp. z_B) denotes the value of z that is used in state `Learning` by agent A (resp. B).

The following step of the proof consists in explaining what $2(3j+k) > 3|D(\ell_A)|$ and $2(3j+k) > 3|D(\ell_B)|$ imply. By Proposition 5.1, it means that there exists a positive integer $s \leq j$ such that the s-th bits in the doubled labels of A and B are different. In view of Algorithm 5.3, this means that for their $(3s-2)$-th executions of `Group`, one of them executes `Group`$(X_N, N, 0)$ while the other one executes `Group`$(X_N, N, 1)$. Hence, in view of Corollary 5.2, there exist a node v_3 and a round r_3 such that each good agent that completes its $(3s-2)$-th execution of `Merge`, does it in v_3 at round r_3. This means that each good agent that starts its $(3s-2)$-th execution of `Learn`, and thus enters state `Learning`, does it in v_3 at round r_3+1. By assumption, these good agents are at least $g \geq (5f+1)(f+1)+1$ and among them, there are A and B. This means that the number of agents in state `Learning` that A (resp. B) notices during its $(3s-2)$-th execution of `Learn`, and thus z_A (resp. z_B) is at least g.

The next part of this proof is built on the consequences of the fact that A (resp. B) notices at least $z_A - \max\{y_A | (5y_A+1)(y_A+1)+1 \leq z_A\}$ (resp. $z_B - \max\{y_B | (5y_B+1)(y_B+1)+1 \leq z_B\}$) agents in state `Learning` while in the same state during its $(3j+k)$-th execution of `Learn`. In view of Proposition 5.3, both $\max\{y_A | (5y_A+1)(y_A+1)+1 \leq z_A\}$ and $\max\{y_B | (5y_B+1)(y_B+1)+1 \leq z_B\}$ are at least f. Assume without loss of generality that $z_B - \max\{y_B | (5y_B+1)(y_B+1)+1 \leq z_B\}$ is at least $z_A - \max\{y_A | (5y_A+1)(y_A+1)+1 \leq z_A\}$. Hence, the sum of the numbers of agents in state `Learning` noticed by A or B while in the same state during their $(3j+k)$-th execution of `Learn` is at least $2(z_A - \max\{y_A | (5y_A+1)(y_A+1)+1 \leq z_A\})$ i.e., at least $g + z_A - 2\max\{y_A | (5y_A+1)(y_A+1)+1 \leq z_A\}$. Besides, $z_A - 2\max\{y_A | (5y_A+1)(y_A+1)+1 \leq z_A\}$ is greater than $2\max\{y_A | (5y_A+1)(y_A+1)+1 \leq z_A\} \geq 2f$. This means that the total number q of agents in state `Learning` noticed by A or B while in the same state during their $(3j+k)$-th executions of `Learn` is greater than $g+2f$. However, this is impossible as explained in the next paragraph.

In view of Proposition 5.4, when A or B starts its $(3j+k)$-th execution of `Learn`, each good agent in state `Learning` is also starting its $(3j+k)$-th execution of `Learn`. No good agent can be in state `Learning` during its $(3j+k)$-th execution of `Learn` both in v_A and in v_B, which means that among the $q > g+2f$ agents noticed by A or B, at most g are good. However, in every round, there are at most f Byzantine agents in v_A or v_B. Hence, q cannot be greater than $g+2f$: this leads to a contradiction that proves the theorem. □

Lemma 5.10. *Let i and j two integers such that $j \in \{1;2\}$. Assume that at least $(5f+1)(f+1)+1$ good agents start (at possibly different nodes or rounds) their $(3i+1)$-th executions of subroutine* `Group`*. If the $(3i+j)$-th execution of subroutine* `Learn` *by at least one good agent returns a couple whose the first element is 0, then for every integer $j < k \leq 3$, there exist a round r and a node v such that every $(3i+k)$-th execution of* `Learn` *by any good agent finishes at round r in node v and returns a pair whose first element is 0.*

Proof. To prove this lemma, it is enough to show that for all integers i and j such that $j \in \{1;2\}$, if at least $(5f+1)(f+1)+1$ good agents start (at possibly different nodes or rounds) their $(3i+1)$-th executions of `Group`, and the $(3i+j)$-th execution of subroutine `Learn` by at least one good agent A returns a couple whose the first element is 0, then the following property is verified: there exist a round r and a node v such every $(3i+j+1)$-th execution of `Learn` by any good agent finishes in node v at round r, and returns a couple whose the first element is 0.

Three cases are considered. In the first case, $j=1$. In the second case, $j=2$ and there is no good agent whose $(3i+1)$-th execution of `Learn` returns a couple in which the first element is 0. In the third case, $j=2$ and there is at least one good agent B (not necessarily different from A) whose $(3i+1)$-th execution of `Learn` returns a couple in which the first element is 0.

Consider the first case. This paragraph aims at proving that all the good agents which complete their $(3i+2)$-th execution of `Merge` do it at the same round, and in the same node. In view Algorithm 5.3, during the $(3i+2)$-th execution of `Group`(X_N, N, ρ) by agent A, $\rho = 0$. So, if there exists a good agent that uses 1 for parameter ρ during its $(3i+2)$-th execution of `Group`(X_N, N, ρ), then in view of Corollary 5.2, all the good agents that complete their $(3i+2)$-th executions of `Merge` do it at the same round and in the same node. The situation, in which each good agent that starts its $(3i+2)$-th execution of `Group`(X_N, N, ρ) uses $\rho = 0$, is a little trickier to analyze. In this situation, in view of Algorithm 5.3, this means that every $(3i+1)$-th execution of `Learn` by any good agent C returns a couple whose first element is 0. Thus, during this execution agent C either enters state `Optimist`, or while in state `Pessimist` it notices at least one agent in state `Optimist` for at least $2T_N$ consecutive rounds. Moreover, in view of Lemma 5.7 and the definitions of values G_N and M_N, every good agent that starts its $(3i+1)$-th execution of `Learn`, does it between rounds t_{3i+j} and $t_{3i+j}+G_N+M_N+X_N-1 = t_{3i+j}+T_N-1$ (T_N is defined in the description of `Learn`). According to the description of state `Learning` (resp. state `Pessimist`), every good agent that enters (resp. exits) state `Pessimist`, does it between rounds $t_{3i+j}+1$ and $t_{3i+j}+T_N$ (resp. $t_{3i+j}+3T_N$ and $t_{3i+j}+4T_N-1$). Hence, there are at most $4T_N-1$ rounds at which at least one good agent is in state `Pessimist` during its $(3i+1)$-th execution of `Learn`. This implies there is at least one round r_1 that overlaps all the intervals of $2T_N$ rounds noticed by the good agents in state `Pessimist`, and such that in round r_1 each good agent in state `Pessimist` is in the same node as at least one agent in state `Optimist`. By Lemma 5.9, there is at most one node where good agents can enter state `Optimist` during their $(3i+1)$-th executions of `Learn`. This implies that there are at most $f+1$ nodes in the graph from which any good agent can exit state `Learning` during its $(3i+1)$-th execution of `Learn`. Since by assumption at least $(5f+1)(f+1)+1$ good agents execute their $(3i+1)$-th executions of `Learn`, there exists at least one node v_1 such that at least $5f+2$ good agents are in v_1 during their $(3i+1)$-th executions of `Learn` as well as at the beginning of their $(3i+2)$-th executions of `Group`(X_N, N, ρ). Recall that these good agents all use 0 as value for parameter ρ during their $(3i+2)$-th executions of `Group`(X_N, N, ρ). Hence, by Lemma 5.8 it follows that all the good agents that complete their $(3i+2)$-th executions of `Merge` do it at the same round and in the same node.

So, in the first case, all the good agents that complete their $(3i+2)$-th executions of `Merge` do it at the same round and in the same node. This paragraph aims at showing that these agents all complete at the same round and in the same node their $(3i+2)$-th executions of `Learn` that returns a couple whose the first element is 0. In view of the description of `Learn`, these good agents do not move and spend exactly $3T_N+1$ rounds during their $(3i+2)$-th executions

of `Learn`. Hence, they enter and exit state `Learning` at the same round, and complete their $(3i+2)$-th executions of `Learn` at the same round and in the same node. Moreover, since the $(3i+1)$-th execution of subroutine `Learn` by agent A returns a couple whose the first element is 0, during its $(3i+2)$-th of subroutine `Learn`, the value of its variable ω is different from 0 and it enters state `Optimist`. Hence, each good agent that enters state `Pessimist` during its $(3i+2)$-th execution of `Learn` notices agent A in state `Optimist` during $3T_N \geq 2T_N$ consecutive rounds. As a result, there exist a round r and a node v such that every $(3i+2)$-th execution of `Learn` by any good agent finishes in node v at round r, and returns a couple whose the first element is 0.

Using similar arguments to those used in the first case, it is possible to show in the second case that there exist a round and a node in which every $(3i+3)$-th execution of `Learn` by any good agent finishes and returns a couple whose the first element is 0.

Consider the third case i.e., $j=2$ and there is at least one good agent B whose $(3i+1)$-th execution of `Learn` returns a couple in which the first element is 0. Using similar arguments to those used in the first case, it is possible to show that, there exist a round and a node in which every $(3i+2)$-th execution of `Learn` by any good agent finishes and returns a couple whose first element is 0. All these good agents start their $(3i+3)$-th executions of `Group`(X_N, N, ρ) from the same node with $\rho = 0$. In view of Lemma 5.8, this implies that there exist a round r_2 and a node v_2 such that every $(3i+3)$-th executions of `Merge` by any good agent finishes in node v_2 at round r_2. Moreover, since every $(3i+2)$-th execution of `Learn` of any good agent C returns a couple whose the first element is 0, during its $(3i+3)$-th execution of `Learn`, agent C enters state `Optimist`: this execution of `Learn` by agent C lasts exactly $3T_N+1$ rounds during which it does not move from v_2. Hence, the $(3i+3)$-th execution of `Learn` of agent C returns a couple whose the first element is 0 at round r_2+3T_N+1, which completes the proof. □

Lemma 5.11. *Assume there is a group G of at least $(5f+1)(f+1)+1$ good agents executing procedure* `Gather` *at a round r_1. If at least one agent of G declares that gathering is achieved at round r_1 in a node v_1, then all agents of G declare that gathering is achieved at r_1 in v_1.*

Proof. By assumption, there is at least one good agent A that declares that gathering is achieved at r_1. Let i_1 be the value of the variable i of agent A at round r_1. In view of Algorithm 5.3, it declares that gathering is achieved after executing subroutine `CheckGathering`, and there exists an integer i_2 such that $i_1 = (3i_2+3)$.

In view of subroutine `CheckGathering`, since A declares gathering achieved at round r_1, the value of its variable ω is either 2 or 3. In view of Algorithm 5.3, this means that there are either two or three executions of `Learn`, out of the three since the beginning of the $(3i_2+1)$-th execution of `Group` by agent A, which have returned a couple whose the first element is 0. In view of Lemma 5.10, this means that there exist a round r_2 and a node v_1 such that each agent of G completes at r_2 in v_1 its $(3i_2+3)$-th execution of `Learn`, the returned value of which is a couple whose the first element is 0.

Consider the set of the values of variable ω of every good agent of G at the end of its $(3i_2+3)$-th execution of `Learn`, and denote by ω_1 the maximum one. Since at the end of the $(3i_2+3)$-th execution of `Learn` by agent A, the variable ω of A is either 2 or 3, ω_1 is also either 2 or 3. Lemma 5.10 implies that at the end of the $(3i_2+3)$-th execution of `Learn` by every good agent B of G (including A), the variable ω of B is either ω_1 or $\omega_1 - 1$.

Each agent of G starts its (i_2+1)-th execution of subroutine `CheckGathering` at r_2+1 in v_1. According to the description of this subroutine and Algorithm 5.3, agent A declares that gathering is achieved at round r_1 because at the previous round, while executing `CheckGathering`, it notices strictly more than $\max\{y|(5y+1)(y+1)+1 \leq p\}$ distinct agents executing the same procedure and transmitting 3. Thus, the round at which all the agents of G execute `CheckGathering` in v_1 is r_1-1. Since at least $(5f+1)(f+1)+1$ good agents are in the same node, by Proposition 5.3, at least one good agent transmits 3 at round r_1-1. In view of the

fact that the integer transmitted by any good agent executing `CheckGathering` is the value of its variable ω, ω_1 is 3 and the value of variable ω of each agent of G is either 2 or 3. This means the execution of `CheckGathering` of every good agent of G returns `true` at round $r_1 - 1$ in v_1, and every good agent of G declares that gathering is achieved at round r_1 with agent A, which completes this proof. □

The next result is the last of this section. Recall that a strong team is a team in which the number of good agents is at least $5f^2 + 6f + 2$. As the reader would have noticed, a good agent can execute several iterations of the while loop of Algorithm 5.3 (refer to lines 6 to 28): given a good agent A, the i-th iteration of this while loop by agent A is said to be of *order* i.

Theorem 5.3. *Assuming that $\mathcal{GK} = \lceil \log \log n \rceil$, procedure* `Gather` *solves Byzantine gathering with every strong team in all graph of size at most n, and has a time complexity that is polynomial in n and $|\ell_{\min}|$.*

Proof. Let r be the first round in which a good agent finishes the execution of procedure `Gather`. Since, the adversary wakes up at least one good agent, round r exists. Since $\mathcal{GK} = \lceil \log \log n \rceil$, $N = 2^{(2^{\mathcal{GK}})}$ is at least n, and thus according to line 5 of Algorithm 5.3, all the good agents are executing procedure `Gather` at round r. As a result, in view of Lemma 5.11, it is enough to prove the following two properties to state that the theorem holds. The first property is that there exists at least one good agent that declares gathering is achieved at round r (note that although it is proved impossible in the sequel, the possibility that an agent might finish the execution of procedure `Gather` without declaring gathering is achieved cannot be ruled out for now). The second property is that at round r, the first woken-up agent (or one of the first, if there are several such agents) has spent a time that is at most polynomial in n and $|\ell_{\min}|$ to execute procedure `Gather`.

First focus on the first property and consider the good agent A with the smallest label $\ell_{\min}$. Let $\alpha = 3|D(\ell_{\min})|$. In view of Algorithm 5.3, each good agent executes at least α iterations of the while loop of Algorithm 5.3, unless it declares that gathering is achieved before. Two cases are considered: either there is at least one good agent B that never starts executing its α-th iteration of the while loop, or every good agent start executing at some point its α-th iteration of the while loop.

Concerning the first case, assume without loss of generality that B is the first agent that stops executing procedure `Gather` before starting its α-th iteration of the while loop. According to Lemma 5.7, the time spent executing an iteration is the same regardless of the executing good agent and the order of the iteration, and this time is greater than the difference between the rounds at which any two good agents start iterations of the same order. Hence, when agent B stops executing procedure `Gather`, no good agent has completed its α-th iteration of the while loop. This implies that agent B finishes its execution of procedure `Gather` at round r. Moreover, the fact that B stops executing procedure `Gather` before starting its α-th iteration of the while loop, implies that B declares the gathering is achieved at round r: this proves that the first property holds in the first case.

Now consider the second case. In view of Proposition 5.1, for any given good agent C different from A, there exist two positive integers i and j such that $2i \leq |D(\ell_{\min})|$, $|D(\ell_{\min})| < 2j \leq 2|D(\ell_{\min})|$ and the i-th (resp. j-th) bits in the doubled labels of A and C are different. Hence, at round r, each good agent has at least started executing its α-th iteration of the while loop, and thus has completed its $(3i-2)$-th iteration and at least started its $(3j-2)$-th iteration.

Moreover, in view of Algorithm 5.3, for its $(3i-2)$-th (resp. $(3j-2)$-th) execution of `Group`(X_N, N, ρ), agent A uses for parameter ρ a value belonging to $\{0;1\}$ that is different of that used by agent C (which also belongs to $\{0;1\}$) during its $(3i-2)$-th (resp. $(3j-2)$-th) execution of `Group`(X_N, N, ρ). By Corollary 5.2, there exist a round r_i and a node v_i (resp. r_j and v_j) such that each good agent completes its $(3i-2)$-th (resp. $(3j-2)$-th) execution of `Merge` at r_i in v_i (resp. at r_j in v_j). At round $r_i + 1$, each good agent enters state `Learning` in

node v_i. Thus, at this point the value of variable γ of each good agent is at least the number of good agents and at most the total number of agents. Since there are at least $(5f+1)(f+1)+1$ good agents, in view of Proposition 5.3, $\max\{y|(5y+1)(y+1)+1 \leq \gamma\}$ is at least f, and $\gamma - \max\{y|(5y+1)(y+1)+1 \leq \gamma\}$ is at most the number of good agents.

Furthermore, the length of the doubled label of A is $|D(\ell_{\min})|$. This means that during its $(3j-2)$-th execution of `Learn`, at round r_j+2, while all good agents are in v_j, agent A enters state `Optimist`. At the same round, every other good agent is also in v_j entering either state `Optimist` or state `Pessimist`. Whichever the state, they spend $3T_N$ rounds in it so that all the good agents in state `Pessimist` notice agent A in state `Optimist` during at least $2T_N$ rounds. As a result, every good agent finishes its $(3j-2)$-th execution of `Learn` that returns a pair whose first element is 0. From Lemma 5.10, it follows that there exist a round r_1 and a node v_1 such that each good agent completes its $3j$-th execution of subroutine `Learn` at r_1 in v_1, and the value of variable ω of each good agent at round r_1 is 3. From round r_1+1 on, each good agent starts its j-th execution of `CheckGathering`. When executing this procedure, each of them transmits the word "Check-gathering" and the value 3 of its variable ω. In view of Proposition 5.3, there are strictly more than $\max\{y|(5y+1)(y+1)+1 \leq p\}$ good agents. Hence, agent A as well as all good agents return `true`, and thus declare that gathering is achieved at round $r = r_1+2$ in node v_1 which proves that the first property holds in the second case as well.

This paragraph proves the second property. According to the two cases analyzed above, the good agents declare that the gathering is achieved at round r before any of them starts its iteration of the while loop of order $\alpha+1$: the value α is polynomial in $|\ell_{\min}|$ since $\alpha = 3|D(\ell_{\min})|$ and $|D(\ell_{\min})| = 4|\ell_{\min}|+8$. Besides, the number of rounds required to execute any iteration of the while loop is bounded by $4(X_N+G_N+M_N+1)$ in view of Lemma 5.7. Note that in view of the definitions of X_N, G_N and M_N, $4(X_N+G_N+M_N+1)$ is polynomial in N, and thus in n as $N = 2^{(2^{\lceil \log\log n \rceil})}$ (refer to line 1 of Algorithm 5.3). Hence, the total number of rounds spent by any good agent before round r is bounded by $12(4|\ell_{\min}|+8)(X_N+G_N+M_N+1)$, which is polynomial in n and $|\ell_{\min}|$. This concludes the proof of the second property, and by extension, of the theorem. □

5.5 The Negative Result

Algorithm `Gather` introduced in the previous section uses the value $\lceil \log\log n \rceil$ as global knowledge, which can be coded with a binary string of size $O(\log\log\log n)$. This section shows that, to solve Byzantine gathering with all strong teams, in all graphs of size at most n, in a time polynomial in n and $|\ell_{\min}|$, the order of magnitude of the size of knowledge used by algorithm `Gather` is optimal. More precisely, the following theorem is proved.

Theorem 5.4. *There is no algorithm solving Byzantine gathering with all strong teams in all graphs of size at most n, which is polynomial in n and $|\ell_{\min}|$ and which uses a global knowledge of size $o(\log\log\log n)$.*

Proof. Suppose by contradiction that the theorem is false. Hence, there exists an algorithm `Alg` that solves Byzantine gathering with all strong teams for all f in all graphs of size at most n, which is polynomial in n and $|\ell_{\min}|$ and which uses a global knowledge of size $o(\log\log\log n)$. The proof relies on the construction of a family $\mathcal{F}_n$ (for any $n \geq 4$) of initial instances with strong teams such that for each of them the graph size is at most n. The goal is to prove that there is an instance from $\mathcal{F}_n$ for which algorithm `Alg` needs a global knowledge whose size does not belong to $o(\log\log\log n)$, which would be a contradiction with the definition of `Alg`.

An infinite sequence of instances $\mathcal{I} = I_0, I_1, I_2, \ldots, I_i, \ldots$ is constructed by induction on i as follows. Instance I_0 consists of an oriented ring of 4 nodes (i.e., a ring in which at each node the edge going clockwise has port number 0 and the edge going anti-clockwise has port 1). In this

ring, there is no Byzantine agent but there are two good agents labeled 0 and 1 that are placed in diametrically opposed nodes. All the agents in I_0 wake up at the same time.

The construction of instance I_i with $i \geq 1$ uses some features of instance I_{i-1}. Let c be the smallest constant integer such that the time complexity of algorithm `Alg` is at most n^c from every instance made of a graph of size at most n with a strong team in which $|\ell_{\min}| = 1$. Let μ_{i-1} and n_{i-1} be respectively the total number of agents in I_{i-1} and the number of nodes in the graph of I_{i-1}. Instance I_i consists of an oriented ring of $(n_{i-1})^{4c}$ nodes. In this ring an agent labeled 0 is placed on a node denoted by v_0. In each of the nodes that are adjacent to v_0, $(n_{i-1})^c \cdot \mu_{i-1}$ Byzantine agents are placed (which gives a total of $2(n_{i-1})^c \cdot \mu_{i-1}$ Byzantine agents). On the node that is diametrically opposed to v_0, enough good agents are placed in order to have a strong team. The way of assigning labels to all agents that are not at v_0 is arbitrary but respects the condition that initially no two agents share the same label. Finally, all the agents in I_i wake up at the same time. This closes the description of the construction of $\mathcal{I}$, about which the following claim is now proved.

Claim 5.7. *For any two instances I_j and $I_{j'}$ of $\mathcal{I}$, algorithm* `Alg` *requires a distinct global knowledge.*

Proof of the claim: Assume by contradiction that the claim does not hold for two instances I_j and $I_{j'}$ such that $j < j'$. Consider any execution EX_j of algorithm `Alg` from I_j. According to the construction of $\mathcal{I}$, every agent is woken up at the first round of EX_j. Denote by $r_1, r_2, \ldots, r_k$ the sequence of consecutive rounds from the first round of EX_j to the round when all good agents declare that gathering is done. Also denote by G_i the group of agents (possibly empty) that are with the good agent labeled 0 at round r_i of EX_j. Now, using execution EX_j, a possible execution $EX_{j'}$ of algorithm `Alg` from $I_{j'}$ is designed in such a way that it will fool the good agent labeled 0 and will induce it into premature termination. According to the construction of $\mathcal{I}$, all the agents of $I_{j'}$ are woken up in the first round of $I_{j'}$ and all the good ones are executing algorithm `Alg`. In the first round of $EX_{j'}$ the agent labeled 0 is alone (as in the first round of EX_j). Then, for each $i \in 2, \ldots, k$, the good agent labeled 0 in $EX_{j'}$ meets a group of $|G_i|$ Byzantine agents whose the multiset of labels is exactly the same as the multiset of labels belonging to the agents of G_i in the i-th round of EX_j. This is always possible in view of the fact that for each $i \in 1, \ldots, k$, $|G_i| \leq \mu_j$ and the Byzantine agents of $I_{j'}$ can choose to move by ensuring that in the i-th round of $EX_{j'}$ it remains at least $(k-i) \cdot \mu_j$ Byzantine agents in the node adjacent to the one occupied by the agent labeled 0 in the clockwise direction (resp. anti-clockwise direction): indeed according to the construction of $I_{j'}$, in each of both nodes adjacent to the starting node of the good agent labeled 0, there are initially $(n_{j'-1})^c \cdot \mu_{j'-1} \geq k \cdot \mu_{j'-1}$ Byzantine agents, as $k \leq (n_j)^c \leq (n_{j'-1})^c$. Finally, if algorithm `Alg` prescribes some message exchange between agents during their meetings, then the Byzantine agents in execution $EX_{j'}$ give exactly the same information to 0, as the agents with respective labels in execution EX_j. Hence, from the point of view of agent 0, the first k rounds of EX_j look exactly identical to the first k rounds of $EX_{j'}$. This is due to the actions of Byzantine agents, the fact that all nodes in I_j and $I_{j'}$ look identical, and also because $k \leq (n_j)^c$ which implies that, regardless of the algorithm `Alg`, the agent labeled 0 cannot meet any good agent in the first k rounds of $EX_{j'}$ as the distance between agent 0 and any other good agent is initially at least $\frac{(n_{j'-1})^{4c}}{2} \geq \frac{(n_j)^{4c}}{2}$. Therefore, in the k-th round of execution $EX_{j'}$, the good agent labeled 0 declares having met all good agents and stops, which is incorrect, since it has not met any good agent. This contradicts the definition of algorithm `Alg` and closes the proof of this claim. ⋆

Now, consider the largest x such that in each of the $x+1$ first instances $I_0, I_1, \ldots, I_x$ of $\mathcal{I}$, the graph size is at most n: these $x+1$ instances constitute family $\mathcal{F}_n$. In view of the construction of sequence $\mathcal{I}$ and the definition of x, $4^{((4c)^x)} \leq n < 4^{((4c)^{x+1})}$. Hence, x belongs to $\Omega(\log \log n)$. However, according to Claim 5.7, the global knowledge given to distinct instances in this family must be different. Hence, there is at least one instance of $\mathcal{F}_n$ for which algorithm

`Alg` uses a global knowledge of size $\Omega(\log x)$: since $x \in \Omega(\log \log n)$, $\Omega(\log x) \in \Omega(\log \log \log n)$. This contradicts the fact that `Alg` uses a global knowledge of size $o(\log \log \log n)$ and proves the theorem. □

5.6 Conclusion

This chapter presents the first algorithm polynomial in n and $|\ell_{\min}|$ allowing to gather all good agents in presence of Byzantine ones that can act in an unpredictable way and lie about their labels. This algorithm works under the assumption that the team evolving in the network is strong i.e., the number of good agents is roughly at least quadratic in the number f of Byzantine agents. The required global knowledge $\mathcal{GK}$ is of size $O(\log \log \log n)$, which is of optimal order of magnitude to get a time complexity that is polynomial in n and $|\ell_{\min}|$ even with strong teams.

A natural open question that immediately comes to mind is to ask if doing the same is possible when the ratio between the good agents and the Byzantine agents is reduced. For instance, could it be still possible to solve the problem in polynomial time with a global knowledge of size $O(\log \log \log n)$ if the number of good agents is at most $o(f^2)$? Note that the answer to this question may be negative but then may become positive with a little bit more global knowledge. Actually, it can even easily be shown that the answer is true if the agents are initially given a complete map of the graph with all port numbers, and in which each node v is associated to the list of all labels of the good agents initially occupying node v. However, the size of $\mathcal{GK}$ is then huge as it belongs to $\Omega(n^2)$. In fact, in this case what is really interesting is to find the optimal size for $\mathcal{GK}$. This observation allows to conclude with the following open problem that is more general and appealing.

What are the trade-offs among the ratio good/Byzantine agents, the time complexity and the amount of global knowledge to solve Byzantine gathering?

Bringing an exhaustive and complete answer to this question appears to be really challenging but would turn out to be a major step in our understanding of the problem.

Chapter 6

Treasure Hunt in the Plane with Angular Hints

Contents

6.1 Introduction

A tourist visiting an unknown town wants to find her way to the train station or a skier lost on a slope wants to get back to the hotel. Luckily, there are many people that can help. However, often they are not sure of the exact direction: when asked about it, they make a vague gesture with the arm swinging around the direction to the target, accompanying the hint with the words "somewhere there". In fact, they show an angle containing the target. Can such vague hints help the lost traveler to find the way to the target?

In other words, there is only one agent in the Euclidean plane, which aims at finding an inert treasure, modeled as a point, knowing neither the distance between them nor any bound on it. Finding the treasure means reaching a position in which the treasure and the agent are at distance at most 1 from each other. This problem is referred to as treasure hunt. In applications, from such a distance the treasure can be seen. The agent makes a series of moves with the following addition. In the beginning and after each move the agent gets a hint consisting of a positive angle smaller than 2π whose vertex is at the current position of the agent and within which the treasure is contained. This chapter investigates the question of how these hints permit the agent to lower the cost of finding the treasure, using a deterministic algorithm.

6.1.1 Model and Task Formulation

In this chapter, some additions to the model described in Chapter 2 have to be made.

First of all, the mobile agent considered moves in the Euclidean plane, searching for an inert treasure. When moving, just like in Chapter 4, the mobile agent passes by other points than its origin and destination. This is particularly important in this chapter, as visiting these points permits the agent to check from each of them whether it has found the target.

Besides the specificities of moving in the plane, in the present chapter, upon waking up and after each move, the mobile agent obtains hints consisting of an angle containing the treasure.

Deriving the model from Chapter 2 to introduce these features consists of three steps: extending Definition 2.11 to add the treasure and the angular hints to the environment, introducing a new rule to replace Definition 2.21 in the newly defined environment, and explaining that the mobile agent learns hint containing the treasure upon waking up and at the completion of each move.

Definition 6.1 (Treasure hunt environment)**.** *Each treasure hunt environment is a plane environment* $(P, Z, Q, d, c, r), T, L, i, w, g, f)$ *(cf. Definition 4.1) extended by the addition of a position* $x \in P$ *called treasure, and an application* h *from* $P \times T \times L$ *to* $(-\pi, \pi]^2$ *verifying the following. For any position* p*, time* t*, and label* ℓ*, the image* (ρ, ϕ) *by* h *of* (p, t, ℓ) *is such that the angular coordinate of* x *in the polar coordinate system centered at* p *belongs to* $[\rho - \frac{\phi}{2}, \rho + \frac{phi}{2}]$*.*

Definition 6.2 (Moving in a treasure hunt environment)**.** *Let* $(e, \mathcal{A})$ *be any execution with* e *the treasure hunt environment* $(s, T, L, i, w, g, f, x, h)$*. Definition 4.2 applies as if the execution were* $((s, T, L, i, w, g, f), \mathcal{A})$*.*

In other words, since a treasure hunt environment is an extension of a plane environment, the latter is used to determine the progress of the moves following Definition 4.2. This means that when moving from a position to another in the plane, each mobile agent visits each position on the line segment between the origin and the destination. Besides deciding how much time every mobile agent spends in each of its move, the adversary can be viewed as able to move each mobile agent back and forth along the latter segment during the move.

Definition 6.3 (Learning angular hints)**.** *Let* $(e, \mathcal{A})$ *be any execution with* e *the treasure hunt environment* $(s, T, L, i, w, g, f, x, h)$*. For every time* t*, and any label* ℓ*, if, at* t*, the mobile agent with label* ℓ *either wakes up (cf. Definition 2.16) or completes any move, in some position* p*, then still at* t*, it learns* $h(p, t, \ell)$*.*

Definitions 6.4 and 6.5 complete this section by stating respectively the model variant $\mathcal{H}$ considered in this chapter, and the task of treasure hunt.

Definition 6.4 (Model variant $\mathcal{H}$)**.** *This model variant (cf. Definition 2.12) is the set of all treasure hunt environments.*

Definition 6.5 (Treasure hunt)**.** *Let* M *be any model variant (cf. Definition 2.12), and* $\mathcal{A}$ *be any mobile agent algorithm. Algorithm* $\mathcal{A}$ *achieves treasure hunt in* M *if and only if for every treasure hunt environment* $e = (s, T, L, i, w, g, f, x, h)$ *of* M *with* $|L| = 1$*, the execution* $(e, \mathcal{A})$ *(cf. Definition 2.13) meets the following condition. There exists a time at which the position* p *of the mobile agent and the treasure are within each other's range (cf. Definitions 2.6 and 2.10).*

Hence, treasure hunt is achieved whenever the treasure is found which occurs when the latter is within the range of the current position of the mobile agent. This means that the treasure is found when at distance at most 1 from the mobile agent.

6.1.2 Contribution

It is shown that if all angles given as hints are at most π, then the cost of treasure hunt can be lowered to $O(\Delta)$ (where Δ denotes the initial distance to the treasure), which is optimal. The real challenge here is in the fact that hints can be angles of size *exactly* π, in which case the design of a trajectory always leading to the treasure, while being cost-efficient in terms of traveled distance, is far from obvious.

If all angles are at most β, where $\beta < 2\pi$ is a constant unknown to the agent, then the cost is at most $O(\Delta^{2-\epsilon})$, for some $\epsilon > 0$. Finally, arbitrary angles smaller than 2π given as hints cannot be of significant help: using such hints the cost $\Theta(\Delta^2)$ cannot be beaten.

For both positive results, deterministic algorithms achieving the above costs are presented. Both algorithms work in phases "assuming" that the treasure is contained in increasing squares centered at the initial position of the agent. The common principle behind these algorithms is to move the agent to strategically chosen points in the current square, depending on previously obtained hints, and sometimes perform exhaustive search of small rectangles from these points, in order to guarantee that the treasure is not there. This is done in such a way that, in a given phase, obtained hints together with small rectangles exhaustively searched, eliminate a sufficient area of the square assumed in the phase to eventually permit finding the treasure.

In both algorithms, the points to which the agent travels and where it gets hints are chosen in a natural way, although very differently in each of the algorithms. The main difficulty is to prove that the distance traveled by the agent is within the promised cost. In the case of the first algorithm, it is possible to cheaply exclude large areas not containing the treasure, and thus find the treasure asymptotically optimally. For the second algorithm, the agent eliminates smaller areas at each time, due to less precise hints, and thus finding the treasure costs more.

6.2 Preliminaries

Since for $\Delta \leq 1$ treasure hunt is solved immediately, in the sequel we assume $\Delta > 1$. Since the agent has a compass, it can establish an orthogonal coordinate system with point O with coordinates $(0, 0)$ at its starting position, the x-axis going East-West and the y-axis going North-South. Lines parallel to the x-axis will be called horizontal, and lines parallel to the y-axis will be called vertical. When the agent at a current point a decides to go to a previously computed point b (using a straight line), we describe this move simply as "Go to b". A hint given to the agent currently located at point a is formally described as an ordered pair (P_1, P_2) of half-lines originating at a such that the angle clockwise from P_1 to P_2 (including P_1 and P_2) contains the treasure.

The line containing points A and B is denoted by (AB). A segment with extremities A and B is denoted by $[AB]$ and its length is denoted $|AB|$. Throughout the paper, a polygon is defined as a closed polygon (i.e., together with the boundary). For a polygon S, we will denote by $\mathcal{B}(S)$ (resp. $\mathcal{I}(S)$) the boundary of S (resp. the interior of S, i.e., the set $S \setminus \mathcal{B}(S)$). A rectangle is defined as a non-degenerate rectangle, i.e., with all sides of strictly positive length. A rectangle with vertices A, B, C, D (in clockwise order) is denoted simply by $ABCD$. A rectangle is *straight* if one of its sides is vertical.

In our algorithms we use the following procedure `RectangleScan`(R) whose aim is to traverse a closed rectangle R (composed of the boundary and interior) with known coordinates, so that the agent initially situated at some point of R gets at distance at most 1 from every point of it and returns to the starting point. We describe the procedure for a straight rectangle whose vertical side is not shorter than the horizontal side. The modification of the procedure for arbitrarily positioned rectangles is straightforward. Let the vertices of the rectangle R be A, B, C and D, where A is the North-West vertex and the others are listed clockwise. Let a be the point at which the agent starts the procedure.

The idea of the procedure is to go to vertex A, then make a snake-like movement in which consecutive vertical segments are separated by a distance 1, and then go back to point a. The agent ignores all hints gotten during the execution of the procedure. Suppose that the horizontal side of R has length m and the vertical side has length n, with $n \geq m$. Let $k = \lfloor m \rfloor$. Let $a_0, a_1, \ldots, a_k$ be points on the North horizontal side of the rectangle, such that $a_0 = A$ and the distance between consecutive points is 1. Let $b_0, b_1, \ldots, b_k$ be points on the South horizontal side of the rectangle, such that $b_0 = D$ and the distance between consecutive points is 1.

The pseudocode of procedure `RectangleScan`(R) is given in Algorithm 6.1.

Proposition 6.1. *For every point p of the rectangle R, the agent is at distance at most 1 from p at some time of the execution of procedure* `RectangleScan`(R)*. The cost of the procedure is*

Algorithm 6.1 Procedure `RectangleScan`(R)

```
1: if k is odd then
2:     for i = 0 to k − 1 step 2 do
3:         Go to a_i; Go to b_i;
4:         Go to b_{i+1}; Go to a_{i+1}
5:     end for
6:     Go to a
7: else
8:     for i = 0 to k − 2 step 2 do
9:         Go to a_i; Go to b_i;
10:        Go to b_{i+1}; Go to a_{i+1}
11:    end for
12:    Go to a_k; Go to b_k
13:    Go to a
14: end if
```

at most $5n \cdot \max(m, 2)$*, where* $n \geq m$ *are the lengths of the sides of the rectangle.*

Proof. During the execution of procedure `RectangleScan`(R) the agent traverses all segments $[a_i, b_i]$, for $i = 0, 1, \ldots, k$. Every point of R is at distance at most 1 from some point of this union. This proves the first assertion. The cost of vertical moves is upper bounded by $(m+1)n$, the cost of horizontal moves is upper bounded by m, and the cost of getting from a to A and of returning back to a after the scan is upper bounded by $2(m + n)$. Hence the total cost of procedure `RectangleScan`(R) is at most $(m+1)n+m+2(m+n) \leq mn+6n \leq 5n \cdot \max(m, 2)$. □

6.3 Angles at most π

In this section we consider the case when all angles given as hints are at most π. Without loss of generality we can assume that they are all equal to π, completing any smaller angle to π in an arbitrary way: this makes the situation even harder for the agent, as hints become less precise. For such hints we show Algorithm `TreasureHunt1` that finds the treasure at cost $O(\Delta)$. This is of course optimal, as the treasure can be at any point at distance at most Δ from the starting point of the agent.

For angles of size π, every hint is in fact a half-plane whose boundary line L contains the current location of the agent. For simplicity, we will code such a hint as $(L, right)$ or $(L, left)$, whenever the line L is not horizontal, depending on whether the indicated half-plane is to the right (i.e., East) or to the left (i.e., West) of L. For any non-horizontal line L this is non-ambiguous. Likewise, when L is horizontal, we will code a hint as (L, up) or $(L, down)$, depending on whether the indicated half-plane is up (i.e., North) from L or down (i.e., South) from L.

In view of the work on ϕ-self-approaching curves (refer to [3]) we first note that there is a big difference of difficulty between obtaining our result in the case when angles given as hints are bounded by some angle ϕ_0 strictly smaller than π and when they are *at most* π, as we assume. A ϕ-self-approaching curve is a planar oriented curve such that, for each point B on the curve, the rest of the curve lies inside a wedge of angle ϕ with apex in B. In [3], the authors prove the following property of these curves: for every $\phi < \pi$ there exists a constant $c(\phi)$ such that the length of any ϕ-self-approaching curve is at most $c(\phi)$ times the distance Δ between its endpoints. Hence, for hints bounded by some angle ϕ_0 strictly smaller than π, our result could possibly be derived from the existing literature: roughly speaking, the agent should follow a trajectory corresponding to any ϕ_0-self-approaching curve to find the treasure at a cost linear in Δ. Even then, transforming the continuous scenario of self-approaching curves to our discrete scenario presents some difficulties. However, the crucial problem is this: the constant $c(\phi)$ from [3] diverges to infinity as ϕ approaches π, hence the result from [3] cannot be used when hints are

arbitrary angles smaller than π. Moreover, the result of [3] holds only when $\phi < \pi$ (the authors also emphasize that for each $\phi \geq \pi$, the property is false), and thus the above derivation is no longer possible for our purpose when $\phi = \pi$. Actually, this is the real difficulty of our problem: handling angles equal to π, i.e., half-planes.

We further observe that a rather straightforward treasure hunt algorithm of cost $O(\Delta \log \Delta)$, for hints being angles of size π, can be obtained using an immediate corollary of a theorem proven in [66] by Grünbaum: each line passing through the centroid of a convex polygon cuts the polygon into two convex polygons with areas differing by a factor of at most $\frac{5}{4}$. Suppose for simplicity that Δ is known. Starting from the square of side length 2Δ, centered at the initial position of the agent, this permits to reduce the search area from P to at most $\frac{5P}{9}$ in a single move. Hence, after $O(\log \Delta)$ moves, the search area is small enough to be exhaustively searched by procedure `RectangleScan` at cost $O(\Delta)$. However, the cost of each move during the reduction is not under control and can be only bounded by a constant multiple of Δ, thus giving the total cost bound $O(\Delta \log \Delta)$. By contrast, our algorithm controls both the remaining search area and the cost incurred in each move, yielding the optimal cost $O(\Delta)$.

6.3.1 High Level Idea of the Algorithm

In Algorithm `TreasureHunt1` the agent acts in phases $j = 1, 2, 3, \ldots$ where in each phase j the agent "supposes" that the treasure is in a straight square R_j centered at the initial position of the agent, and of side length 2^j. When executing a phase j, the agent successively moves to distinct points with the aim of using the hints at these points to narrow the search area that initially corresponds to R_j. In our algorithm, this narrowing is made in such a way that the remaining search area is always a straight rectangle. Often this straight rectangle is a strict superset of the intersection of all hints that the agent was given previously. This would seem to be a waste, as we are searching some areas that have been previously excluded. However, this loss is compensated by the ease of searching description and subsequent analysis of the algorithm, due to the fact that, at each stage, the search area is very regular.

During a phase, the agent proceeds to successive reductions of the search area by moving to distinct locations, until it obtains a rectangular search area that is small enough to be searched directly at low cost using procedure `RectangleScan`. In our algorithm, such a final execution of `RectangleScan` in a phase is triggered as soon as the rectangle has a side smaller than 4. If the treasure is not found by the end of this execution of procedure `RectangleScan`, the agent learns that the treasure cannot be in the supposed straight square R_j and starts the next phase from scratch by forgetting all previously received hints. This forgetting again simplifies subsequent analysis. The algorithm terminates at the latest by the end of phase $j_0 = \lceil \log_2 \Delta \rceil + 1$, in which the supposed straight square R_{j_0} is large enough to contain the treasure. Hence, if the cost of a phase j is linear in 2^j, then the cost of the overall solution is linear in the distance Δ.

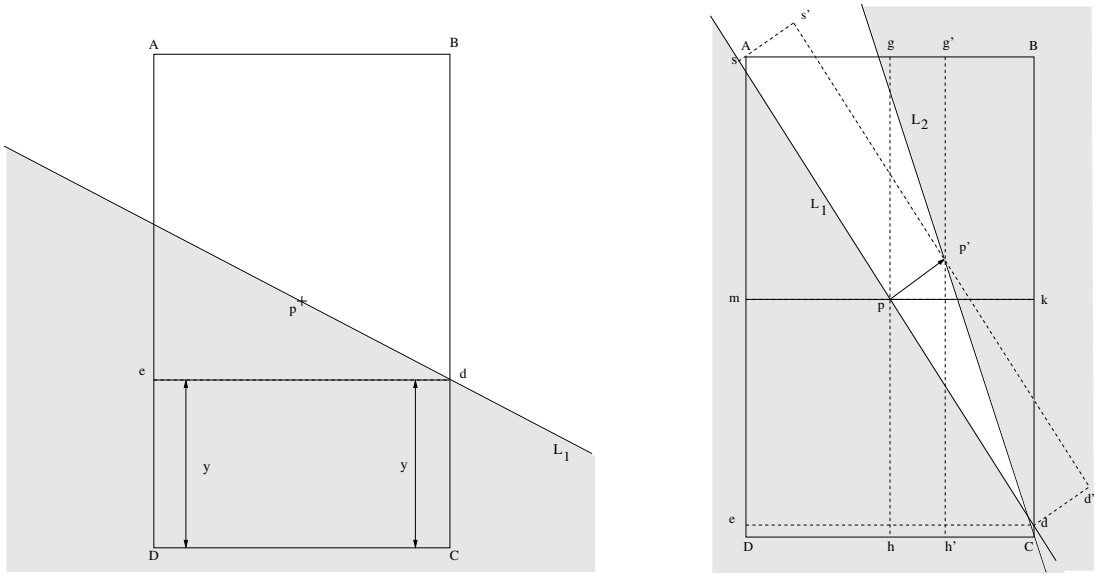
In order to give the reader deeper insights in the reasons why our solution is valid and has linear cost, we need to give more precise explanations on how the search area is reduced during a given phase $j \geq 2$ (when $j = 1$, the agent makes no reduction and directly scans the small search area using procedure `RectangleScan`). Suppose that in phase $j \geq 2$ the agent is at the center p of a search area corresponding to a straight rectangle R, every side of which has length between 4 and 2^j (note that this is the case at the beginning of the phase), and denote by A, B, C and D the vertices of R starting from the top left corner and going clockwise. In order to reduce rectangle R, the agent uses the hint at point p. The obtained hint denoted by (L_1, x_1) can be of two types: either a *good* hint or a *bad* hint. A good hint is a hint whose line L_1 divides one of the sides of R into two segments such that the length y of the smaller one is at least 1. A bad hint is a hint that is not good.

If the received hint (L_1, x_1) is good, then the agent narrows the search area to a rectangle $R' \subset R$ having the following three properties:

1. $R \setminus R'$ does not contain the treasure.
2. The difference between the perimeters of R and R' is $2y \geq 2$.
3. The distance from p to the center of R' is exactly $\frac{y}{2}$.

and then moves to the center of R'.

An illustration of such a reduction is depicted in Figure 6.1(a). The reduced search area R' is the rectangle $ABde$.



(*a*) A good hint $(L_1, right)$ (*b*) A bad hint $(L_1, right)$

Figure 6.1: In Figure (a) the agent received a good hint $(L_1, right)$ at the point p of a rectangular search area $ABCD$. In Figure (b) it received a bad hint $(L_1, right)$ at the point p and hence it moved to point p' and got a hint $(L_2, left)$. In both figures the excluded half-planes are shaded.

If the agent receives a bad hint, say $(L_1, right)$, at the center of a rectangular search area R, we cannot apply the same method as the one used for a good hint: this is the reason for the distinction between good and bad hints. If we applied the same method as before, we could obtain a rectangular search area R' such that the difference between the perimeters of R and R' is at least $2y$. However, in the context of a bad hint, the difference $2y$ may be very small (even null), and hence there is no significant reduction of the search area. In order to tackle this problem, when getting a bad hint at the center p of R, the agent moves to another point p' which is situated in the half-plane $(L_1, right)$ at distance 2 from p, perpendicularly to L_1. This point p' is chosen in such a way that, regardless of what is the second hint, we can ensure that two important properties described below are satisfied.

The first property is that by combining the two hints, the agent can decrease the search area to a rectangle $R' \subset R$ whose perimeter is smaller by 2 compared to the perimeter of R, as it is the case for a good hint, and such that $R \setminus R'$ does not contain the treasure. This decrease follows either directly from the pair of hints, or indirectly after having scanned some relatively small rectangles using procedure `RectangleScan`. In the example depicted in Fig. 6.1 (b), after getting the second hint $(L_2, left)$, the agent executes procedure `RectangleScan`$(ss'd'd)$ followed by `RectangleScan`$(gg'h'h)$ and moves to the center of the new search area R' that is the rectangle $Agpm$. Note that the part of R' not excluded by the two hints and by the procedure `RectangleScan` executed in rectangles $ss'd'd$ and $gg'h'h$ is only the small quadrilateral bounded by line L_2 and the segments $[AB]$, $[s'd']$ and $[gh]$. However, in order to preserve the homogeneity

of the process, we consider the entire new search area R' which is a straight rectangle whose perimeter is smaller by at least 2, compared to that from R. This follows from the fact that no side of R has length smaller than 4. The agent finally moves to the center of R'.

The second property is that all of this (i.e., the move from p to p', the possible scans of small rectangles and finally the move to the center of R') is done at a cost linear in the difference of perimeters of R and R', as shown in Lemma 6.1. The two properties together ensure that, even with bad hints, the agent manages to reduce the search area in a significant way and at a small cost. So, regardless of whether hints are good or not, we can show that the cost of phase j is in $\mathcal{O}(2^j)$ and the treasure is found during this phase if the initial square is large enough. The difficulty of the solution is in showing that the moves prescribed by our algorithm in the case of bad hints guarantee the two above properties, and thus ensure the correctness of the algorithm and the cost linear in Δ.

6.3.2 Algorithm and Analysis

In this subsection we describe our algorithm in detail, prove its correctness and analyze its complexity. Due to many possible positions of the line L from the hint (L, x) obtained by the agent (the line L cutting horizontal or vertical sides of the current search area, the slope of L being positive or negative, and x being $right$, $left$, up or $down$), there are many cases that the algorithm should consider. However, many of these cases can be treated similarly to one another, due to symmetry considerations. Hence, in order to reduce the number of cases, we introduce some geometric transformations that enable us to consider only one representative case in each class. This case will be called a basic configuration.

We define a *configuration* as a couple $(R, (L, x))$, where R is a straight rectangle, and (L, x) is a hint, i.e., a half-plane such that the line L contains the center of R.

A configuration $(R, (L, x))$ is called *lying* iff the line L passes through a point that is in the interior of a vertical side of R. A configuration that is not lying is called *standing*. A configuration $(R, (L, x))$ is called *perfect* iff L is horizontal or vertical. A configuration that is not perfect is called *imperfect*.

A perfect lying (resp. standing) configuration $(R, (L, x))$ can be of two types:

- **Type 1.** $x = up$ (resp. $x = left$)
- **Type 2.** $x = down$ (resp. $x = right$)

An imperfect configuration $(R, (L, x))$ can be of four types:

- **Type 1.** The slope of L is negative and $x = right$
- **Type 2.** The slope of L is negative and $x = left$
- **Type 3.** The slope of L is positive and $x = right$
- **Type 4.** The slope of L is positive and $x = left$

The following proposition follows immediately from the above definitions.

Proposition 6.2. *For every configuration, there exists a unique positive integer $i \leq 4$ such that this configuration is a perfect or imperfect configuration of type i.*

A configuration $(R, (L, x))$ is called *critical* iff the line L divides a side of R into two parts such that the length of the smaller part is less than 1 (possibly 0).

We will denote by $Rot_{v,\alpha}$ the rotation by the angle α with center v, and by Sym_P the axial symmetry with axis P.

The set of all configurations is denoted by $\mathcal{C}$. Given a configuration $(R, (L, x))$, we denote by r and H, respectively, the center of R and the vertical line passing through r. For every $i \in$

$\{0, 1, 2, 3\}$, we define the following functions that are intuitively rotations and axial symmetries of configurations.

$\sigma_i : \mathcal{C} \to \mathcal{C}$ is defined by the formula $\sigma_i((R, (L, x))) = (Rot_{r, \frac{i\pi}{2}}(R), Rot_{r, \frac{i\pi}{2}}((L, x)))$

$\rho : \mathcal{C} \to \mathcal{C}$ is defined by the formula $\rho((R, (L, x))) = (Sym_H(R), Sym_H((L, x)))$

Using the above functions, we now define the following eight *elementary transformations* $\phi_i : \mathcal{C} \to \mathcal{C}$, for $i \in \{0, \ldots, 7\}$.

For $i \in \{0, 1, 2, 3\}$, we have $\phi_i((R, (L, x))) = \sigma_i((R, (L, x)))$.
For $i \in \{4, 5, 6, 7\}$, we have $\phi_i((R, (L, x))) = \rho(\sigma_{i-4}((R, (L, x))))$.

We say that a configuration is *basic* iff it is either a lying perfect configuration of type 1 or a lying imperfect configuration of type 1.

The following proposition asserts that from every configuration we can obtain a basic configuration by at least one of the elementary transformations. This follows directly from the definitions.

Proposition 6.3. *For every configuration* $(R, (L, x))$*, there exists* $i \in \{0, \ldots, 7\}$ *and a basic configuration* $(R', (L', x'))$ *such that* $(R', (L', x')) = \phi_i((R, (L, x)))$

For every configuration, the elementary transformation with the smallest index i for which the above proposition is true will be called the *basic transformation* of this configuration.

Note that, by applying to a configuration $(R, (L, x))$ its basic transformation ϕ_k in order to obtain $(R', (L', x')) = \phi_k((R, (L, x)))$, each point s of (L, x) is rotated and possibly symmetrically reflected to obtain a new point s' in (L', x'). By a slight abuse of notation we will write $s' = \phi_k(s)$ and $s = \phi_k^{-1}(s')$, and, more generally, for any set of points S, we will write $S' = \phi_k(S)$ and $S = \phi_k^{-1}(S')$.

Algorithm 6.2 gives a pseudo-code of our main algorithm. It uses function `ReduceRectangle` described in Algorithm 6.3 that is the key technical tool permitting the agent to reduce its search area. The agent interrupts the execution of Algorithm 6.2 as soon as it gets at distance 1 from the treasure, at which point it can "see" it and thus treasure hunt stops.

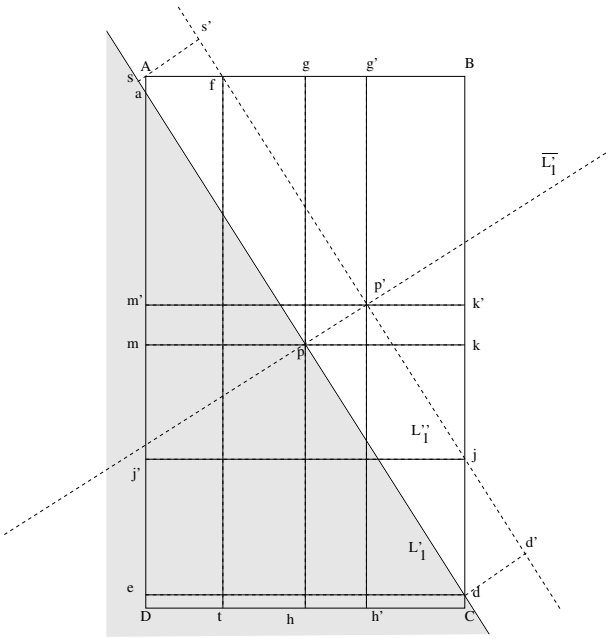


Figure 6.2: Illustration of the geometric objects used in Algorithm 6.3 and in the proof of Lemma 6.1. We show an example of a basic configuration $(R', (L'_1, x'_1))$ that is critical, in which R' is the rectangle $ABCD$ and $x'_1 = right$. We also show projections and intersections points defined in Algorithm 6.3. The excluded area is shaded.

Algorithm 6.2 Procedure `TreasureHunt1`

```
1: O ← the initial position of the agent
2: i ← 1
3: loop
4:     R_i ← the straight square centered at O with sides of length 2^i
5:     while R_i has no side with length smaller than 4 do
6:         R_i ← ReduceRectangle(R_i)
7:     end while
8:     execute RectangleScan(R_i)
9:     go to O
10:    i ← i + 1
11: end loop
```

We now proceed to the proof of correctness and the complexity analysis of our algorithm. In the following lemma, for every rectangle R, the function $Perimeter(R)$ returns the perimeter of the rectangle R.

Lemma 6.1. *Let R be a straight rectangle with no side of length less than 4. If the agent executes* `ReduceRectangle`(R) *from the center of R, then at the end of this execution the following properties are satisfied.*

1. *The function* `ReduceRectangle`(R) *returns a straight rectangle Rec, such that $Rec \subset R$, and either $R \setminus Rec$ does not contain the treasure or the agent has seen the treasure.*
2. *$Perimeter(R) - Perimeter(Rec) \geq 2$.*
3. *The agent is at the center of rectangle Rec.*
4. *The agent traveled a distance of at most $21(Perimeter(R) - Perimeter(Rec))$ during the execution of* `ReduceRectangle`(R).

Proof. Most of the geometric objects used in the proof are explicitly defined in Algorithm 6.3: in particular, this is the case of intersections or orthogonal projections (e.g., those in lines 10 to 21). All other necessary objects will be defined within the proof. For the notation, refer to Fig. 6.2.

Consider the execution of function `ReduceRectangle`(R) starting at the center p of R, where R is a straight rectangle with no sides of length less than 4. Denote by z the position of the treasure in (L_1, x_1). We have $(R', (L'_1, x'_1)) = \phi_k((R, (L_1, x_1)))$. In view of Proposition 6.3, it is enough to prove that the following three properties hold when the agent executes the last line of Algorithm 6.3.

- **P1**. The variable $NewRectangle$ is set to a straight rectangle R'' such that $R'' \subset R'$, and either $R' \setminus R''$ does not contain $\phi_k(z)$ or the agent has seen the treasure.
- **P2**. The inequality $Perimeter(R') - Perimeter(R'') \geq 2$ holds.
- **P3**. The agent traveled a distance of at most $21(Perimeter(R) - Perimeter(R''))$ during the execution of function `ReduceRectangle`(R).

We first prove the above properties when $(R', (L'_1, x'_1))$ is a non-critical configuration. In this case, the variable $NewRectangle$ is set to the straight rectangle $ABde$. Note that the points defined in lines 4 to 6 (in particular the points A,B, d and e) exist and $ABde$ is a straight rectangle such that $ABde \subset R'$ in view of the fact that $(R', (L'_1, x'_1))$ is a basic configuration. Moreover, since $z \in (L_1, x_1)$, we have $\phi_k(z) \in (L'_1, x'_1)$. However, $edCD \cap (L'_1, x'_1) \subset [de]$ and $R' \setminus ABde = edCD \setminus [de]$. So we have $(R' \setminus ABde) \cap (L'_1, x'_1) = \emptyset$ and Property P1 is satisfied. Property P2 also holds because $(R', (L'_1, x'_1))$ is a basic configuration that is not

Algorithm 6.3 Function `ReduceRectangle`(R)

$p \leftarrow$ the center of rectangle R
let (L_1, x_1) and ϕ_k be respectively the hint obtained at p and the basic transformation of $(R, (L_1, x_1))$
let $(R', (L'_1, x'_1))$ be the configuration such that $(R', (L'_1, x'_1)) = \phi_k((R, (L_1, x_1)))$
let A,B,C and D be the vertices of R' in clockwise order, starting from the top-left corner
let a (resp. d) be the intersection between L'_1 and (AD) (resp. (BC))
let e be the orthogonal projection of d onto segment $[AD]$
if $(R', (L'_1, x'_1))$ is not critical **then**
    $NewRectangle \leftarrow$ rectangle $ABde$
else
    let $\overline{L'_1}$ be the line that is perpendicular to L'_1
    let p' be the point at distance 2 from p in $\overline{L'_1} \cap (L'_1, x')$
    let L''_1 be the parallel line to L'_1 passing through p'
    let f (resp. j) be the intersection of L''_1 and segment $[AB]$ (resp. segment $[BC]$)
    let j' be the orthogonal projection of j onto segment $[AD]$
    let t be the orthogonal projection of f onto segment $[DC]$
    let m' (resp. k') be the orthogonal projection of p' onto segment $[AD]$ (resp. $[BC]$)
    let m (resp. k) be the orthogonal projection of p onto segment $[AD]$ (resp. $[BC]$)
    let g' (resp. h') be the orthogonal projection of p' onto segment $[AB]$ (resp. $[DC]$)
    let g (resp. h) be the orthogonal projection of p onto segment $[AB]$ (resp. $[DC]$)
    let s (resp. s') be the orthogonal projection of A onto line L'_1 (resp. L''_1)
    let d' be the orthogonal projection of d onto line L''_1
    go to $\phi_k^{-1}(p')$
    let (L_2, x_2) be the hint obtained at $\phi_k^{-1}(p')$ and let $(L'_2, x'_2) = \phi_k^{-1}((L_2, x_2))$
    if $x'_2 = right$ and L'_2 is clockwise between L''_1 (included) and (pp') (excluded) **then**
        $NewRectangle \leftarrow$ rectangle $fBCt$
    end if
    if $x'_2 = right$ and L'_2 is clockwise between (pp') (included) and $(m'k')$ (excluded) **then**
        execute `RectangleScan`$(\phi_k^{-1}(m'k'km))$
        $NewRectangle \leftarrow$ rectangle $gBCh$
    end if
    if $x'_2 \in \{down, left\}$ and L'_2 is clockwise between $(m'k')$ (included) and L''_1 (excluded) **then**
        execute `RectangleScan`$(\phi_k^{-1}(ss'd'd))$
        execute `RectangleScan`$(\phi_k^{-1}(m'k'km))$
        $NewRectangle \leftarrow$ rectangle $pkCh$
    end if
    if $x'_2 = left$ and L'_2 is clockwise between L''_1 (included) and $(g'h')$ (excluded) **then**
        execute `RectangleScan`$(\phi_k^{-1}(ss'd'd))$
        execute `RectangleScan`$(\phi_k^{-1}(gg'h'h))$
        $NewRectangle \leftarrow$ rectangle $Agpm$
    end if
    if ($x'_2 = left$ and L'_2 is clockwise between $(g'h')$ (included) and (pp') (excluded)) **or** ($x'_2 = left$ and L'_2 is clockwise between (pp') (included) and $(m'k')$ (excluded)) **or** ($x'_2 \in \{up, right\}$ and L'_2 is clockwise between $(m'k')$ (included) and $(p'k)$ (excluded)) **then**
        execute `RectangleScan`$(\phi_k^{-1}(gg'h'h))$
        $NewRectangle \leftarrow$ rectangle $ABkm$
    end if
    if $x'_2 = right$ and L'_2 is clockwise between $(p'k)$ (included) and L''_1 (excluded) **then**
        $NewRectangle \leftarrow$ rectangle $ABjj'$
    end if
end if
let o' be the center of $NewRectangle$
go to $\phi_k^{-1}(o')$
return $\phi_k^{-1}(NewRectangle)$

critical. Indeed, in that case we know that the length $|Bd| \leq |BC| - 1$, as $|dC| \geq 1$. Hence, $|Ae| + |Bd| \leq |AD| + |BC| - 2$, and thus $Perimeter(R') - Perimeter(ABde) \geq 2$. It remains to prove Property P3. If we denote by δ the difference $|BC| - |Bd|$, the distance from $\phi_k(p) = p$ to the center o' of rectangle ABde is exactly $\frac{\delta}{2}$. Moreover, the distance from p to $\phi_k^{-1}(o')$ is also $\frac{\delta}{2}$, as ϕ_k^{-1} is a distance-preserving transformation. As a result, since the only movement of the agent is from p to $\phi_k^{-1}(o')$ and $\delta = \frac{Perimeter(R') - Perimeter(ABde)}{2}$, when the agent executes the last line of Algorithm 6.3, it has traveled a distance of $\frac{Perimeter(R') - Perimeter(ABde)}{4}$ during the execution of function `ReduceRectangle`(R). Thus the lemma holds if $(R', (L'_1, x'_1))$ is a non-critical configuration.

Let us now consider the more difficult situation when $(R', (L'_1, x'_1))$ is a critical configuration. In Algorithm 6.3, this situation is handled by moving the agent to the point $\phi_k^{-1}(p')$ (refer to line 22) where p' is the point defined at line 11, in order to get a second hint (L_2, x_2) at $\phi_k^{-1}(p')$. We have six cases to consider depending on the nature of (L_2, x_2). Similarly as for non-critical configurations, we do not study the six cases directly on (L_2, x_2), but on (L'_2, x'_2) instead, where (L'_2, x'_2) is such that $(L'_2, x'_2) = \phi_k((L_2, x_2))$. Note that if the list of cases for (L'_2, x'_2) covers all possible situations, and in each of those cases Properties P1 to P3 are satisfied, then the lemma will be proven.

The six cases correspond to the six conditional statements that are in lines 24 to 45 of Algorithm 6.3. The fact that these cases cover all possible situations follows from the fact that $(R', (L'_1, x'_1))$ is a basic configuration, by Proposition 6.3, and from the fact that the objects defined in lines 10 to 21 of Algorithm 6.3 exist. In turn, the existence of these objects follows from the definition of R' and of (L'_1, x'_1) as well as from the following three claims (note that R' has no side with length less than 4, as ϕ_k is a distance-preserving transformation and R has no side with length less than 4).

Claim 6.1. *$(R', (L'_1, x'_1))$ is an imperfect lying configuration of type 1.*

Proof of the claim: Since $(R', (L'_1, x'_1))$ is basic, we just have to show that it is not a perfect lying configuration of type 1. Suppose by contradiction that it is. So, $x'_1 = up$ and line L'_1 divides the west vertical side $[AD]$ (resp. the east vertical side $[BC]$) of rectangle R' into two parts of equal length. Since, $(R', (L'_1, x'_1))$ is critical, each of these parts has length less than 1. As a result, $|AD|$ (resp. $|BC|$) is smaller than 2. This implies that R' has a side with a length smaller than 4, which is a contradiction and concludes the proof of the claim. ⋆

Claim 6.2. *The point p' belongs to $\mathcal{I}(R')$.*

Proof of the claim: Since p is the center of rectangle R' that has no side of length less than 4, every point that is at distance at most 2 from p, and which is not one of the four orthogonal projections of p on the sides of R', necessarily belongs to $\mathcal{I}(R')$. However, by Algorithm 6.3, point p' is at distance 2 from p on a line perpendicular to L'_1 and that passes through p. Moreover, by Claim 6.1, $(R', (L'_1, x'_1))$ is an imperfect vertical configuration of type 1, and thus the slope of L'_1 is negative. Hence the claim holds. ⋆

Claim 6.3. *The line L''_1 divides the northern side $[AB]$ (resp. the east side $[BC]$) of R' into two parts of positive length.*

Proof of the claim: In view of Claim 6.1 and the fact that L''_1 is a line parallel to L'_1 passing through p' that is a point belonging to $\mathcal{I}(R')$ (refer to Claim 6.2), it follows that L''_1 divides the east side $[BC]$ of R' into two parts of positive length. It also follows that L''_1 intersects the northern side $[AB]$ or the west side $[AD]$ of R'. So to prove the claim, it is enough to show that L''_1 cannot intersect $[AD]$ (i.e., cannot pass through any points of $[AD]$ including the corners A and D). Assume by contradiction that it does. Since $L''_1 \subset (L'_1, x'_1)$ and the distance from any point of L'_1 to any point of L''_1 is at least $|pp'|$, then according to the definition of L'_1 and L''_1, we know that the segment $[AD] \cap (L'_1, x'_1)$ has a length that is at least $|pp'| = 2$. However, by

Claim 6.1 and the fact that $(R', (L'_1, x'_1))$ is a critical configuration, the segment $[AD] \cap (L'_1, x'_1)$ has a length that is smaller than 1, which is a contradiction and proves the claim. ⋆

Hence, since we have a list of six cases covering all possible situations, it is enough to show that Properties P1 to P3 are satisfied in each case, in order to conclude the proof of the lemma. Before analyzing them, let us give another claim that will be useful in the sequel.

Claim 6.4. *The length of segment* $[Af]$ *(resp.* $[jC]$*) is at least* 1.

Proof of the claim: As mentioned previously, the distance from any point of L'_1 to any point of L''_1 is at least $|pp'| = 2$. Hence, $|af| \geq 2$ and $|jd| \geq 2$. Since $d \in [jC]$ and $|aA| < 1$ (because the configuration is critical) and $[af]$ is the hypotenuse of the right triangle Afa, the claim follows. ⋆

The fact that each object that is assigned to variable $NewRectangle$ or given as input parameter to procedure `RectangleScan` is a rectangle, can be shown using the above claims. Moreover, from the definitions of intersections and projections given in Algorithm 6.3, it follows that each time procedure `RectangleScan` is called with an input parameter corresponding to a rectangle X, the agent is in the rectangle X (this is necessary in order to obtain a correct execution of the procedure). In the rest of the proof we will not mention this fact. Similarly, it follows that a rectangle that is assigned to variable $NewRectangle$ is always straight.

Now, we consider the six cases and we start with the first one in which $x'_2 = right$ and L'_2 is clockwise between L''_1 (included) and (pp') (excluded). In this case, variable $NewRectangle$ is set to the straight rectangle $fBCt \subset R'$. Since $z \in (L_1, x_1) \cap (L_2, x_2)$, we have $\phi_k(z) \in (L'_1, x'_1) \cap (L'_2, x'_2)$. However, $R' \setminus fBCt = AftD \setminus [ft]$, and in view of the value of x'_2 and the position of L'_2, we have $AftD \cap (L'_1, x'_1) \cap (L'_2, x'_2) \subseteq \{f\}$ (more precisely, $AftD \cap (L'_1, x'_1) \cap (L'_2, x'_2) = \{f\}$ if $L'_2 = L''_1$, and $AftD \cap (L'_1, x'_1) \cap (L'_2, x'_2) = \emptyset$ for all the other positions of L'_2 within the considered case). Hence, $(R' \setminus fBCt) \cap (L'_1, x'_1) \cap (L'_2, x'_2) = \emptyset$ and Property P1 is satisfied. Concerning Property P2, we know that $|fB| = |AB| - |Af|$, which implies $|fB| \leq |AB| - 1$ because $|Af| \geq 1$ according to Claim 6.4. So, $Perimeter(R') - Perimeter(fBCt) \geq 2$, and thus Property P2 holds. Concerning Property P3, we need to evaluate the distance traveled by the agent when it moves from p to $\phi_k^{-1}(p')$, and then from $\phi_k^{-1}(p')$ to $\phi_k^{-1}(o')$ (where o' is the center of rectangle $fBCt$). Note that the distance from p to $\phi_k^{-1}(o')$ is $\frac{\delta}{2}$ where δ is the difference $|AB| - |fB|$. Moreover, $|pp'| = |p\phi_k^{-1}(p')| = 2$. Hence $|p\phi_k^{-1}(p')| \leq 2\delta$ because $|AB| - |fB| = |Af|$ and $|Af| \geq 1$ according to Claim 6.4. Thus, moving from p to $\phi_k^{-1}(p')$ makes the agent travel a distance of at most 2δ. Moving from $\phi_k^{-1}(p')$ to $\phi_k^{-1}(o')$ makes the agent travel a distance that is upper-bounded by $|\phi_k^{-1}(p')p| + |p\phi_k^{-1}(o')| \leq \frac{5\delta}{2}$. As a result, the total distance traveled by the agent is at most $\frac{9\delta}{2} = \frac{9(Perimeter(R') - Perimeter(fBCt))}{4}$, as $\delta = \frac{Perimeter(R') - Perimeter(fBCt)}{2}$. Hence Properties P1, P2 and P3 hold in this case.

Let us now consider the situation when $x'_2 = right$ and L'_2 is clockwise between (pp') (included) and $(m'k')$ (excluded). The variable $NewRectangle$ is then set to the straight rectangle $gBCh \subset R'$. Note that $R' \setminus gBCh \subset AghD$. In view of the value of x'_2 and the position of L'_2, $AghD \cap (L'_1, x'_1) \cap (L'_2, x'_2)$ is included in the rectangle $m'k'km$. Since $\phi_k(z) \in (L'_1, x'_1) \cap (L'_2, x'_2)$, if the agent has not seen the treasure after having executed `RectangleScan`$(\phi_k^{-1}(m'k'km))$, then in view of Proposition 6.1 and the definition of transformation ϕ_k we know that $\phi_k(z)$ cannot be in the rectangle $AghD$. Thus, Property P1 is satisfied. Property P2 follows from the facts that $|gB| = \frac{|AB|}{2}$ (since g is the orthogonal projection of the center p of R' on the top side $[AB]$ of R') and that $|AB| \geq 4$ (as R' has no side of length less than 4). So, it remains to check the validity of Property P3 in the current case. The move of the agent is composed of three parts: the first part is when it moves from p to $\phi_k^{-1}(p')$, the second part corresponds to the move made when executing procedure `RectangleScan`$(\phi_k^{-1}(m'k'km))$, and the third part is when the agent moves to $\phi_k^{-1}(o')$ (where o' is the center of the rectangle $gBCh$). Note that the execution of `RectangleScan`$(\phi_k^{-1}(m'k'km))$ starts and finishes at point $\phi_k^{-1}(p')$. This implies that the third part corresponds precisely to a

move from point $\phi_k^{-1}(p')$ to point $\phi_k^{-1}(o')$. So, by similar arguments to those used in the previous case, we can show that the distance traveled in the first part plus the distance traveled in the third part gives a total of at most $\frac{9\delta}{2}$ where, in the current situation, δ is the difference $|AB| - |gB|$. For the second part, corresponding to the execution of procedure `RectangleScan`$(\phi_k^{-1}(m'k'km))$, note that since $|p\phi_k^{-1}(p')| = 2$, we have $|kk'| \leq 2$ because k (resp. k') is the orthogonal projection of p (resp. p') onto $[BC]$. Moreover, $|m'k'| = |AB|$, and in view of the definition of R', we have $|AB| \geq 4$. Hence, according to Proposition 6.1, we know that the agent travels a distance of at most $10|AB|$ during the second part. So the total distance traveled by the agent is at most $\frac{9\delta}{2} + 10|AB|$. As explained for Property P2, we know that $|gB| = \frac{|AB|}{2}$. Hence, $\delta = \frac{|AB|}{2}$, $Perimeter(R') - Perimeter(gBCh) = |AB|$, and thus the total distance traveled by the agent is at most $\frac{49(Perimeter(R') - Perimeter(gBCh))}{4}$. As a result, Properties P1, P2 and P3 hold in this case.

We continue by analyzing the situation when $x_2' \in \{down, left\}$ and L_2' is clockwise between $(m'k')$ included and L_1'' (excluded). In this situation, variable $NewRectangle$ is set to the straight rectangle $pkCh \subset R'$. In view of the value of x_2' and the position of L_2', we have $(L_1', x_1') \cap (L_2', x_2') \cap (R' \setminus pkCh) \subset (ss'd'd \cup m'k'km)$. Since $\phi_k(z) \in (L_1', x_1') \cap (L_2', x_2')$, if the agent has not seen the treasure after having executed `RectangleScan`$(\phi_k^{-1}(ss'd'd))$ followed by `RectangleScan`$(\phi_k^{-1}(m'k'km))$, then in view of Proposition 6.1 and the definition of transformation ϕ_k we know that $\phi_k(z)$ cannot be in $R' \setminus pkCh$. Thus, Property P1 is satisfied. We can show that Property P2 also holds by similar arguments to those used to show Property P2 in the previous case. Concerning Property P3, note that the move of the agent can be divided into four parts: the first part is when it moves from p to $\phi_k^{-1}(p')$, the second (resp. third) part corresponds to the move made when executing procedure `RectangleScan`$(\phi_k^{-1}(ss'd'd))$ (resp. `RectangleScan`$(\phi_k^{-1}(m'k'km))$), and the fourth part is when the agent moves to $\phi_k^{-1}(o')$ (where o' is here the center of rectangle $pkCh$). Note that the execution of `RectangleScan`$(\phi_k^{-1}(ss'd'd))$ (resp. `RectangleScan`$(\phi_k^{-1}(m'k'km))$) starts and finishes at point $\phi_k^{-1}(p')$. So, the fourth part is actually a move from point $\phi_k^{-1}(p')$ to point $\phi_k^{-1}(o')$. It is worth mentioning that moving from $\phi_k^{-1}(p')$ to point $\phi_k^{-1}(o')$ costs the same or less than first moving from $\phi_k^{-1}(p')$ to p, and then moving from p to $\phi_k^{-1}(o')$. Moreover, moving from p to $\phi_k^{-1}(o')$ costs at most $\frac{\delta_1+\delta_2}{2}$ where δ_1 (resp. δ_2) is the difference $|gh| - |pg|$ (resp. $|mk| - |pm|$). Hence during the fourth part, the agent travels a distance of at most $2 + \frac{\delta_1+\delta_2}{2}$. During the first part, the agent travels a distance 2. What about the second and third parts? To evaluate these costs we need to evaluate the lengths and widths of rectangles $ss'd'd$ and $m'k'km$. In the analysis of the previous case, we have shown that the length and width of rectangle $m'k'km$ are respectively $|AB|$ and at most 2. Concerning rectangle $ss'd'd$, we have the following claim.

Claim 6.5. *$|ss'| = 2$ and $2 < |sd| < 1 + |AC|$.*

Proof of the claim: Note that $|ss'|$ is exactly 2 because s (resp. s') is the orthogonal projection of the corner A onto line L_1' (resp. L_1''). Also note that $|sd| = |sa| + |ad|$ where $[sa]$ is a side of the right triangle asA whose hypotenuse is $[Aa]$. However, by Claim 6.1 and the fact that $(R', (L_1', x_1'))$ is critical, we know that $|Aa| < 1$. Moreover, $[ad] \subset R'$ and $|ad| \geq |AB| \geq 4$. Hence, $2 < |sd| < 1 + |AC|$, which concludes the proof of the claim. ⋆

As a result, according to Proposition 6.1, we know that the agent travels a distance of at most $10(1 + |AC|)$ during the second part and a distance of at most $10|AB|$ during the third part. Hence, the total distance traveled by the agent is at most $2 + \frac{\delta_1+\delta_2}{2} + 10(|AB| + |AC| + 1)$. Note that $|gh| - |pg| = \frac{|AD|}{2}$, $|mk| - |pm| = \frac{|AB|}{2}$ and $|AC| < |AB| + |AD|$. Furthermore, in view of the fact that R' has no side of length less than 4, we have $\frac{|AD|}{2} \geq 2$ and $\frac{|AB|}{2} \geq 2$. So, the total distance traveled by the agent is at most $\frac{|AD|}{2} + \frac{|AD|+|AB|}{4} + 10(2|AB| + |AD| + 1)$. This in turn gives us a traveled distance that is upper-bounded by $21(Perimeter(R') - Perimeter(pkCh))$ as $Perimeter(R') - Perimeter(pkCh) = 2(\frac{|AD|}{2} + \frac{|AB|}{2}) = |AD| + |AB|$. Consequently, Properties P1, P2 and P3 are valid in this case.

So far, we have analyzed the first three cases among the six cases that permit us to cover entirely the situation when $(R', (L'_1, x'_1))$ is a critical configuration. However, the arguments we need to use in order to analyze the last three cases are similar to those already used to analyze the first three cases. In particular, this is true for the fourth case when $x'_2 = left$ and L'_2 is clockwise between L''_1 (included) and $(g'h')$ (excluded): using a similar reasoning to that for the third case we have analyzed just above, we can show that Properties P1, P2 and P3 are also valid here. For the fifth case, which corresponds to the boolean expression of line 41, Properties P2 and P3 can be proven using a similar reasoning to that used above to prove Properties P2 and P3 when $x'_2 = right$ and L'_2 is clockwise between (pp') (included) and $(m'k')$ (excluded). Concerning Property P1, note that variable $NewRectangle$ is set here to the straight rectangle $ABkm \subset R'$. In view of the possible values of x'_2 and the possible positions of L'_2, we have $(L'_1, x'_1) \cap (L'_2, x'_2) \cap (R' \setminus ABkm) \subset gg'h'h$ if $x'_2 = left$ and L'_2 is clockwise between $(g'h')$ (included) and (pp') (excluded). Otherwise, we have $(L'_1, x'_1) \cap (L'_2, x'_2) \cap (R' \setminus ABkm) = \emptyset$. Since $\phi_k(z) \in (L'_1, x'_1) \cap (L'_2, x'_2)$, if the agent has not seen the treasure after having executed `RectangleScan`$(\phi_k^{-1}(gg'h'h))$, then in view of Proposition 6.1 and the definition of transformation ϕ_k we know that $\phi_k(z)$ cannot be in $R' \setminus ABkm$. Thus Property P1 is also true in this case. Finally, the fact that Properties P1, P2 and P3 are true in the last of the six cases i.e., when $x'_2 = right$ and L'_2 is clockwise between $(p'k)$ (included) and L''_1 (excluded) can be proven using similar arguments to those used for the first case, i.e., when $x'_2 = right$ and L'_2 is clockwise between L''_1 (included) and (pp') (excluded). This completes the proof of the lemma. □

Theorem 6.1. *Consider an agent A and a treasure located at distance at most Δ from the initial position of A. By executing Algorithm* `TreasureHunt1`*, agent A finds the treasure after having traveled a distance* $O(\Delta)$.

Proof. The execution of Algorithm 6.2 can be divided into phases $1, 2, 3, \ldots$ where phase $j \geq 1$ is the part of the execution in which variable i of Algorithm 6.2 is equal to j.

In view of the second and third properties of Lemma 6.1 and lines 4 to 7 of Algorithm 6.2, the number of calls to function `ReduceRectangle` is bounded by the perimeter of a square with side length 2^j. Hence we have the following claim.

Claim 6.6. *For every $j \geq 1$, the number of calls to function* `ReduceRectangle`*, within phase j, is bounded by 2^{j+2}.*

In order to conclude the proof of the theorem, it is enough to prove the following two statements:

1. for all $j \geq 1$, the following property $\mathcal{H}_j$ holds:
   at the beginning of phase j the agent has traveled a distance of at most 2^{j+7}.

2. the agent finds the treasure before starting phase $\lceil \log_2 \Delta \rceil + 2$.

We start by proving the first statement by induction on j. Note that property $\mathcal{H}_1$ is true because at the beginning of phase 1 the agent has traveled a distance 0. So, assume that, for a positive integer λ, property $\mathcal{H}_\lambda$ is true. We prove that property $\mathcal{H}_{\lambda+1}$ is also true. Within phase λ, the move of the agent can be divided into two parts: the first part corresponds to the moves made when executing lines 4 to 7 of Algorithm 6.2, while the second part corresponds to the moves made when executing lines 8 and 9 of Algorithm 6.2. By Claim 6.6, we know that the number τ of calls to function `ReduceRectangle` during phase λ is upper-bounded by $2^{\lambda+2}$. For all $1 \leq s \leq \tau$, we denote by Q_s (resp. Q'_s) the rectangle that is the input parameter (resp. the returned value) of the sth call to function `ReduceRectangle` during phase λ. Note that, for all $2 \leq s \leq \tau$, $Q_s = Q'_{s-1}$. So, by the fourth property of Lemma 6.1, the distance traveled by the agent during the first part of phase λ is upper-bounded by

$$21 \sum_{s=1}^{s=\tau} (Perimeter(Q_s) - Perimeter(Q'_s)) \tag{6.1}$$

$$\leq 21(Perimeter(Q_1) - Perimeter(Q'_\tau) + \sum_{s=2}^{s=\tau} (Perimeter(Q_s) - Perimeter(Q'_{s-1}))) \tag{6.2}$$

$$\leq 21(Perimeter(Q_1) - Perimeter(Q'_\tau)) \qquad \text{(because for all } 2 \leq s \leq \tau,\ Q_s = Q'_{s-1}) \tag{6.3}$$

$$\leq 21 Perimeter(Q_1) = 21 \cdot 2^{\lambda+2} \tag{6.4}$$

Concerning the second part of phase λ, it is worth mentioning that when the agent starts executing line 8 of Algorithm 6.2, variable R_i is set to a straight rectangle whose at least one side has length smaller than 4 (according to line 5), and no sides have length larger than 2^λ: indeed, using the first property of Lemma 6.1, it follows by induction on s that the straight rectangle Q'_s is included in the straight rectangle Q_1, for all $1 \leq s \leq \tau$. Moreover, the distance between any two points of Q_1 (and thus the cost of line 9 of Algorithm 6.2) is at most $2^{\lambda+1}$. Hence, in view of Proposition 6.1, we know that the distance traveled by the agent during the second part of phase λ is upper-bounded by $22 \cdot 2^\lambda$. From this and (6.4), we know that the total distance traveled during phase λ is at most $2^{\lambda+7}$. Moreover, by the inductive hypothesis, $\mathcal{H}_\lambda$ is true i.e., at the beginning of phase λ the agent has traveled a distance of at most $2^{\lambda+7}$. As a result, when starting phase $\lambda+1$, the agent has traveled a total distance of at most $2^{\lambda+8}$. Thus, property $\mathcal{H}_{\lambda+1}$ is true, which concludes the inductive proof and thus proves the validity of the first statement.

Now let us focus on the second statement: the agent finds the treasure before starting phase $\lceil \log_2 \Delta \rceil + 2$. Suppose by contradiction that this is not the case. By Claim 6.6 and Lemma 6.1, at some point the agent starts executing phase $\lceil \log_2 \Delta \rceil + 1$. In view of Algorithm 6.2, when the agent finishes the execution of line 4 in phase $\lceil \log_2 \Delta \rceil + 1$, the value of variable R_i is a square S containing the treasure: indeed this square is centered at the initial position O of the agent and it contains all points at distance at most Δ from O because its side length is $2^{\lceil \log_2 \Delta \rceil + 1} \geq 2\Delta$, since $\Delta > 1$.

Denote by Q_{final} the rectangle returned by the last call to function `ReduceRectangle` in phase $\lceil \log_2 \Delta \rceil + 1$: since the side length of S is at least $2^{\lceil \log_2 \Delta \rceil + 1} \geq 2^2$, this rectangle exists because the agent executes at least once line 6 of Algorithm 6.2. By Claim 6.6 and Lemma 6.1, at some point the agent executes line 8 of Algorithm 6.2 and when the agent starts executing this line we know that it is at the center of Q_{final}. Moreover, from Lemma 6.1, it follows by induction on the number of calls to function `ReduceRectangle` within phase $\lceil \log_2 \Delta \rceil + 1$, that the treasure does not belong to $S \backslash Q_{final}$, as otherwise the agent would have found the treasure before starting phase $\lceil \log_2 \Delta \rceil + 2$ which would be a contradiction. However, the treasure belongs to square S. Hence, the treasure belongs to Q_{final}, and by applying procedure `RectangleScan`(Q_{final}) (refer to line 8) from the center of Q_{final}, the agent necessarily finds the treasure by the end of the execution of this procedure, and thus by the end of phase $\lceil \log_2 \Delta \rceil + 1$. This gives a contradiction that proves the second statement.

Hence the agent finds the treasure before starting the execution of phase $\lceil \log_2 \Delta \rceil + 2$. By the first statement, the total distance traveled by the agent during the first $\lceil \log_2 \Delta \rceil + 1$ phases is at most $2^{(\lceil \log_2 \Delta \rceil + 2) + 7} \leq 2^{10}\Delta$. Hence, the theorem holds. □

6.4 Angles Bounded by $\beta < 2\pi$

In this section we consider the case when all hints are angles upper-bounded by some constant $\beta < 2\pi$, unknown to the agent. The main result of this section is procedure `TreasureHunt2` whose cost is at most $O(\Delta^{2-\epsilon})$, for some $\epsilon > 0$. For a hint (P_1, P_2) we denote by $\overline{(P_1, P_2)}$ the complement of (P_1, P_2).

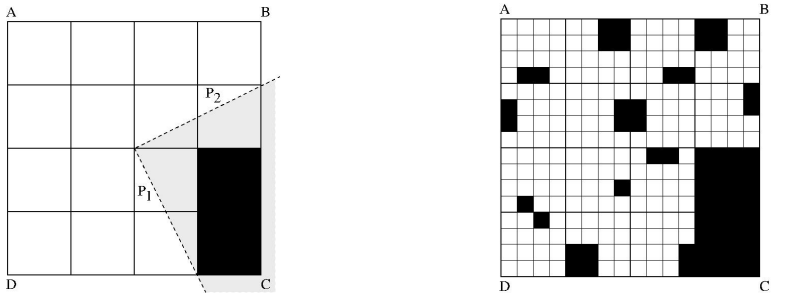
6.4.1 High Level Idea

In procedure `TreasureHunt2`, similarly as in the previous procedure, the agent acts in phases $j = 1, 2, 3, \ldots$, where in each phase j the agent "supposes" that the treasure is in the straight square centered at its initial position and of side length 2^j. The intended goal is to search each supposed square at relatively low cost, and to ensure the discovery of the treasure by the time the agent finishes the first phase for which the initial supposed square contains the treasure. However, the similarity with the previous solution ends there: indeed, the hints that may now be less precise do not allow us to use the same strategy within a given phase. Hence we adopt a different approach that we outline below and that uses the following notion of tiling. Given a square S with side of length $x > 0$, $Tiling(i)$ of S, for any non-negative integer i, is the partition of square S into 4^i squares with side of length $\frac{x}{2^i}$. Each of these squares, called *tiles*, is closed, i.e., contains its border, and hence neighboring tiles overlap in the common border.

Let us consider a simpler situation in which the angle of every hint (P_1, P_2) is always equal to the bound β: the general case, when the angles may vary while being at most β, adds a level of technical complexity that is unnecessary to understand the intuition. In the considered situation, the angle of each excluded zone $\overline{(P_1, P_2)}$ is always the same as well. The following property holds in this case: there exists an integer i_β such that for every square S and every hint (P_1, P_2) given at the center of S, at least one tile of $Tiling(i_\beta)$ of S belongs to the excluded zone $\overline{(P_1, P_2)}$.

In phase j, the agent performs k steps: we will indicate later how the value of k should be chosen. At the beginning of the phase, the entire square S is white. In the first step, the agent gets a hint (P_1, P_2) at the center of S. By the above property, we know that $\overline{(P_1, P_2)}$ contains at least one tile of $Tiling(i_\beta)$ of S, and we have the guarantee that such a tile cannot contain the treasure. All points of all tiles included in $\overline{(P_1, P_2)}$ are painted black in the first step. This operation does not require any move, as painting is performed in the memory of the agent. As a result, at the end of the first step, each tile of $Tiling(i_\beta)$ of S is either black or white, in the following precise sense: a black tile is a tile all of whose points are black, and a white tile is a tile all of whose *interior* points are white.

In the second step, the agent repeats the painting procedure at a finer level. More precisely, the agent moves to the center of each white tile t of $Tiling(i_\beta)$ of S. When it gets a hint at the center of a white tile t, there is at least one tile of $Tiling(i_\beta)$ of t that can be excluded. As in the first step, all points of these excluded tiles are painted black. Note that a tile of $Tiling(i_\beta)$ of t is actually a tile of $Tiling(2i_\beta)$ of S. Moreover, each tile of $Tiling(i_\beta)$ of S is made of exactly 4^{i_β} tiles of $Tiling(2i_\beta)$ of S. Hence, as depicted in Figure 6.3, the property we obtain at the end of the second step is as follows: each tile of $Tiling(2i_\beta)$ of S is either black or white.



(*a*) At the end of a first step for a hint (P_1, P_2)

(*b*) At the end of a second step

Figure 6.3: White and black tiles at the end of the first and the second step of a phase, for square $S = ABCD$ and $i_\beta = 2$.

In the next steps, the agent applies a similar process at increasingly finer levels of tiling. More

precisely, in step $2 < s \leq k$, the agent moves to the center of each white tile of $Tiling((s-1)i_\beta)$ of S and gets a hint that allows it to paint black at least one tile of $Tiling(s \cdot i_\beta)$ of S. At the end of step s, each tile of $Tiling(s \cdot i_\beta)$ of S is either black or white. We can show that at each step s the agent paints black at least $\frac{1}{4^{i_\beta}}$th of the area of S that is white at the beginning of step s.

After step k, each tile of $Tiling(k \cdot i_\beta)$ of S is either black or white. These steps permit the agent to exclude some area without having to search it directly, while keeping some regularity of the shape of the black area. The agent paints black a smaller area than excluded by the hints but a more regular one. This regularity enables in turn the next process in the area remaining white. Indeed, the agent subsequently executes a brute-force searching that consists in moving to each white tile of $Tiling(k \cdot i_\beta)$ of S in order to scan it using the procedure `RectangleScan`. If, after having scanned all the remaining white tiles, it has not found the treasure, the agent repaints white all the square S and enters the next phase. Thus we have the guarantee that the agent finds the treasure by the end of phase $\lceil \log_2 \Delta \rceil + 1$, i.e., a phase in which the initial supposed square is large enough to contain the treasure. The question is: how much do we have to pay for all of this? In fact, the cost depends on the value that is assigned to k in each phase j. The value of k must be large enough so that the distance traveled by the agent during the brute-force searching is relatively small. At the same time, this value must be small enough so that the the distance traveled during the k steps is not too large. A good trade-off can be reached when $k = \lceil \log_{4^{i_\beta}} \sqrt{2^j} \rceil$. Indeed, as highlighted in the proof of correctness, it is due to this carefully chosen value of k that we can beat the cost $\Theta(\Delta^2)$ necessary without hints, and get a complexity of $O(\Delta^{2-\epsilon})$, where ϵ is a positive real depending on i_β, and hence depending on the angle β.

6.4.2 Algorithm and Analysis

In this subsection we describe our algorithm in detail, prove its correctness and analyze its complexity. We will use the notion of a slicing of a square. Given a straight square S, the $Slicing(i)$ of S, for any integer $i \geq 3$, is the partition of the square S into 2^i triangles with a common vertex at the center q of the square, resulting from partitioning the angle 2π into angles $\frac{2\pi}{2^i}$ using lines containing the point q, one of which is horizontal.

Consider any $Slicing(i)$ of a square S. Let Σ be the set of all side lengths of triangles into which $Slicing(i)$ partitions S. We define ρ_i to be the maximum of all integers $\lceil a/b \rceil$, where $a, b \in \Sigma$. Note that ρ_i depends only on i and not on the side length of S. Moreover, $\rho_{i+1} \geq \rho_i$.

For every integer $i \geq 3$, we define $\phi(i) = i\rho_i$.

In order to define some objects used by our algorithm, we need the following technical proposition.

Proposition 6.4. *The following properties hold.*

1. *For every angle $0 < \alpha < 2\pi$ with vertex at the center of a square S, the angle α contains some triangle of $Slicing(\max(3, \lceil \log_2(\frac{2\pi}{\alpha}) \rceil + 1))$ of S.*

2. *For every integer $i \geq 3$ and for every triangle T of $Slicing(i)$ of a square S, at least one tile of $Tiling(4\phi(i))$ of S is included in the interior of T.*

Proof. We start by proving the first property. Let S be a square and let $0 < \alpha < 2\pi$ be an angle with the vertex in the center of S. Let $i = max(3; \lceil \log_2(\frac{2\pi}{\alpha}) \rceil + 1) \geq 3$. The angle at the center of square S in each of the triangles of $Slicing(i)$ of S is at most $\frac{\alpha}{2}$. Hence one of the triangles formed by $Slicing(i)$ is included in the angle α. This proves the first property.

In the proof of the second property, all tilings and slicings are for square S: for ease of reading we omit mentioning it. In order to prove the second property, we first prove by induction on i the following statement denoted by $\mathcal{H}_i$:

For every integer $i \geq 3$ and for every triangle T of $Slicing(i)$, there is at least one tile t of $Tiling(4\phi(i)-2)$, such that $t \subset T$ and one side of t is included in a side of S.

For the base case $i = 3$, note that each triangle of $Slicing(3)$ contains at least one tile of $Tiling(2)$ with one side included in a side of S. Since $\phi(3) = 3\rho_3$ and $\rho_3 \geq 1$, we know that $4\phi(3) - 2 \geq 10$. Moreover, each side of every tile t' of $Tiling(r)$ contains at least one side of a tile of $Tiling(r')$, included in t', for all integers $r < r'$. Hence $\mathcal{H}_3$ is true.

Assume that $\mathcal{H}_j$ is true for some integer $j \geq 3$ and let us prove that $\mathcal{H}_{j+1}$ is also true. Suppose by contradiction that $\mathcal{H}_{j+1}$ is false. This means that there exists a triangle T_1 of $Slicing(j+1)$ that contains no tile of $Tiling(4\phi(j+1)-2)$ with one side included in a side of S. Denote by L the side of S that contains a side of T_1. There exists a triangle T of $Slicing(j)$ and a triangle T_2 of $Slicing(j+1)$ such that $T_1 \cup T_2 = T$ and $T_1 \cap T_2 = l$, where l is the common segment of boundaries of T_1 and T_2. Note that triangle T_2 also has a side included in L.

By the inductive hypothesis, there exists a tile t' of $Tiling(4\phi(j)-2)$ such that $t' \subset T$ and one side of t' is included in L. For any integers $r < r'$, every tile of $Tiling(r)$ contains exactly $4^{r'-r}$ tiles of $Tiling(r')$ that are organized in $2^{r'-r}$ rows of $2^{r'-r}$ squares. So, tile t' contains exactly $4^{2\phi(j+1)-2\phi(j)}$ rows that are parallel to L and such that each of them is made of $4^{2\phi(j+1)-2\phi(j)}$ tiles of $Tiling(4\phi(j+1)-2)$. Among these rows consider the one that has a common boundary with L and denote it by R. Note that R contains at least $4^{2\rho_{j+1}}$ tiles of $Tiling(4\phi(j+1)-2)$ because $2\phi(j+1)-2\phi(j) = 2(j+1)\rho_{j+1} - 2j\rho_j$ and $\rho_{j+1} \geq \rho_j$. Denote by R' the row of $Tiling(4\phi(j+1)-2)$ that contains R and by R'' the part of R' made of tiles t'' of $Tiling(4\phi(j+1)-2)$, such that $t'' \subset T$. Note that $R \subseteq R''$ and thus R'' contains at least $4^{2\rho_{j+1}}$ tiles of $Tiling(4\phi(j+1)-2)$. Moreover, note that the smaller of the two angles formed by l and L cannot be smaller than $\frac{\pi}{4}$ or larger than $\frac{\pi}{2}$. As a result, l can intersect at most 2 adjacent tiles s_1, s_2 of R''. We will show that l cannot intersect a tile that is at an end of row R''. Let x be the side length of a tile of $Tiling(4\phi(j+1)-2)$. Suppose that l intersects a tile that is at an end of row R''. In view of the fact that R'' contains all tiles of R that are included in T, a side of T_1 or of T_2 included in L (say the side of T_1 without loss of generality), has length at most $3x$, while the side of T_2 included in L has length at least $(4^{2\rho_{j+1}} - 2)x \geq 14\rho_{j+1}x$. However, $\frac{14\rho_{j+1}x}{3x} > \rho_{j+1}$, which contradicts the definition of ρ_{j+1}. Hence l cannot intersect a tile that is at an end of row R''. This implies that one of the two tiles at the ends of R'' belongs to T_1: by construction this tile belongs to $Tiling(4\phi(j+1)-2)$ with one side belonging to L. Hence we get a contradiction. As a result, $\mathcal{H}_{j+1}$ is true, which ends the proof by induction of $\mathcal{H}_i$.

It remains to conclude the proof of the second property of our proposition. In view of property $\mathcal{H}_i$, we know that for every integer $i \geq 3$ and for every triangle T of $Slicing(i)$, at least one tile of $Tiling(4\phi(i)-2)$ is included in T. Moreover, each tile of $Tiling(4\phi(i)-2)$ contains 4 rows, each made of 4 tiles belonging to $Tiling(4\phi(i))$. Hence, the interior of each tile of $Tiling(4\phi(i)-2)$ contains a tile of $Tiling(4\phi(i))$. This proves the second property and concludes the proof of the proposition. □

For any angle $0 < \alpha < 2\pi$, the index of α, denoted $index(\alpha)$, is the integer $4\phi(max(3, \lceil \log_2(\frac{2\pi}{\alpha}) \rceil + 1))$. Proposition 6.4 implies

Proposition 6.5. *For every angle $0 < \alpha < 2\pi$, the following properties hold.*

1. *For every square S and for every hint (P_1, P_2) of size $2\pi - \alpha$ obtained at the center of S, there exists a tile of $Tiling(index(\alpha))$ of S included in $\overline{(P_1, P_2)}$.*
2. *For every angle $\alpha' < \alpha$, we have $index(\alpha) \leq index(\alpha')$.*

Algorithm 6.4 gives a pseudo-code of the main algorithm of this section. It uses the function `Mosaic` described in Algorithm 6.5 that is the key technical tool permitting the agent to reduce its search area. The agent interrupts the execution of Algorithm 6.4 as soon as it gets at distance 1 from the treasure, at which point it can "see" it and thus treasure hunt stops.

Algorithm 6.4 Procedure `TreasureHunt2`

```
1: IndexNew ← 1
2: i ← 1
3: loop
4:     repeat
5:         IndexOld ← IndexNew
6:         IndexNew ← Mosaic(i, IndexOld)
7:     until IndexNew = IndexOld
8:     i ← i + 1
9: end loop
```

In the following, a square is called black if all its points are black. A square is called white if all points of its interior are white. (In a white square, some points of its border may be black).

Algorithm 6.5 Function `Mosaic`(i, k)

```
1: O ← the initial position of the agent
2: S ← the straight square centered at O with sides of length 2^i
3: paint white all points of S
4: IndexMax ← k
5: for j ← 1 to ⌈log_{4^k} √(2^i)⌉ do
6:     for all tiles t of Tiling((j − 1)k) of S do
7:         if t is white then
8:             go to the center of t
9:             let (P1, P2) be the obtained hint
10:            k' ← index of (P1, P2)
11:            k' ← index of (P1, P2)
12:            if k' > IndexMax then
13:                IndexMax ← k'
14:            end if
15:            if IndexMax = k then
16:                for all tiles t' of Tiling(k) of t such that t' ⊂ (P1, P2) do
17:                    paint black all points of t'
18:                end for
19:            end if
20:        end if
21:    end for
22: end for
23: if IndexMax = k then
24:    for all tiles t of Tiling(k(⌈log_{4^k} √(2^i)⌉)) of S do
25:        if t is white then
26:            go to the center of t
27:            execute RectangleScan(t)
28:        end if
29:    end for
30: end if
31: go to O
32: return IndexMax
```

Lemma 6.2. *For any positive integers i and k, consider an agent executing function* `Mosaic`(i, k) *from its initial position O. Let S be the straight square centered at O with side of length 2^i. For every positive integer $j \leq \lceil \log_{4^k} \sqrt{2^i} \rceil$, at the end of the j-th execution of the first loop (lines 5 to 21) in* `Mosaic`(i, k)*, each tile of $Tiling(jk)$ of S is either black or white.*

Proof. Assume by contradiction that there exists a positive integer $j \leq \lceil \log_{4^k} \sqrt{2^i} \rceil$ such that at the end of the j-th execution of the first loop, there exists at least one tile σ of $Tiling(jk)$ of S that is neither black nor white. Without loss of generality, we assume that j is the first integer for which this occurs.

In view of the minimality of j, we know that just before starting the j-th execution of the first loop, each tile of $Tiling((j-1)k)$ is either black or white. Moreover, for every positive integers

$z' \leq z$, every couple of points that belong to the same tile of $Tiling(z)$ of S, also belong to the same tile of the coarser tiling $Tiling(z')$ of S. Hence, just before starting the j-th execution of the first loop, each tile of $Tiling(jk)$ is either black or white.

During the execution of the first loop, the points that become black remain always black thereafter. Since there exists a tile σ of $Tiling(jk)$ of S that becomes neither black nor white during the j-th execution of the first loop, at some point during this execution, the agent does not paint black all points of σ when executing line 17 of Algorithm 6.5. However, each time the agent executes line 17 of Algorithm 6.5 within the j-th execution of the first loop, when a point of a tile t' of $Tiling(k)$ of any tile of $Tiling((j-1)k)$ of S is painted black, then all points inside and on the boundary of tile t' are painted black. By definition, t' is a tile of $Tiling(jk)$ of S. Hence, at the end of the j-th execution of the first loop, each tile of $Tiling(jk)$ of S is either black or white. Hence, we get a contradiction with the existence of σ which proves the lemma. □

Lemma 6.3. *For every positive integers i and k, a call to function* `Mosaic`(i,k) *has cost at most* $2^{i\frac{3+\log_{4^k}(4^k-1)}{2}+2k+8}$.

Proof. The walk made by the agent executing function `Mosaic`(i,k) can be divided into two parts: the first part $\mathcal{P}_1$ is the walk made by executing lines 1 to 22 of Algorithm 6.5, while the second part $\mathcal{P}_2$ is the walk made by executing lines 23 to 32 of Algorithm 6.5. The distance traveled in $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) will be denoted by $|\mathcal{P}_1|$ (resp. $|\mathcal{P}_2|$). We first focus on the distance traveled in part $\mathcal{P}_1$, in which the walk made by the agent is as follows: for each $j \in \{1, \dots \lceil \log_{4^k} \sqrt{2^i} \rceil\}$, starting from the center of S, the agent moves to the center of every white tile of $Tiling((j-1)k)$ of S. By Algorithm 6.5, the side length of S is 2^i, and thus the distance between any two points of S is upper bounded by 2^{i+1}. Moreover, if for every non-negative integer s, we denote by $\mathcal{Q}_s$ the number of tiles in $Tiling(s)$ of S, then we have

$$|\mathcal{P}_1| \leq 2^{i+1} \sum_{j=1}^{\lceil \log_{4^k} \sqrt{2^i} \rceil} \mathcal{Q}_{(j-1)k} \tag{6.5}$$

In view of the definition of a tiling, for all $j \in \{1, \dots \lceil \log_{4^k} \sqrt{2^i} \rceil\}$ we have

$$\mathcal{Q}_{(j-1)k} = \frac{\mathcal{Q}_{(\lceil \log_{4^k} \sqrt{2^i} \rceil - 1)k}}{4^{(\lceil \log_{4^k} \sqrt{2^i} \rceil - 1)k - (j-1)k}} \tag{6.6}$$

$$= \frac{\mathcal{Q}_{(\lceil \log_{4^k} \sqrt{2^i} \rceil - 1)k}}{4^{(\lceil \log_{4^k} \sqrt{2^i} \rceil - j)k}} \tag{6.7}$$

Hence, in view of (6.5) and (6.7), we have

$$|\mathcal{P}_1| \leq 2^{i+1} \sum_{j=1}^{\lceil \log_{4^k} \sqrt{2^i} \rceil} \frac{\mathcal{Q}_{(\lceil \log_{4^k} \sqrt{2^i} \rceil - 1)k}}{4^{(\lceil \log_{4^k} \sqrt{2^i} \rceil - j)k}} \tag{6.8}$$

$$\leq 2^{i+2} \mathcal{Q}_{(\lceil \log_{4^k} \sqrt{2^i} \rceil - 1)k} \tag{6.9}$$

In view of the definition of a tiling, we have

$$\mathcal{Q}_{(\lceil \log_{4^k} \sqrt{2^i} \rceil - 1)k} = 4^{(\lceil \log_{4^k} \sqrt{2^i} \rceil - 1)k} \tag{6.10}$$

$$= (4^k)^{\lceil \log_{4^k} \sqrt{2^i} \rceil - 1} \tag{6.11}$$

$$\leq \sqrt{2^i} \tag{6.12}$$

Hence from (6.9) and (6.12), we obtain

$$|\mathcal{P}_1| \leq 2^{\frac{3i}{2}+2} \tag{6.13}$$

We now consider the distance traveled in part $\mathcal{P}_2$. Here, there are two cases: either $IndexMax \neq k$ when the agent starts executing line 23 of Algorithm 6.5, or $IndexMax = k$ when the agent starts executing line 23 of Algorithm 6.5. In the first case, $\mathcal{P}_2$ corresponds only to the move made when executing line 31 of Algorithm 6.5. However, during the entire execution of Algorithm 6.5, the agent never leaves the straight square S, centered at O, whose sides have length 2^i. Hence in the first case, $|\mathcal{P}_2| \leq 2^{i+1}$.

The second case is trickier to analyze. Indeed, we have to take into account the distance traveled when executing line 31 of Algorithm 6.5 (that is upper bounded by 2^{i+1} in this case as well) but also the distance traveled when executing lines 24 to 29: note that since those lines are executed, we necessarily have the following claim in the second case.

Claim 6.7. *Once variable IndexMax is assigned the value k (cf. line 4 of Algorithm 6.5), variable IndexMax does not change anymore thereafter.*

The above claim is used in the proof of the following one that is crucial to determine the traveled distance $|\mathcal{P}_2|$. As for Claim 6.7, Claim 6.8 holds in the second case that we currently analyze.

Claim 6.8. *At the end of part $\mathcal{P}_1$, the area of the white surface is at most $2^{i\frac{3+\log_{4^k}(4^k-1)}{2}}$.*

Proof of the claim: To prove the claim, we first show by induction on j the following property $\mathcal{K}_j$:
For every integer $j \in \{1, \ldots, \lceil \log_{4^k} \sqrt{2^i} \rceil\}$, at the end of the j-th execution of the first loop of Algorithm 6.5 the area of the part of the square S that is still white is at most $(\frac{4^k-1}{4^k})^j 2^{2i}$.

During the first execution of the first loop of Algorithm 6.5, the agent is located at the center of S. By Claim 6.7, the agent executes line 17 during this first execution, and by Proposition 6.5, there is at least one tile t' of $Tiling(k)$ of S such that all points of t' are black. Since there are 4^k tiles in $Tiling(k)$ of S, it follows that property $\mathcal{K}_j$ is true for $j = 1$. Now suppose that property $\mathcal{K}_s$ holds for a positive integer s. We show that $\mathcal{K}_{s+1}$ is also true. It is enough to show that at the end of the $(s+1)$-th execution of the first loop of Algorithm 6.5 the part of the square S that is still white has area at most $(\frac{4^k-1}{4^k})^{s+1} 2^{2i}$. In view of Claim 6.7 and Algorithm 6.5, during this $(s+1)$-th execution the agent goes to the center of every white tile of $Tiling(sk)$ of S from which it executes line 17 of Algorithm 6.5. Moreover, by Claim 6.7, we know that the value of variable k' is never larger than k. Hence, by Proposition 6.5, it follows that the agent paints black at least $(\frac{1}{4^k})$-th of each white tile of $Tiling(sk)$ of S during this $(s+1)$-th execution. However, at the beginning of the $(s+1)$-th execution of the first loop, we know from the inductive hypothesis and from Lemma 6.2, that the sum of the areas of the white tiles of $Tiling(sk)$ is at most $(\frac{4^k-1}{4^k})^s 2^{2i}$. Moreover, by painting black at least $(\frac{1}{4^k})$-th of each white tile of $Tiling(sk)$ of S, the agent paints black at least $(\frac{1}{4^k})$-th of the remaining surface that is white at the beginning of the $(s+1)$-th execution of the first loop. This implies $\mathcal{K}_{s+1}$, which concludes the proof by induction of $\mathcal{K}_j$.

From property $\mathcal{K}_j$ with $j \in \{1, \ldots, \lceil \log_{4^k} \sqrt{2^i} \rceil\}$, we know that at the end of part $\mathcal{P}_1$, the area of the white surface is at most

$$2^{2i}(\frac{4^k-1}{4^k})^{\lceil \log_{4^k} \sqrt{2^i} \rceil} \leq 2^{2i}(\frac{4^k-1}{4^k})^{\log_{4^k} \sqrt{2^i}} \tag{6.14}$$

However, we have

$$(\frac{4^k-1}{4^k})^{\log_{\frac{4^k-1}{4^k}} \sqrt{2^i}} = \sqrt{2^i} \tag{6.15}$$

which implies

$$(\frac{4^k-1}{4^k})^{\log_{4^k} \sqrt{2^i}} = 2^{\frac{i}{2}\log_{4^k}(\frac{4^k-1}{4^k})}. \tag{6.16}$$

It follows from (6.14) and (6.16) that the area of the white surface at the end of part $\mathcal{P}_1$ is at most

$$2^{2i+\frac{i}{2}\log_{4^k}(\frac{4^k-1}{4^k})} = 2^{i\frac{3+\log_{4^k}(4^k-1)}{2}}, \tag{6.17}$$

which concludes the proof of the claim. ⋆

Now, we are ready to compute $|\mathcal{P}_2|$ in the case where the condition $IndexMax = k$ holds when the agent executes line 23 of Algorithm 6.5. The value of $|\mathcal{P}_2|$ is the sum of the distance traveled when executing line 31 (upper bounded by 2^{i+1}) and of the distance traveled when executing lines 24 to 29. When executing the latter block of lines, for each white tile t of $Tiling(k(\lceil \log_{4^k} \sqrt{2^i} \rceil))$ of S, the agent performs successively the two following actions:

1. The agent moves to the center of t, at a cost of at most 2^{i+1}.
2. Once the center of t is reached, the agent executes procedure `RectangleScan`(t), at a cost of at most $5l \cdot max(2, l)$ (cf. Proposition 6.1) with l equal to the side length of tile t.

Hence, if we denote by w the number of white tiles in $Tiling(k(\lceil \log_{4^k} \sqrt{2^i} \rceil))$ of S, we have

$$\begin{aligned} |\mathcal{P}_2| &\leq 2^{i+1} + w(2^{i+1} + 5l \cdot max(2, l)) && (6.18)\\ &\leq 2^{i+1}(w+1) + 8w \cdot l \cdot max(2, l) && (6.19)\\ &\leq 2^{i+1}(w+1) + 8w \cdot l^2 + 32w && (6.20) \end{aligned}$$

By the definition of tiling we have $w \leq 4^{k(\lceil \log_{4^k} \sqrt{2^i} \rceil)} \leq 2^{2k+\frac{i}{2}}$. Moreover, in view of Claim 6.8 and Lemma 6.2, we know that $w \cdot l^2 \leq 2^{i\frac{3+\log_{4^k}(4^k-1)}{2}}$. Thus, from (6.20) we have the following:

$$\begin{aligned} |\mathcal{P}_2| &\leq 2^{2k+\frac{3i}{2}+1} + 2^{i+1} + 2^{i\frac{3+\log_{4^k}(4^k-1)}{2}+3} + 2^{2k+\frac{i}{2}+5} && (6.21)\\ &\leq 2^{i\frac{3+\log_{4^k}(4^k-1)}{2}+2k+7} \qquad (because\ k \geq 1). && (6.22) \end{aligned}$$

So, whether $IndexMax = k$ or not when the agent starts executing line 23 of Algorithm 6.5, we have $|\mathcal{P}_2| \leq 2^{i\frac{3+\log_{4^k}(4^k-1)}{2}+2k+7}$. Hence, $|\mathcal{P}_1|+|\mathcal{P}_2| \leq 2^{\frac{3i}{2}+2}+2^{i\frac{3+\log_{4^k}(4^k-1)}{2}+2k+7} \leq 2^{i\frac{3+\log_{4^k}(4^k-1)}{2}+2k+8}$, which concludes the proof of the lemma. □

Let ψ be the index of $2\pi - \beta$. The next proposition follows from Proposition 6.5.

Proposition 6.6. *Let (P_1, P_2) be any hint. The index of $\overline{(P_1, P_2)}$ is at most ψ.*

We are now ready to prove the final result of this section.

Theorem 6.2. *Consider an agent A and a treasure located at distance at most Δ from the initial position of A. By executing procedure* `TreasureHunt2`*, agent A finds the treasure after having traveled a distance in $\mathrm{O}(\Delta^{2-\epsilon})$, for some $\epsilon > 0$.*

Proof. We will use the following two claims.

Claim 6.9. *Let $i \geq 1$ be an integer. The number of executions of the repeat loop in the i-th execution of the external loop in Algorithm 6.4 is bounded by ψ.*

Proof of the claim: Suppose by contradiction that the claim does not hold for some $i \geq 1$. So, the number of executions of the repeat loop in the i-th execution of the external loop in Algorithm 6.4 is at least $\psi + 1$. In each of these executions of the repeat loop, the agent calls function `Mosaic`$(i, *)$ exactly once. For all $1 \leq j \leq \psi + 1$ ($\psi \geq 1$, by definition of an index), denote by v_j the returned value of function `Mosaic`$(i, *)$ in the j-th execution of the repeat loop in the i-th execution of the external loop. Note that $v_1 \neq 1$: indeed, if $v_1 = 1$ the repeat loop would be executed exactly once, which would be a contradiction because it is executed at least $\psi + 1 \geq 2$ times.

In view of Algorithm 6.4 and Proposition 6.6, the returned value of `Mosaic`$(i, *)$ is a positive integer that is at most ψ. Since $v_1 \neq 1$, this implies that $\psi \geq 2$. Moreover, for all $2 \leq j \leq \psi$, we have $v_j \geq v_{j-1}$ (refer to lines 5-6 of Algorithm 6.4 and lines 4, 12-13 of Algorithm 6.5). Hence, there exists an integer $k \leq \psi$ such that $v_k = v_{k-1}$. However, according to Algorithm 6.4, this implies that the number of executions of the repeat loop in the i-th execution of the external loop is at most $k \leq \psi$. This is a contradiction which concludes the proof of the claim. ⋆

Claim 6.10. *The distance traveled by the agent before variable i becomes equal to $\lceil \log_2 \Delta \rceil + 2$ in the execution of Algorithm 6.4 belongs to $\mathrm{O}(\Delta^{2-\epsilon})$, where $\epsilon = \frac{1}{2}(1 - \log_{4^\psi}(4^\psi - 1)) > 0$.*

Proof of the claim: In view of the fact that the returned value of every call to function `Mosaic` in the execution of Algorithm 6.4 is at most ψ, it follows that in each call to function `Mosaic`$(*, k)$ the parameter k is always at most ψ. Hence, in view of Claim 6.9 and Lemma 6.3, as long as variable i does not reach the value $\lceil \log_2 \Delta \rceil + 2$, the agent traveled a distance at most

$$\psi \cdot \sum_{i=1}^{\lceil \log_2 \Delta \rceil + 1} 2^{i \frac{3 + \log_{4^\psi}(4^\psi - 1)}{2} + 2\psi + 8} \tag{6.23}$$

$$\leq \psi 2^{(\lceil \log_2 \Delta \rceil + 1) \frac{3 + \log_{4^\psi}(4^\psi - 1)}{2} + 2\psi + 9} \tag{6.24}$$

$$\leq \psi 2^{2\psi + 12 + \log_{4^\psi}(4^\psi - 1)} 2^{(\log_2 \Delta) \frac{3 + \log_{4^\psi}(4^\psi - 1)}{2}} \tag{6.25}$$

$$= \psi 2^{2\psi + 12 + \log_{4^\psi}(4^\psi - 1)} \Delta^{2 - \frac{1}{2}(1 - \log_{4^\psi}(4^\psi - 1))} \tag{6.26}$$

By (6.26), the total distance traveled by the agent executing Algorithm 6.4 belongs to $\mathrm{O}(D^{2-\epsilon})$ where $\epsilon = \frac{1}{2}(1 - \log_{4^\psi}(4^\psi - 1))$. Since ψ is a positive integer, we have $0 < \log_{4^\psi}(4^\psi - 1) < 1$ and hence $\epsilon > 0$. This ends the proof of the claim. ⋆

Assume that the theorem is false. As long as variable i does not reach $\lceil \log_2 \Delta \rceil + 2$, the agent cannot find the treasure, as this would contradict Claim 6.10. Thus, in view of Claim 6.9, before the time τ when variable i reaches $\lceil \log_2 \Delta \rceil + 2$ the treasure is not found. By Algorithm 6.4, this implies that during the last call to function `Mosaic` before time τ, the function returns a value that is equal to its second input parameter. This implies that during this call, the agent has executed lines 24 to 29 of Algorithm 6.5: more precisely, there is some integer x such that from each white tile t of $Tiling(x)$ of the straight square S that is centered at the initial position of the agent and that has sides of length $2^{\lceil \log_2 \Delta \rceil + 1}$, the agent has executed function `RectangleScan`(t). Hence, at the end of the execution of lines 24 to 29, the agent has seen all points of each white tile of $Tiling(x)$ of S. Moreover, in view of Lemma 6.2, we know that the tiles that are not

white, in $Tiling(x)$ of S, are necessarily black. Given a black tile σ of $Tiling(x)$, each point of σ is black, which, in view of lines 16 to 18 of Algorithm 6.5, implies that σ cannot contain the treasure. Since square S necessarily contains the treasure, it follows that the agent must find the treasure by the end of the last execution of function `Mosaic` before time τ. As a consequence, the agent stops the execution of Algorithm 6.4 before assigning $\lceil \log_2 \Delta \rceil + 2$ to variable i and thus, we get a contradiction with the definition of time τ, which proves the theorem. □

6.5 Arbitrary Angles

In this section we observe that if hints can be arbitrary angles smaller than 2π then the treasure hunt cost $\Theta(\Delta^2)$ cannot be improved in the worst case. We prove the following proposition.

Proposition 6.7. *If hints can be arbitrary angles smaller than 2π then the optimal cost of treasure hunt for a treasure at distance at most Δ from the starting point of the agent is $\Omega(\Delta^2)$.*

Proof. Consider the disc $\mathcal{D}$ of radius Δ centered at the initial position of the agent. Consider any position of the agent and suppose that the angle given as hint has size $\gamma > \pi$. Call the complement of the hint the *forbidden angle*. It has size $\alpha = 2\pi - \gamma < \pi$. The forbidden angle has the property that the treasure must be outside of it. The forbidden angle of size α can be chosen in such a way that its intersection with $\mathcal{D}$ has area at most $\frac{\alpha}{2\pi}\pi(2\Delta)^2 = \alpha 2\Delta^2$. More precisely, if the current position p of the agent is at the center c of $\mathcal{D}$ then the forbidden angle can be chosen arbitrarily, otherwise it is chosen so that it does not contain c and its bisector is line (cp).

Suppose that there exists a treasure hunt algorithm at cost at most $\Delta^2/2$. Let the sizes of forbidden angles corresponding to consecutive hints be $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, ... etc., each of size half of the preceding, and such that the forbidden angle is chosen with respect to the above strategy. The total area of the intersection of $\mathcal{D}$ with the forbidden angles is at most $(\sum_{i=1}^{\infty} \frac{1}{2^i})2\Delta^2 = 2\Delta^2$. This leaves out a part of the disc $\mathcal{D}$ whose area is at least $(\pi - 2)\Delta^2$. During the walk of length at most $\Delta^2/2$ of the agent, the set of points of $\mathcal{D}$ from which the agent is at distance at most 1 at some point of the walk has area at most $2\frac{\Delta^2}{2} + \pi = \Delta^2 + \pi$. For $\Delta > 5$ we have $(\pi - 2)\Delta^2 > \Delta^2 + \pi$. Hence there exists a point of $\mathcal{D}$ not included in any of the forbidden angles, from which the agent has never been at distance at most 1. Placing the treasure in this point refutes the correctness of the treasure hunt algorithm. This implies that the trajectory of the agent must have length larger than $\Delta^2/2$, for $\Delta > 5$, hence the optimal cost of treasure hunt belongs to $\Omega(\Delta^2)$. □

6.6 Conclusion

For hints that are angles at most π we gave a treasure hunt algorithm with optimal cost linear in Δ. For larger angles we showed a separation between the case where angles are bounded away from 2π, when we designed an algorithm with cost strictly subquadratic in Δ, and the case where angles have arbitrary values smaller than 2π, when we showed a quadratic lower bound on the cost. The optimal cost of treasure hunt with large angles bounded away from 2π remains open. In particular, the following questions seem intriguing. Is the optimal cost linear in Δ in this case, or is it possible to prove a super-linear lower bound on it? Does the order of magnitude of this optimal cost depend on the bound $\pi < \beta < 2\pi$ on the angles given as hints?

Chapter 7

Conclusion of the Thesis

7.1 Sum up of the Main Parts

The gathering task is studied as a representant of coordination without prior agreement in spite of locality. Two major difficulties are tackled in this thesis: asynchrony and occurrence of Byzantine faults.

Chapters 3 and 4 both address asynchrony for a simpler version of the gathering problem: rendezvous in finite graphs and in the infinite grid respectively.

In the former chapter, a new variant of the model with traversal durations per agent-edge couple is studied as an intermediary between synchrony and asynchrony in which strong rendezvous is possible. More precisely, it is the known intermediary closest to asynchrony in which strong rendezvous is shown to be possible. The main contribution of this chapter is precisely a deterministic strong rendezvous algorithm whose duration is polynomial in the number of nodes n of the graph, the length $\ell_{\min}$ of the shortest label, and $\tau_{\max}$ the maximum over all traversal durations assigned by the adversary.

In Chapter 4, the model variant considered is not an intermediary, but the asynchronous one. In this harshest case, the first deterministic rendezvous algorithm working in the infinite grid at cost polynomial in the initial Manhattan distance D between the agents and $\ell_{\min}$ is presented. This algorithm in particular allows to construct (by reduction to rendezvous in the infinite grid) the first deterministic asynchronous approach algorithm working at cost polynomial in the initial Euclidean distance Δ between the agents and $\ell_{\min}$.

Chapter 5 addresses Byzantine gathering in finite graphs and presents the first deterministic algorithm for this problem whose duration is polynomial in n and $\ell_{\min}$. In this chapter, attention is also paid to the length of the some piece of advice given to all agents and called global knowledge. The length of the global knowledge needed by the algorithm presented belongs to $\mathrm{O}(\log\log\log n)$. It is proved to be of optimal order of magnitude, since no deterministic algorithm for this problem, polynomial n and $\ell_{\min}$ can use a global knowledge belonging to $\mathrm{o}(\log\log\log n)$.

Lastly, Chapter 6 studies a simpler version of the rendezvous: treasure hunt, in the plane. It introduces the scenario in which after each move, the mobile agent obtains an hint consisting of an angle centered at its current position and in which lies the treasure. It is shown that in some cases, these hints allow the agent to find the treasure at a lower than $\mathrm{O}(\Delta^2)$ cost (which is known to be optimal without hints), while in others, the cost still belongs to $\Omega(\Delta^2)$ where Δ denotes the initial distance between the mobile agent and the treasure. More precisely, when each angle is at most π, an deterministic algorithm with cost belonging to $\mathrm{O}(\Delta)$ is provided. Moreover, when there exists $\beta < 2\pi$ such that each angle is at most β, then the hints are helpful too: a deterministic algorithm whose cost belongs to $\mathrm{O}(\Delta^{2-\epsilon})$ is shown. Finally, when angles can be arbitrary close to 2π, it is shown that $\Theta(\Delta^2)$ cannot be beaten.

7.2 Perspectives of the Thesis

The conclusions of Chapters 3 to 6 present questions left open by the contributions of the corresponding chapter. This section broadens the focus.

Study of the Universal eXploration Sequences. The algorithms for finite graphs (introduced in Chapters 3 and 5) rely on the exploration procedure based on Universal eXploration Sequences (`UXS`) [76] and denoted by `Explo` in this thesis. This is not surprising since most algorithms from the literature of gathering in arbitrary finite graphs rely on similar combinatorial tools. For this reason, the cost and duration of these algorithms depend on the length of these tools.

However, the latter is mostly unknown. There exists a polynomial Q such that `UXS` for all graphs with at most n nodes and whose length is $Q(n)$ have been proved to exist [5, 76]. An algorithm for constructing such sequences whose time and space complexities are respectively polynomial and logarithmic in n can be derived from a celebrated result [96]. However, the length of the built `UXS`, although polynomial in n, is believed to be high. Besides the lack of an algorithm building short `UXS`, there is currently no non-trivial lower bound over the length of such tools.

With an algorithm constructing short `UXS`, it would become possible to provide fine grained analysis of the gathering algorithms in finite graphs. Moreover, lower bounds on the complexity of such algorithms could possibly be derived from a lower bound on the length of these sequences.

Towards simpler algorithms with more reliable proofs. Most gathering algorithms and in particular those which tackle asynchrony or faults (just like those presented in this thesis) are complicated. Their descriptions are long and involve several subroutines. This increases the effort needed to understand them as well as the likelihood that their handwritten proofs contain errors.

In synchronous settings, some clarity is gained by designing algorithms in a modular way. For instance, gathering can be reduced to rendezvous thanks to the "stick together" strategy [77], and rendezvous can be reduced to treasure hunt thanks to the "wait for mummy" strategy [7, 98]. Thus, presenting a treasure hunt algorithm for some settings is enough to introduce a gathering algorithm for the same settings, provided that the two reductions are valid in the latter.

When facing asynchrony and Byzantine agents, this is where the problem lies: these reductions are not valid. Hence, some new reductions or improvements of the existing synchronous reductions, for asynchrony and fault tolerance, would be an important step forward in facing these issues.

Designing such reductions is really challenging since they should combine the two following properties. On one hand, they must significantly ease the proposal of new contributions by including powerful techniques. On the other hand, they must be widely reusable and thus apply in several settings, without depriving of the freedom of innovating required to address these issues.

Formal verification can be another, complimentary, approach to improve the reliability of the algorithms involving mobile entities. Recently, some efforts have been dedicated to verify their proofs in particular thanks to the proof assistant called Coq using the framework Pactole [12] or the Maude LTL model checker [54].

Considering dynamic environments. In this thesis, like in most articles regarding distributed systems, the considered environments are static i.e., the ability of the entities to communicate, or move does not evolve with time. However, this is not the case of real life communication networks (e.g., peer to peer or involving mobile devices), or of the road network.

For this reason, some studies model the environment with dynamic graphs [32, 73, 83] which differ from the static ones by the appearance and disappearance of edges (and nodes).

In such structures, some graph theory notions such as path or diameter are reconsidered, and new notions, taking time account, are proposed. For instance, a journey is a temporal path. Since it is generally assumed that conveying information through some communication channel

or walking between two places requires time, the existence of a path between two vertices u and v in the static graph obtained when taking a snapshot of a the dynamic graph i.e., looking at its state at a given time, is not enough to ensure that some piece of information or mobile entity manages to travel from u to v: Some of the edges which compose the path may disappear while the message is transiting, and make v unreachable. On the contrary, a journey is a sequence of neighboring edges in increasing order of presence. After crossing one of the edges of a journey, it is possible (maybe after some waiting period) to cross the next one.

Families of graphs are also reconsidered, and classes of dynamic graphs are presented on the basis of the properties verified in particular by the reappearance of the edges. For instance, one may consider the class of all dynamic graphs whose edges reappear periodically often, the one in which they reappear infinitely often, or the one in which, for every time t, the static graph representing the state of the graph at t is connected.

In such settings, some authors have addressed in particular the task of gathering, but there are still many open questions in the area [29, 84, 102].

Another approach to dynamic environments is appropriate when the topological changes (appearance and disappearance of edges and nodes) occur rarely: when the time interval between two topological changes is long enough for an algorithm to be executed integrally between them. In such a context, the topological changes can be viewed as transient (i.e., rare with the above sense and with finite duration) faults corrupting the memory of the computing entities involved (e.g., lists of neighbors, spanning structures). Such faults are addressed by the paradigm of self-stabilization [9, 55, 87]. Although just after the occurrence of such a fault, little can be ensured, a self-stabilizing algorithm makes the system reach, in finite and bounded time, a configuration in which it verifies the desired properties. By the way, some recent efforts are dedicated to certify the validity of self-stabilizing algorithms thanks to the proof assistant Coq [8].

Bibliography

[1] Serge Abiteboul, Haim Kaplan, and Tova Milo. "Compact Labeling Schemes for Ancestor Queries". In: *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*. Ed. by S. Rao Kosaraju. ACM/SIAM, 2001, pp. 547–556. ISBN: 978-0-89871-490-6. URL: http://dl.acm.org/citation.cfm?id=365411.365529 (visited on 04/11/2019).

[2] Noa Agmon and David Peleg. "Fault-Tolerant Gathering Algorithms for Autonomous Mobile Robots". In: *SIAM J Comput* 36.1 (2006), pp. 56–82. DOI: 10.1137/050645221.

[3] Oswin Aichholzer, Franz Aurenhammer, Christian Icking, Rolf Klein, Elmar Langetepe, and Günter Rote. "Generalized Self-Approaching Curves". In: *Discrete Appl. Math.* 109.1-2 (2001), pp. 3–24. DOI: 10.1016/S0166-218X(00)00233-X.

[4] Manuel Alcantara, Armando Castañeda, David Flores-Peñaloza, and Sergio Rajsbaum. "The Topology of Look-Compute-Move Robot Wait-Free Algorithms with Hard Termination". In: *Distrib. Comput.* 32.3 (2019), pp. 235–255. DOI: 10.1007/s00446-018-0345-3.

[5] Romas Aleliunas, Richard M. Karp, Richard J. Lipton, László Lovász, and Charles Rackoff. "Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems". In: *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*. IEEE Computer Society, 1979, pp. 218–223. DOI: 10.1109/SFCS.1979.34.

[6] Steve Alpern. "The Rendezvous Search Problem". In: *SIAM J Control Optim* 33.3 (May 1995), pp. 673–683. ISSN: 0363-0129. DOI: 10.1137/S0363012993249195.

[7] Steve Alpern. "Rendezvous Search: A Personal Perspective". In: *Oper. Res.* 50.5 (2002), pp. 772–795. DOI: 10.1287/opre.50.5.772.363.

[8] Karine Altisen, Pierre Corbineau, and Stéphane Devismes. "A Framework for Certified Self-Stabilization". In: *Log. Methods Comput. Sci.* 13.4 (2017). DOI: 10.23638/LMCS-13(4:14)2017.

[9] Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019. DOI: 10.2200/S00908ED1V01Y201903DCT015.

[10] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. "Sharing Memory Robustly in Message-Passing Systems". In: *J ACM* 42.1 (1995), pp. 124–142. DOI: 10.1145/200836.200869.

[11] Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. "Searching in the Plane". In: *Inf Comput* 106.2 (1993), pp. 234–252. DOI: 10.1006/inco.1993.1054.

[12] Thibaut Balabonski, Amélie Delga, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. "Synchronous Gathering without Multiplicity Detection: A Certified Algorithm". In: *Theory Comput Syst* 63.2 (2019), pp. 200–218. DOI: 10.1007/s00224-017-9828-z.

[13] Evangelos Bampas, Jurek Czyzowicz, Leszek Gasieniec, David Ilcinkas, and Arnaud Labourel. “Almost Optimal Asynchronous Rendezvous in Infinite Multidimensional Grids”. In: *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*. DISC. Ed. by Nancy A. Lynch and Alexander A. Shvartsman. Vol. 6343. Lecture Notes in Computer Science. Springer, 2010, pp. 297–311. ISBN: 978-3-642-15762-2. DOI: 10.1007/978-3-642-15763-9_28.

[14] Michael Barborak and Miroslaw Malek. “The Consensus Problem in Fault-Tolerant Computing”. In: *ACM Comput Surv* 25.2 (1993), pp. 171–220. DOI: 10.1145/152610.152612.

[15] Lali Barrière, Paola Flocchini, Eduardo Mesa Barrameda, and Nicola Santoro. “Uniform Scattering of Autonomous Mobile Robots in a Grid”. In: *Int J Found Comput Sci* 22.3 (2011), pp. 679–697. DOI: 10.1142/S0129054111008295.

[16] Vic Baston and Shmuel Gal. “Rendezvous Search When Marks Are Left at the Starting Points”. In: *Nav. Res. Logist. NRL* 48.8 (2001), pp. 722–731. ISSN: 1520-6750. DOI: 10.1002/nav.1044.

[17] Anatole Beck and D. J. Newman. “Yet More on the Linear Search Problem”. In: *Israel J. Math.* 8.4 (Dec. 1, 1970), pp. 419–429. ISSN: 1565-8511. DOI: 10.1007/BF02798690.

[18] Anthony Bonato and Richard Nowakowski. *The Game of Cops and Robbers on Graphs*. Vol. 61. The Student Mathematical Library. Providence, Rhode Island: American Mathematical Society, Aug. 16, 2011. ISBN: 978-0-8218-5347-4 978-1-4704-1656-0. DOI: 10.1090/stml/061. URL: http://www.ams.org/stml/061 (visited on 04/10/2019).

[19] Sébastien Bouchard, Marjorie Bournat, Yoann Dieudonné, Swan Dubois, and Franck Petit. “Asynchronous Approach in the Plane: A Deterministic Polynomial Algorithm”. In: *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*. DISC. Ed. by Andréa W. Richa. Vol. 91. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 8:1–8:16. ISBN: 978-3-95977-053-8. DOI: 10.4230/LIPIcs.DISC.2017.8.

[20] Sébastien Bouchard, Marjorie Bournat, Yoann Dieudonné, Swan Dubois, and Franck Petit. “Approche asynchrone dans le plan : un algorithme déterministe polynomial”. In: *ALGOTEL 2018 - 20èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*. ALGOTEL. May 29, 2018. URL: https://hal.archives-ouvertes.fr/hal-01782388/document (visited on 04/24/2019).

[21] Sébastien Bouchard, Marjorie Bournat, Yoann Dieudonné, Swan Dubois, and Franck Petit. “Asynchronous Approach in the Plane: A Deterministic Polynomial Algorithm”. In: *Distrib. Comput.* 32.4 (2019), pp. 317–337. DOI: 10.1007/s00446-018-0338-2.

[22] Sébastien Bouchard, Yoann Dieudonné, and Bertrand Ducourthial. “Byzantine Gathering in Networks”. In: *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015, Post-Proceedings*. SIROCCO. Ed. by Christian Scheideler. Vol. 9439. Lecture Notes in Computer Science. Springer, 2015, pp. 179–193. ISBN: 978-3-319-25257-5. DOI: 10.1007/978-3-319-25258-2_13.

[23] Sébastien Bouchard, Yoann Dieudonné, and Bertrand Ducourthial. “Byzantine Gathering in Networks”. In: *Distrib. Comput.* 29.6 (2016), pp. 435–457. DOI: 10.1007/s00446-016-0276-9.

[24] Sébastien Bouchard, Yoann Dieudonné, and Bertrand Ducourthial. “Rassemblement byzantin dans les réseaux”. In: ALGOTEL. May 29, 2018. URL: https://hal.archives-ouvertes.fr/hal-01782387 (visited on 07/04/2019).

[25] Sébastien Bouchard, Yoann Dieudonné, and Anissa Lamani. “Byzantine Gathering in Polynomial Time”. In: *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. ICALP. Ed. by Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella. Vol. 107. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 147:1–147:15. ISBN: 978-3-95977-076-7. DOI: 10.4230/LIPIcs.ICALP.2018.147.

[26] Sébastien Bouchard, Yoann Dieudonné, Andrzej Pelc, and Franck Petit. “Deterministic Treasure Hunt in the Plane with Angular Hints”. In: *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*. ISAAC. Ed. by Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao. Vol. 123. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 48:1–48:13. ISBN: 978-3-95977-094-1. DOI: 10.4230/LIPIcs.ISAAC.2018.48.

[27] Sébastien Bouchard, Yoann Dieudonné, Andrzej Pelc, and Franck Petit. “On Deterministic Rendezvous at a Node of Agents with Arbitrary Velocities”. In: *Inf Process Lett* 133 (2018), pp. 39–43. DOI: 10.1016/j.ipl.2018.01.003.

[28] Sébastien Bouchard, Yoann Dieudonné, Andrzej Pelc, and Franck Petit. “Trouver un trésor plus rapidement avec des conseils angulaires”. In: ALGOTEL. June 4, 2019. URL: https://hal.inria.fr/hal-02118362 (visited on 06/11/2019).

[29] Marjorie Bournat, Swan Dubois, and Franck Petit. “Gracefully Degrading Gathering in Dynamic Rings”. In: *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*. SSS. Ed. by Taisuke Izumi and Petr Kuznetsov. Vol. 11201. Lecture Notes in Computer Science. Springer, 2018, pp. 349–364. ISBN: 978-3-030-03231-9. DOI: 10.1007/978-3-030-03232-6_23.

[30] Quirijn W. Bouts, Thom Castermans, Arthur van Goethem, Marc J. van Kreveld, and Wouter Meulemans. “Competitive Searching for a Line on a Line Arrangement”. In: *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*. Ed. by Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao. Vol. 123. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 49:1–49:12. ISBN: 978-3-95977-094-1. DOI: 10.4230/LIPIcs.ISAAC.2018.49.

[31] Quentin Bramas and Sébastien Tixeuil. “Brief Announcement: Probabilistic Asynchronous Arbitrary Pattern Formation”. In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*. Ed. by George Giakkoupis. ACM, 2016, pp. 443–445. ISBN: 978-1-4503-3964-3. DOI: 10.1145/2933057.2933074.

[32] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. “Time-Varying Graphs and Dynamic Networks”. In: *IJPEDS* 27.5 (2012), pp. 387–408. DOI: 10.1080/17445760.2012.668546.

[33] Jérémie Chalopin, Yoann Dieudonné, Arnaud Labourel, and Andrzej Pelc. “Rendezvous in Networks in Spite of Delay Faults”. In: *Distrib. Comput.* 29.3 (2016), pp. 187–205. DOI: 10.1007/s00446-015-0259-2.

[34] Jérémie Chalopin, Emmanuel Godard, Yves Métivier, and Rodrigue Ossamy. “Mobile Agent Algorithms Versus Message Passing Algorithms”. In: *Principles of Distributed Systems, 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Proceedings*. Ed. by Alexander A. Shvartsman. Vol. 4305. Lecture Notes in Computer Science. Springer, 2006, pp. 187–201. ISBN: 978-3-540-49990-9. DOI: 10.1007/11945529_14.

[35] Timothy H. Chung, Geoffrey A. Hollinger, and Volkan Isler. "Search and Pursuit-Evasion in Mobile Robotics - A Survey". In: *Auton Robots* 31.4 (2011), pp. 299–316. DOI: 10.1007/s10514-011-9241-4.

[36] Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. "Asynchronous Arbitrary Pattern Formation: The Effects of a Rigorous Approach". In: *Distrib. Comput.* 32.2 (2019), pp. 91–132. DOI: 10.1007/s00446-018-0325-7.

[37] Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. "Distributed Computing by Mobile Robots: Gathering". In: *SIAM J Comput* 41.4 (2012), pp. 829–879. DOI: 10.1137/100796534.

[38] Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. "Label-Guided Graph Exploration by a Finite Automaton". In: *ACM Trans Algorithms* 4.4 (2008), 42:1–42:18. DOI: 10.1145/1383369.1383373.

[39] Reuven Cohen and David Peleg. "Convergence of Autonomous Mobile Robots with Inaccurate Sensors and Movements". In: *SIAM J Comput* 38.1 (2008), pp. 276–302. DOI: 10.1137/060665257.

[40] Andrew Collins, Jurek Czyzowicz, Leszek Gasieniec, and Arnaud Labourel. "Tell Me Where I Am So I Can Meet You Sooner". In: *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*. Ed. by Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis. Vol. 6199. Lecture Notes in Computer Science. Springer, 2010, pp. 502–514. ISBN: 978-3-642-14161-4. DOI: 10.1007/978-3-642-14162-1_42.

[41] Jurek Czyzowicz, Konstantinos Georgiou, Evangelos Kranakis, Danny Krizanc, Lata Narayanan, Jaroslav Opatrny, and Sunil M. Shende. "Search on a Line by Byzantine Robots". In: *27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia*. Ed. by Seok-Hee Hong. Vol. 64. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 27:1–27:12. ISBN: 978-3-95977-026-2. DOI: 10.4230/LIPIcs.ISAAC.2016.27.

[42] Jurek Czyzowicz, Adrian Kosowski, and Andrzej Pelc. "How to Meet When You Forget: Log-Space Rendezvous in Arbitrary Graphs". In: *Distrib. Comput.* 25.2 (2012), pp. 165–178. DOI: 10.1007/s00446-011-0141-9.

[43] Jurek Czyzowicz, Andrzej Pelc, and Arnaud Labourel. "How to Meet Asynchronously (Almost) Everywhere". In: *ACM Trans Algorithms* 8.4 (2012), 37:1–37:14. DOI: 10.1145/2344422.2344427.

[44] Gianlorenzo D'Angelo, Gabriele Di Stefano, and Alfredo Navarra. "Gathering on Rings under the Look-Compute-Move Model". In: *Distrib. Comput.* 27.4 (2014), pp. 255–285. DOI: 10.1007/s00446-014-0212-9.

[45] Shantanu Das, Dariusz Dereniowski, Adrian Kosowski, and Przemyslaw Uznanski. "Rendezvous of Distance-Aware Mobile Agents in Unknown Graphs". In: *Structural Information and Communication Complexity - 21st International Colloquium, SIROCCO 2014, Takayama, Japan, July 23-25, 2014. Proceedings*. Ed. by Magnús M. Halldórsson. Vol. 8576. Lecture Notes in Computer Science. Springer, 2014, pp. 295–310. ISBN: 978-3-319-09619-3. DOI: 10.1007/978-3-319-09620-9_23.

[46] Xavier Défago, Maria Gradinariu, Stéphane Messika, and Philippe Raipin Parvédy. "Fault-Tolerant and Self-Stabilizing Mobile Robots Gathering". In: *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*. Ed. by Shlomi Dolev. Vol. 4167. Lecture Notes in Computer Science. Springer, 2006, pp. 46–60. ISBN: 978-3-540-44624-8. DOI: 10.1007/11864219_4.

[47] Erik D. Demaine, Sándor P. Fekete, and Shmuel Gal. "Online Searching with Turn Cost". In: *Theor Comput Sci* 361.2-3 (2006), pp. 342–355. DOI: 10.1016/j.tcs.2006.05.018.

[48] Anders Dessmark, Pierre Fraigniaud, Dariusz R. Kowalski, and Andrzej Pelc. "Deterministic Rendezvous in Graphs". In: *Algorithmica* 46.1 (2006), pp. 69–96. DOI: 10.1007/s00453-006-0074-2.

[49] Yoann Dieudonné and Andrzej Pelc. "Deterministic Polynomial Approach in the Plane". In: *Distrib. Comput.* 28.2 (2015), pp. 111–129. DOI: 10.1007/s00446-014-0216-5.

[50] Yoann Dieudonné and Andrzej Pelc. "Anonymous Meeting in Networks". In: *Algorithmica* 74.2 (2016), pp. 908–946. DOI: 10.1007/s00453-015-9982-0.

[51] Yoann Dieudonné, Andrzej Pelc, and David Peleg. "Gathering Despite Mischief". In: *ACM Trans Algorithms* 11.1 (2014), 1:1–1:28. DOI: 10.1145/2629656.

[52] Yoann Dieudonné, Andrzej Pelc, and Vincent Villain. "How to Meet Asynchronously at Polynomial Cost". In: *SIAM J Comput* 44.3 (2015), pp. 844–867. DOI: 10.1137/130931990.

[53] Yoann Dieudonné and Franck Petit. "Self-Stabilizing Gathering with Strong Multiplicity Detection". In: *Theor Comput Sci* 428 (2012), pp. 47–57. DOI: 10.1016/j.tcs.2011.12.010.

[54] Ha Thi Thu Doan, François Bonnet, and Kazuhiro Ogata. "Model Checking of Robot Gathering". In: *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*. OPODIS. Ed. by James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão. Vol. 95. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 12:1–12:16. ISBN: 978-3-95977-061-3. DOI: 10.4230/LIPIcs.OPODIS.2017.12.

[55] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. ISBN: 978-0-262-04178-2.

[56] Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. "How Many Ants Does It Take to Find the Food?" In: *Theor Comput Sci* 608 (2015), pp. 255–267. DOI: 10.1016/j.tcs.2015.05.054.

[57] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Vardi. *Reasoning About Knowledge*. MIT Press, Jan. 9, 2004. 533 pp. ISBN: 978-0-262-30782-6.

[58] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. DOI: 10.2200/S00440ED1V01Y201208DCT010.

[59] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, eds. *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*. Vol. 11340. Lecture Notes in Computer Science. Springer, 2019. ISBN: 978-3-030-11071-0. DOI: 10.1007/978-3-030-11072-7.

[60] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. "Gathering of Asynchronous Robots with Limited Visibility". In: *Theor Comput Sci* 337.1-3 (2005), pp. 147–168. DOI: 10.1016/j.tcs.2005.01.001.

[61] Pierre Fraigniaud, Cyril Gavoille, David Ilcinkas, and Andrzej Pelc. "Distributed Computing with Advice: Information Sensitivity of Graph Coloring". In: *Distrib. Comput.* 21.6 (2009), pp. 395–403. DOI: 10.1007/s00446-008-0076-y.

[62] Pierre Fraigniaud, David Ilcinkas, and Andrzej Pelc. "Tree Exploration with Advice". In: *Inf Comput* 206.11 (2008), pp. 1276–1287. DOI: 10.1016/j.ic.2008.07.005.

[63] Pierre Fraigniaud and Andrzej Pelc. “Deterministic Rendezvous in Trees with Little Memory”. In: *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*. Ed. by Gadi Taubenfeld. Vol. 5218. Lecture Notes in Computer Science. Springer, 2008, pp. 242–256. ISBN: 978-3-540-87778-3. DOI: 10.1007/978-3-540-87779-0_17.

[64] Pierre Fraigniaud and Andrzej Pelc. “Delays Induce an Exponential Memory Gap for Rendezvous in Trees”. In: *ACM Trans Algorithms* 9.2 (2013), 17:1–17:24. DOI: 10.1145/2438645.2438649.

[65] G. Matthew Fricke, Joshua P. Hecker, Antonio D. Griego, Linh T. Tran, and Melanie E. Moses. “A Distributed Deterministic Spiral Search Algorithm for Swarms”. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016*. IEEE, 2016, pp. 4430–4436. ISBN: 978-1-5090-3762-9. DOI: 10.1109/IROS.2016.7759652.

[66] B. Grünbaum. “Partitions of Mass-Distributions and of Convex Bodies by Hyperplanes.” In: *Pacific J. Math.* 10.4 (1960), pp. 1257–1261. ISSN: 0030-8730. URL: https://projecteuclid.org/euclid.pjm/1103038065 (visited on 04/10/2019).

[67] Samuel Guilbault and Andrzej Pelc. “Gathering Asynchronous Oblivious Agents with Local Vision in Regular Bipartite Graphs”. In: *Theor Comput Sci* 509 (2013), pp. 86–96. DOI: 10.1016/j.tcs.2012.07.004.

[68] Joseph Y. Halpern. “Computer Science and Game Theory: A Brief Survey”. In: *CoRR* abs/cs/0703148 (2007). URL: http://arxiv.org/abs/cs/0703148 (visited on 05/31/2019).

[69] Taisuke Izumi, Samia Souissi, Yoshiaki Katayama, Nobuhiro Inuzuka, Xavier Défago, Koichi Wada, and Masafumi Yamashita. “The Gathering Problem for Two Oblivious Robots with Unreliable Compasses”. In: *SIAM J Comput* 41.1 (2012), pp. 26–46. DOI: 10.1137/100797916.

[70] Artur Jez and Jakub Lopuszanski. “On the Two-Dimensional Cow Search Problem”. In: *Inf Process Lett* 109.11 (2009), pp. 543–547. DOI: 10.1016/j.ipl.2009.01.020.

[71] Ming-Yang Kao, John H. Reif, and Stephen R. Tate. “Searching in an Unknown Environment: An Optimal Randomized Algorithm for the Cow-Path Problem”. In: *Inf Comput* 131.1 (1996), pp. 63–79. DOI: 10.1006/inco.1996.0092.

[72] Michal Katz, Nir A. Katz, Amos Korman, and David Peleg. “Labeling Schemes for Flow and Connectivity”. In: *SIAM J Comput* 34.1 (2004), pp. 23–40. DOI: 10.1137/S0097539703433912.

[73] David Kempe, Jon M. Kleinberg, and Amit Kumar. “Connectivity and Inference Problems for Temporal Networks”. In: *J Comput Syst Sci* 64.4 (2002), pp. 820–842. DOI: 10.1006/jcss.2002.1829.

[74] Adrian Kosowski and Alfredo Navarra. “Graph Decomposition for Memoryless Periodic Exploration”. In: *Algorithmica* 63.1-2 (2012), pp. 26–38. DOI: 10.1007/s00453-011-9518-1.

[75] David Kotz and Robert S. Gray. “Mobile Agents and the Future of the Internet”. In: *Oper. Syst. Rev.* 33.3 (1999), pp. 7–13. DOI: 10.1145/311124.311130.

[76] Michal Koucký. “Universal Traversal Sequences with Backtracking”. In: *J Comput Syst Sci* 65.4 (2002), pp. 717–726. DOI: 10.1016/S0022-0000(02)00023-5.

[77] Dariusz R. Kowalski and Adam Malinowski. “How to Meet in Anonymous Network”. In: *Theoretical Computer Science*. Structural Information and Communication Complexity (SIROCCO 2006) 399.1 (June 3, 2008), pp. 141–156. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.02.010.

[78] Evangelos Kranakis, Danny Krizanc, Euripides Markou, Aris Pagourtzis, and Felipe Ramírez. "Different Speeds Suffice for Rendezvous of Two Agents on Arbitrary Graphs". In: *SOFSEM 2017: Theory and Practice of Computer Science - 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Limerick, Ireland, January 16-20, 2017, Proceedings*. Ed. by Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria. Vol. 10139. Lecture Notes in Computer Science. Springer, 2017, pp. 79–90. ISBN: 978-3-319-51962-3. DOI: 10.1007/978-3-319-51963-0_7.

[79] Evangelos Kranakis, Nicola Santoro, Cindy Sawchuk, and Danny Krizanc. "Mobile Agent Rendezvous in a Ring". In: *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*. IEEE Computer Society, 2003, pp. 592–599. ISBN: 978-0-7695-1920-3. DOI: 10.1109/ICDCS.2003.1203510.

[80] Elmar Langetepe. "On the Optimality of Spiral Search". In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*. Society for Industrial and Applied Mathematics, Jan. 17, 2010, pp. 1–12. ISBN: 978-0-89871-701-3 978-1-61197-307-5. DOI: 10.1137/1.9781611973075.1.

[81] Elmar Langetepe. "Searching for an Axis-Parallel Shoreline". In: *Theor Comput Sci* 447 (2012), pp. 85–99. DOI: 10.1016/j.tcs.2011.12.069.

[82] Tobias Langner, Barbara Keller, Jara Uitto, and Roger Wattenhofer. "Overcoming Obstacles with Ants". In: *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Ed. by Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru. Vol. 46. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 9:1–9:17. ISBN: 978-3-939897-98-9. DOI: 10.4230/LIPIcs.OPODIS.2015.9.

[83] Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. "Stream Graphs and Link Streams for the Modeling of Interactions over Time". In: *Soc. Netw Anal. Min.* 8.1 (2018), 61:1–61:29. DOI: 10.1007/s13278-018-0537-7.

[84] Giuseppe Antonio Di Luna, Paola Flocchini, Linda Pagli, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. "Gathering in Dynamic Rings". In: *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*. SIROCCO. Ed. by Shantanu Das and Sébastien Tixeuil. Vol. 10641. Lecture Notes in Computer Science. Springer, 2017, pp. 339–355. ISBN: 978-3-319-72049-4. DOI: 10.1007/978-3-319-72050-0_20.

[85] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN: 978-1-55860-348-6.

[86] Gianluca De Marco, Luisa Gargano, Evangelos Kranakis, Danny Krizanc, Andrzej Pelc, and Ugo Vaccaro. "Asynchronous Deterministic Rendezvous in Graphs". In: *Theor Comput Sci* 355.3 (2006), pp. 315–326. DOI: 10.1016/j.tcs.2005.12.016.

[87] Toshimitsu Masuzawa and Hirotsugu Kakugawa. "Self-Stabilization in Spite of Frequent Changes of Networks: Case Study of Mutual Exclusion on Dynamic Rings". In: *Self-Stabilizing Systems, 7th International Symposium, SSS 2005, Barcelona, Spain, October 26-27, 2005, Proceedings*. Ed. by Ted Herman and Sébastien Tixeuil. Vol. 3764. Lecture Notes in Computer Science. Springer, 2005, pp. 183–197. ISBN: 978-3-540-29814-4. DOI: 10.1007/11577327_13.

[88] Avery Miller and Andrzej Pelc. "Fast Rendezvous with Advice". In: *Theor Comput Sci* 608 (2015), pp. 190–198. DOI: 10.1016/j.tcs.2015.09.025.

[89] Avery Miller and Andrzej Pelc. "Time versus Cost Tradeoffs for Deterministic Rendezvous in Networks". In: *Distrib. Comput.* 29.1 (2016), pp. 51–64. DOI: 10.1007/s00446-015-0253-8.

[90] Nicolas Nisse and David Soguet. "Graph Searching with Advice". In: *Theor Comput Sci* 410.14 (2009), pp. 1307–1318. DOI: 10.1016/j.tcs.2008.08.020.

[91] Linda Pagli, Giuseppe Prencipe, and Giovanni Viglietta. "Getting Close without Touching: Near-Gathering for Autonomous Mobile Robots". In: *Distrib. Comput.* 28.5 (2015), pp. 333–349. DOI: 10.1007/s00446-015-0248-5.

[92] Petrisor Panaite and Andrzej Pelc. "Exploring Unknown Undirected Graphs". In: *J Algorithms* 33.2 (1999), pp. 281–295. DOI: 10.1006/jagm.1999.1043.

[93] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. "Reaching Agreement in the Presence of Faults". In: *J ACM* 27.2 (1980), pp. 228–234. DOI: 10.1145/322186.322188.

[94] Andrzej Pelc. "Deterministic Rendezvous in Networks: A Comprehensive Survey". In: *Networks* 59.3 (2012), pp. 331–347. DOI: 10.1002/net.21453.

[95] Eric Rasmusen. *Games and Information: An Introduction to Game Theory*. Wiley, Nov. 28, 2006. 559 pp. ISBN: 978-1-4051-3666-2.

[96] Omer Reingold. "Undirected Connectivity in Log-Space". In: *J ACM* 55.4 (2008), 17:1–17:24. DOI: 10.1145/1391289.1391291.

[97] Thomas C. Schelling. *The Strategy of Conflict*. Harvard University Press, 1980. 332 pp. ISBN: 978-0-674-84031-7.

[98] Amnon Ta-Shma and Uri Zwick. "Deterministic Rendezvous, Treasure Hunts, and Strongly Universal Exploration Sequences". In: *ACM Trans Algorithms* 10.3 (2014), 12:1–12:15. DOI: 10.1145/2601068.

[99] Kevin Spieser and Emilio Frazzoli. "The Cow-Path Game: A Competitive Vehicle Routing Problem". In: *Proceedings of the 51th IEEE Conference on Decision and Control, CDC 2012, December 10-13, 2012, Maui, HI, USA*. IEEE, 2012, pp. 6513–6520. ISBN: 978-1-4673-2065-8. DOI: 10.1109/CDC.2012.6426279.

[100] Ichiro Suzuki and Masafumi Yamashita. "Distributed Anonymous Mobile Robots: Formation of Geometric Patterns". In: *SIAM J Comput* 28.4 (1999), pp. 1347–1363. DOI: 10.1137/S009753979628292X.

[101] Mikkel Thorup and Uri Zwick. "Approximate Distance Oracles". In: *J ACM* 52.1 (2005), pp. 1–24. DOI: 10.1145/1044731.1044732.

[102] Yukiko Yamauchi, Tomoko Izumi, and Sayaka Kamei. "Mobile Agent Rendezvous on a Probabilistic Edge Evolving Ring". In: *Third International Conference on Networking and Computing, ICNC 2012, Okinawa, Japan, December 5-7, 2012*. ICNC. IEEE Computer Society, 2012, pp. 103–112. ISBN: 978-1-4673-4624-5. DOI: 10.1109/ICNC.2012.23.

[103] Yukiko Yamauchi, Taichi Uehara, Shuji Kijima, and Masafumi Yamashita. "Plane Formation by Synchronous Mobile Robots in the Three-Dimensional Euclidean Space". In: *J ACM* 64.3 (2017), 16:1–16:43. DOI: 10.1145/3060272.