EFFICIENT GRAPH EXPLORATION

Jan Hackfeld

Acknowledgements

TODO

Jan Hackfeld

Contents

Chapter 1

Introduction

> The first of these factors is the compelling urge of man to explore and to discover, the thrust of curiosity that leads men to try to go where no one has gone before. Most of the surface of the earth has now been explored and men now turn on the exploration of outer space as their next objective.
>
> "Introduction to Outer Space", President's Science Advisory Committee, 1958.

The exploration of an unknown environment is a central challenge in many applications ranging from searching the internet or a large set of linked data [Pen+12; Mir+13] to physical exploration of unknown terrain [BMS02; Plo+17] or even the universe [Mau03]. In this work, we consider an abstraction of this exploration problem and model the unknown environment as a graph. In many settings the environment is discrete (e.g., the webgraph describing links between pages of the World Wide Web) or it can be discretized (e.g., road networks) without losing the essence of the problem. Another perspective is to view exploration as an abstraction of a process of computing, where every node of the graph corresponds to a configuration (e.g., configuration of a Turing machine or a different model of computation), edges correspond to possible transitions between configurations, and the questions is what configurations are reachable starting in a given initial configuration. In this context, graph exploration has a close connection to complexity theory and the study of the relationship between probabilistic and deterministic space-bounded algorithms [Sav73; CR80; Rei08].

The study of exploration in the context of theoretical computer science originates from investigating how to systematically search a labyrinth for an exit (imagine a garden maze with hedges). One of the first fundamental results in this direction was discovered here in Berlin by Budach, who showed that no finite automaton can find a way out of every finite labyrinth from any initial position [Bud75; Bud78]. Around the same time Shah showed that by utilizing 5 pebbles, that is, some additional markers than can be placed at an arbitrary position in the labyrinth and collected later, a finite automaton can search and find out of any finite labyrinth [Sha74]. This result was subsequently improved by Blum and Kozen who showed that already 2 pebbles are sufficient [BK78] and by Hoffmann who finally showed that this is best possible, i.e., 1 pebble does not suffice to search

and find out of any finite labyrinth [Hof81].

In the following decades the exploration of graphs, as a more abstract and general setting with less structure, was the focus of most research. In these settings typically one or more so-called mobile agents or robots have to deterministically visit all vertices of the given unknown graph. A large variety of different exploration problems have been considered mainly differing in the class of graphs to be explored, the ability of the agent(s) and the objective function. While single agent exploration has been studied for a lot of time and by now is quite well understood, exploration involving multiple agents only has been considered rather recently. The communication between and coordination of multiple robots adds another level of complexity to the exploration problem yielding many interesting open problems in this field of research.

The main focus of this dissertation is to investigate the collaboration of agents in graph-like environments. We study the memory requirement and energy efficiency of collaborating agents exploring a graph and the closely related problem of energy efficient delivery by collaborating agents. The three topics covered in this dissertation are:

Space Efficient Exploration. We study the problem of deterministically exploring an undirected and initially unknown graph with n vertices either by a single agent equipped with a set of pebbles or by a set of collaborating agents. Our goal is to understand how the memory requirement decreases compared to the case of single agent exploration as the agent may mark vertices by dropping and retrieving distinguishable pebbles, or when multiple agents jointly explore the graph. This problem can be seen as a continuation of the starting point in graph exploration, where the central question was how many pebbles does one agent need to explore any finite labyrinth.

Energy Efficient Exploration. We assume that an agent consumes energy proportional to the number of edges it traverses and we investigate the energy efficient exploration of unknown trees by multiple collaborating agents with a fixed energy budget. The objective is to maximize the number of distinct vertices collectively visited by the given agents compared to an algorithm that has complete knowledge of the tree in advance.

Energy Efficient Delivery. We consider the problem of different mobile agents that have to deliver a set of messages in a weighted undirected graph while minimizing the total energy consumption. In our model, the agents consume energy proportional to the distance they travel and different agents can have different rates of energy consumption. The messages have different starting vertices and destinations and different messages can be transported together if the capacity of the agents permits it.

1.1 Contributions and Outline

In this section, we give an outline of the thesis together with a summary of the main results.

Chapter 1: Introduction. In the remainder of this chapter, we introduce the notation and most important concepts used in this thesis. This includes a thorough introduction to the definitions and main concepts common in graph exploration that are necessary to understand the related work and this thesis. Moreover, we give a brief introduction to complexity theory as well as offline and online optimization problems. We further present a detailed overview of previous research in graph exploration and also cover the message delivery literature.

Chapter 2: Space Efficient Graph Exploration. We prove that for a single agent with constant memory $\Theta(\log\log n)$ pebbles are both necessary and sufficient for exploring any undirected graph with n vertices. We further show that collaborating agents are not more powerful than pebbles in this setting as $\Theta(\log\log n)$ agents with constant memory each are necessary and sufficient for the same task. Our results show that the memory requirement can be significantly reduced by utilizing additional pebbles or agents compared to the $\Theta(\log n)$ bits of memory that are necessary and sufficient to explore an undirected graph by a single agent without pebbles [Fra+05; Rei08].

For the upper bounds, we present an algorithm for a single agent with constant memory that explores any n-vertex graph using $O(\log\log n)$ pebbles. The algorithm does not require the number of vertices n as input, terminates after a polynomial number of edge traversals and returns to the starting vertex. We further show that an additional agent is at least as powerful as a pebble and therefore $O(\log\log n)$ agents with constant memory each can also explore any n-vertex graph.

To prove the lower bounds, we construct a family of graphs with $O(s^{2^{5k}})$ vertices that trap any set of k collaborating agents with s states each. Our construction exhibits dramatically smaller traps with only a doubly exponential number of vertices compared to the traps of size $\tilde{O}(s \uparrow\uparrow (2k+1))$ and $\tilde{O}(s \uparrow\uparrow (k+1))$ due to Rollik [Rol80] and Fraigniaud et al. [Fra+06b], respectively. As a consequence of our bound on the size of the trap, we are able to show that, even if we allow $O((\log n)^{1-\epsilon})$ bits of memory for some constant $\epsilon > 0$ for every agent, the number of agents needed for the exploration task is at least $\Omega(\log\log n)$. This construction also yields the lower bound for a single agent with pebbles, as $p+1$ agents with $O((\log n)^{1-\epsilon})$ bits of memory each are more powerful than one agent with $O((\log n)^{1-\epsilon})$ bits of memory and p pebbles.

Our results allow to fully characterize the tradeoff between the number of agents and the memory of each agent. When agents have $\Omega(\log n)$ memory, a single agent without pebbles explores all n-vertex graphs. For agents with $O((\log n)^{1-\epsilon})$ memory, $\Omega(\log\log n)$ agents are needed and it is possible to reduce the memory of every agent to a constant in this case. The tradeoff is similar for pebbles. For an agent with $\Omega(\log n)$ bits of memory, no pebbles are required for the exploration task, whereas for an agent with $O((\log n)^{1-\epsilon})$ bits of memory, already $\Omega(\log\log n)$ pebbles are required. Then again with $\Omega(\log\log n)$ pebbles already a constant number of bits of memory are sufficient for exploration.

Chapter 3: Energy Efficient Tree Exploration. We consider the problem of exploring an unknown tree by k agents initially located at the root of the tree. Every agent has only limited energy and hence can traverse at most B edges. At the beginning, the agents have no knowledge about the

structure of the tree, but they gradually learn its topology as they traverse new edges. We assume that the agents can communicate with each other at arbitrary distances and thus the knowledge obtained by one agent after traversing an edge is instantaneously available to the agents. Our goal is to maximize the number of distinct vertices collectively visited by the agents. We design an online algorithm that carefully balances between sending agents in a depth-first manner to avoid visiting the same set of vertices too often and exploring the tree in a breadth-first manner to make sure that there is no large set of vertices close to the root that was missed by the online algorithm. We show that our algorithm is 3-competitive compared to an optimal solution that we could obtain if we knew the map of the tree in advance. We also show that our analysis is tight by giving a sequence of instances showing that the algorithm is not better than 3-competitive. While it is easy to see that no algorithm can be better than 2-competitive, we give a non-trivial lower bound of 2.17 on the competitive ratio of any online algorithm.

Chapter 4: Energy Efficient Delivery. We study the problem of delivering a set of messages, which are specified as source-target pairs in an undirected weighted graph, by k mobile agents starting at distinct vertices of the graph. Every agent conumes energy proportional to the distance it travels in the graph and the rate of energy consumption may be different for different agents. The goal is to deliver all messages by the agents while minimizing the total energy consumption for this task. The purpose of this chapter is to investigate how the agents benefit from collaborating on delivering the messages compared to the case that every message is only transported by a single agent. We show how an optimal solution to the delivery problem can be 2-approximated by a solution, where messages are only transported by a single agent. We further prove that this is best possible for arbitary number of messages and agent capacity, i.e., number of messages that can be transported at the same time, becomes arbitrarily large. Moreover, for a single message, we present an algorithm that determines an agent which can deliver the message with at most $1/\ln 2 \approx 1.44$-times the cost of an optimal solution improving the general bound of 2. We also show that this is best possible for a single message.

1.2 Preliminaries

In this section, we give an introduction of the terminology and notation of this work. We assume that the reader is familiar with the basic concepts in graph theory, complexity theory and algorithms and therefore only briefly recall the respective definitions in order to introduce a consistent notation. A general introduction to these topics can be found in the textbooks by Korte and Vygen [KV18] or Cormen et al. [CLR89], for instance. We also introduce the basic concepts and definitions used in the context of graph exploration which are necessary to understand the related work and this thesis. Additional more specific definitions can be found in the respective chapters.

1.2.1 Graphs

An **graph** is a tuple $G = (V, E)$, where V is a finite non-empty set and $E \subseteq \{(v, w) \mid v, w \in V, v \neq w\}$ if G is **directed** and $E \subseteq \binom{V}{2}$ if G is **undirected**. In both cases, we call the elements in V **vertices** or **nodes** and the elements of E **edges**. We let $n := |V|$ denote the **order** or number of vertices of G and $m := |E|$ the number of edges. If we additionally have a function $w\colon E \to \mathbb{R}$ assigning a weight or length to every edge, then we call G a **weighted graph**. All graphs considered in this work are **simple**, that is, for any vertex $v \in V$ there is at most one edge $\{v, w\}$ or (v, w) in E for every vertex $w \neq v$ and there are no loops, i.e., $\{v, v\} \notin E$ or $(v, v) \notin E$ for all $v \in V$. For an edge $e = \{v, w\}$ or $e = (v, w)$, we call v and w **endpoints** of e and say that v and w are **incident** with e. The **degree** d_v of a vertex v is the number of edges incident to v. If $d_v = d$ for all $v \in V$, then we call the graph ***d*-regular** or simply **regular**. A graph $G' = (V', E')$ is called a **subgraph** of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. If E' contains all edges in E that have both endpoints in V', then G' is called an **induced subgraph** or the **subgraph induced by V'**.

A **walk** in G is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that $\{v_i, v_{i+1}\} \in E$ or $(v_i, v_{i+1}) \in E$ for all $i \in \{0, \ldots, k-1\}$. As we only consider simple graphs, the sequence of vertices of a walk uniquely determine the edges between the vertices. We call v_0 the **starting vertex** or **first vertex** and v_k the **end vertex** or **last vertex** of the walk. A walk is **closed**, if $v_0 = v_k$. A closed walk is also called a **tour**. If additionally all edges along the closed walk are distinct, then the walk is called a **cycle**. A walk where all vertices $v_0, v_1, \ldots, v_k$ are distinct is called a **path**. The **length** of a path is the number of its edges. A **Eulerian walk** or **Eulerian tour** is a closed walk containing every edge of the graph. A graph containing a Eulerian tour is called **Eulerian**.

An undirected graph G is **connected** if for any two distinct vertices v and w, there is a path from v to w in G. An undirected connected graph without any cycles is called a **tree**. The minimum length of a path connecting two distinct vertices v and w in G is called the **distance** between v and w. The maximum distance over all vertices v and w in G is the **diameter** of G.

1.2.2 Exploration

Formally we model an **agent** exploring a graph as a finite automaton $A = (\Sigma, \bar{\Sigma}, \delta, \sigma^*)$, where Σ is a set of **states**, $\bar{\Sigma} \subseteq \Sigma$ is a set of **halting** or **final states**, $\sigma^* \in \Sigma$ is the **starting state** of the agent, and δ is its **transition function**. The transition function describes how the agent interacts with the graph and possible other agents. Its exact specifics depend on the problem considered, i.e., whether we consider a single agent or a group of agents and whether we allow the agents to use additional markers. In every exploration step an agent A observes the local environment at the current vertex and possible additional information, such as the states of other agents or position of markers, and then performs actions, e.g., traverses an edge, according to the transition function δ. In Section 2.1 we give a formal introduction to some agent models including a full description of the transition function δ. In most settings, however, the agent capabilities are described on an informal and intuitive level as the exact implementation is not important for the analysis.

If an agent can distinguish different vertices and in particular the transition function can depend on the specific vertex of the given graph G, then we call G **vertex-labeled** or simply **labeled**. Formally, this means that there is a bijection $\lambda\colon V \to \{1, \dots, n\}$ and the transition function δ can depend on the label $\lambda(v)$ of the current vertex v of the agent. In many graph exploration models, the agent cannot identify or distinguish different vertices and thus the transition can depend on the degree of the current vertex, but not on its label. In this case, we call the graph **unlabeled** or **anonymous**. In order to enable sensible navigation for an agent in this setting, we assume that the edges inciding to a vertex v have distinct labels $0, \dots, d_v - 1$ at v. Hence, every edge $\{v, w\} \in E$ has two labels called **port numbers**, one at v and one at w. These port numbers can be different at both entpoints and we assume no correlation between two port numbers of an edge. We call a graph with such a labeling a **locally edge-labeled** graph.

A single agent then traverses an anonymous, locally edge-labeled graph G as follows: Starting in a vertex v_0, in every step it observes the degree of the current vertex as well as the local port number of the edge leading back to the previous vertex. Depending on its current state, the vertex degree and port number to the previous vertex, it then transitions to a state given by the transition function δ and traverses the edge corresponding to the port number given by the transition function δ.

A different way to specify the behavior of an agent in a regular graph are **traversal sequences**. A traversal sequence is a sequence of integers $l_0, l_1, l_2, \dots$ with $l_i \in \{0, 1, \dots, \Delta - 1\}$ determining the walk of an agent A in a Δ-regular locally edge-labeled graph G. The agent **follows** a traversal sequence $l_0, l_1, \dots$ if it traverses the edges with port number $l_0, l_1, \dots$ in this order. We further say that a traversal sequence is **universal** for a class of undirected, connected, locally edge-labeled Δ-regular graphs $\mathcal{G}$ if an agent following it explores every graph $G \in \mathcal{G}$ for any starting vertex in G, i.e., for any starting vertex it visits all vertices of G. For a set M, we further use the notation $M^* := \bigcup_{i=1}^{\infty} M^i$ to denote the set of finite sequences with elements in M. This allows use to use to compact notation $\omega \in \{0, 1, \dots, \Delta - 1\}^*$ for a finite traversal sequence ω.

Note that traversal sequences are only defined for regular graphs and the port numbers to the previous edge are not taken into account. In order to overcome these shortcomings, Koucký introduced the concept of **exploration sequences** [Kou02]. An exploration sequence is a sequence of integers $e_0, e_1, e_2, \dots$ with $e_i \in \mathbb{Z}$ that guides the walk of an agent through a graph G as follows: Assume an agent starts in a vertex v_0 of an arbitrary locally edge-labeled graph G and let $l_0 = 0$. Let v_i denote the agent's location in step i and l_i the port number of the edge at v_i leading back to the previous location. Then, the agent **follows** the exploration sequence $e_0, e_1, e_2, \dots$ if, in each step i, it traverses the edge with port number $(l_i + e_i) \bmod d_{v_i}$ at v_i to the next vertex v_{i+1}. This means that an exploration sequence gives edge label offset instead of absolute edge label. Thus, exploration sequences are well-defined for arbitrary graphs and also allow backtracking, i.e., returning to the previous vertex, by specifying the offset 0. Analogously, we say that an exploration sequence is **universal** for a class of undirected, connected, locally edge-labeled graphs $\mathcal{G}$ if an agent following it explores every graph $G \in \mathcal{G}$ for any starting vertex in G.

In order to give an agent in an anonymous graph the power to distinguish a limited number of

vertices, it is possible to equip the agent with one or multiple **pebbles**. A pebble is a tool to mark vertices. It can be dropped at a vertex and picked up again later. Every time an agent visits a vertex where it has dropped a pebble, it will observe this marker. Pebbles can be **distinguishable**, i.e., every pebble has some unique identifier, or **indistinguishable**, i.e., the agent only observes the number of pebbles at the current vertex.

Multiple agents can exchange information when exploring a graph. This exchange of information can only be possible locally, i.e., if the agents share a vertex or are only a small distance appart, or globally, i.e., independent of the agents location in the graph. We can model the case of **local communication** by allowing δ to depend on the state of the agents colocated at the same vertex and for the case of **global communication** to allow δ to depend on the state of all other agents. Another way to allow agents to communicate is by means of so-called **whiteboards**. These are local storages at every vertex that the agents can write to and read information from. The amount of local storage available at a node is typically limited. Whiteboards, similar to pebbles, can also be used to mark certain nodes.

The goal in graph exploration is to visited all vertices of the given graph. We say that a graph G is **explored** when each vertex of G has been visited by at least one agent. There are three variants of the exploration problem, which are in increasing order of difficulty: **perpetual exploration**, **exploration with stop** and **exploration with return**. If we want to achieve perpetual exploration, then the agent(s) are not required to terminate, but can traverse the graph indefinitely. For exploration with stop, we require the agent(s) to terminate, i.e., transition to a halting state, after a finite number of steps. Lastly, for exploration with return we require all agents to return to the starting vertex and then terminate. Note that in some cases, the agent(s) may not be able to recognize if the whole graph is explored and only perpetual exploration is feasable, or in other cases the agent(s) may not be able to return to the starting vertex. See the related work in Section 1.3 for details. A graph that cannot be explored by an agent (a set of agents) is called a **trap** for the agent(s). In some problems, it is additionally required that the agent(s) **map** the given graph, i.e., construct a representation of an edge-labeled graph isomorphic to the given graph.

1.2.3 Complexity

For a detailed introduction of the concepts presented in this section, the reader can refer to the textbook by Garey and Johnson [GJ79] or the respective chapter in the textbook by Korte and Vygen [KV18].

Informally, an **algorithm** is a sequence of well-defined operations or instruction for a set of valid inputs. The **time complexity** or **running time** of an algorithm is the number of operations of the algorithm on a given input, whereas the **space complexity** is the amount of space or memory required to store additional information during the execution of the algorithm. According to Church-Turing thesis everything that is computable by this intuitive idea of an algorithm can also be computed on a Turing machine [Chu36]. There are several equally powerful other formal models for

computation, such random access machines, which are equivalent in terms of time complexity and space complexity, i.e., for a suitable time measure and space measure the machines can simulate each other with polynomial overhead in time and constant factor overhead in space [GJ79; SE84]. That is the reason why we present most algorithms in pseudcode similar to modern programming languages as it would be extremely tedious to give a complete description in terms of a Turing machine. In Chapter 2, however, we also work with a description of an algorithm in form of a Turing machine and introduce an agent model which internally is utilizing a Turing machine. We therefore give an introduction to this model of computation and further cover some complexity classes relevant for this thesis.

A Turing machine consists of an infinite tape divided into cells, a read-write head, a finite set of states and a transition function decribing how the Turing machine transitions from one state to the next depending on the current state and the symbol read from the tape at the current position of the read-write head. Formally, a **deterministic Turing machine** M is a tuple $(Q, q_0, \overline{q}, \delta)$, where

- Q is the finite set of states of M,
- $q_0 \in Q$ is the starting state of M,
- $\overline{q} \in Q$ is the stop state of M,
- $\delta \colon Q \setminus \{\overline{q}\} \times \{0, 1, \sqcup\} \to Q \times \{0, 1, \sqcup\} \times \{L, R\}$ is the transition function of M, where $\{0, 1\}$ is the set of input symbols and $\sqcup$ is the blank symbol representing an empty tape cell.

For an input $x \in \{0, 1\}^*$, we assume that initially the input x is contained in the tape cells, the head of the Turing machine M is at the first symbol of x and all other symbols of x follow to the right of the head position. Every tape cell not containing a symbol of x contains the blank symbol $\sqcup$. The Turing machine M performs a computation as follows: If M reads the symbol $a \in \{0, 1, \sqcup\}$ at the current head possition, is in state $q \in Q$ and $\delta(a, q) = (q', a', S)$, then it writes the symbol $a' \in \{0, 1, \sqcup\}$ to the tape cell of the current head position, changes it state to $q' \in Q$ and moves the head left if $S = L$ or right if $S = R$. The Turing machine M continues its computation until it reaches its final state $\overline{q}$ or it can also run forever.

We define the **output** of the Turing machine to be the string $y \in \{0, 1\}^*$, that is contained in the tape cells when the Turing machine terminates beginning from the head position to the right until the first cell containing a blank symbol $\sqcup$.

The **running time** of the Turing machine M is described by the function $t_M \colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$, where $t_M(n)$ is the maximum number of computation steps that the Turing machine M needs on an input $x \in \{0, 1\}^*$ with length n (or ∞ if M runs forever). If there exists a polynomial p such that for all $n \in \mathbb{N}$, we have $t_M(n) \leq p(n)$, then M is a **polynomial-time** Turing machine.

The **memory requirement** of the Turing machine M is given by a function $m_M \colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$, where $m_M(n)$ is the total number of tape cells that are used in the computation, i.e., that do not contain the blank symbol $\sqcup$ at some point. In order to overcome the fact that the input length as well as the output length is always a lower bound on the memory requirement with this definition, we extend the definition of the Turing machine above to a Turing machine with three tapes and three

heads: a read-only **input tape**, a read-and-write **working tape** and a write only **output tape**. Then the memory requirement is defined as the total number of tape cells of the working tape that are used in the computation of the Turing machine.

In general, Turing machines are defined over an input symbols Σ, but for our purpose the case $\Sigma = \{0, 1\}$ is sufficient and we therefore introduce the Turing machine as above. Note that this does not change the computational power of the Turing machine.

A **language** L is a subset of $\{0, 1\}^*$ and the elements of $\{0, 1\}^*$ are called **words** or **binary strings**. We say that a deterministic Turing machine M accepts a word $x \in \{0, 1\}^*$ if and only if M terminates on the input x and outputs 1. We further say that M **decides** a language L if M terminates on every $x \in \{0, 1\}^*$ and it accepts $x \in \{0, 1\}^*$ if and only if $x \in L$. If additionally M is a polynomial-time Turing machine, then we say that L is **decidable in polynomial time**. A **decision problem** is a pair $\mathcal{P} = (X, Y)$, where $X \subseteq \{0, 1\}^*$ is a language decidable in polynomial time and $Y \subseteq X$. We refer to the elements of X as **instances**, the elements of Y as **yes-instances** and those of $X \setminus Y$ as **no-instances**. Moreover, we say that a deterministic Turing machine M **decides** a decision problem $\mathcal{P} = (X, Y)$, if M accepts exactly all $x \in Y$.

Another variant of a Turing machine is a **non-deterministic Turing machine**. A non-deterministic Turing machine differs from a deterministic Turing machine in the transition function that is a **transition relation** for a non-deterministic Turing machine, i.e., $\delta \subseteq Q \times \{0, 1, \sqcup\} \to Q \times \{0, 1, \sqcup\} \times \{L, R\}$. If a non-deterministic Turing machine M reads the symbol $a \in \{0, 1, \sqcup\}$ at the current head position, is in state $q \in Q$, than it can non-deterministically choose any $(a, q, q', a', S) \in \delta$, transition to the state q', writes $a' \in \{0, 1, \sqcup\}$ to the current tape cell, changes it state to $q' \in Q$ and moves the head left if $S = L$ or right if $S = R$. For a given input $x \in \{0, 1\}^*$, there can now be different possible outputs of the Turing machine depending on the **computation path**, i.e., the transitions chosen in every step of the computation.

The **running time** for a non-deterministic Turing machine M on an input $x \in \{0, 1\}^*$ is defined as the maximum number of computation steps over all computations paths and similarly the **memory requirement** as the maximum number of tape cells used over all computation paths. These definitions allow us to analogously define the running time and memory requirement for non-deterministic Turing machines. Furthermore, we say that a non-deterministic Turing machine M **accepts** a word $x \in \{0, 1\}^*$ if and only if there is one possible computation path of M on input x such that M terminates and outputs 1. The decidability of languages and decision problems for non-deterministic Turing machines is again defined analogously.

We further define the **configuration** of a Turing machine as a tuple (q, t, p), where q is the current state of the Turing machine, $t \in \{0, 1, \sqcup\}^{\mathbb{Z}}$ is the tape content and $p \in \mathbb{Z}$ is the head position. Here we identify every tape cell with an integer $z \in \mathbb{Z}$. Note that the configuration of a deterministic Turing machine completely describes the current state of the computation and uniquely determines the next configuration in the computation. We call a non-deterministic Turing machine **symmetric** if the graph describing the transitions between the configurations of the Turing machine is symmetric, i.e., if the Turing machine can change from a configuration (q, t, p) to a configuration (q', t', p')

by making a transition according to δ, then it can also a make a transition from the configuration (q', t', p') to change to (q, t, p). For a detailed introduction of symmetric Turing machines and related complexity classese, see [LP82].

We are now ready to define the following complexity classes.

P The class containing all decision problems $\mathcal{P}$ for which there is a polynomial-time deterministic Turing machine deciding $\mathcal{P}$.

NP The class containing all decision problems $\mathcal{P}$ for which there is a polynomial-time non-deterministic Turing machine deciding $\mathcal{P}$.

L The class containing all decision problems $\mathcal{P}$ for which there is a deterministic Turing machine deciding $\mathcal{P}$ that uses logarithmic memory.

NL The class containing all decision problems $\mathcal{P}$ for which there is a non-deterministic Turing machine deciding $\mathcal{P}$ that uses logarithmic memory.

SL The class containing all decision problems $\mathcal{P}$ for which there is a non-deterministic symmetric Turing machine deciding $\mathcal{P}$ that uses logarithmic memory.

A decision problem $\mathcal{P}_1 = (X_1, Y_1)$ **polynomially transforms** to a second decision problem $\mathcal{P}_2 = (X_2, Y_2)$ if there is a function $f : X_1 \to X_2$ computable in polynomial time such that $f(x_1) \in Y_2$ if and only if $x_1 \in Y_1$. A polynomial transformation is also referred to as a **Karp reduction**. Furthermore, a decision problem $\mathcal{P} \in$ NP is **NP-complete** if all other problems in NP polynomially transform to $\mathcal{P}$.

1.2.4 Offline and Online Optimization Problems

The introduction of the following concepts and notation in this section is based on the introduction in the textbook of Borodin and El-Yaniv [BE98].

A **discrete optimization problem** problem is a set $\mathcal{I} \subseteq \{0,1\}^*$ of **instances**, a set of **feasable solutions** $\mathcal{S}_I$ for every instance $I \in \mathcal{I}$, a **cost function** $c\colon \{(I, S) \mid I \in \mathcal{I},\ S \in \mathcal{S}_I\} \to \mathbb{R}$ computable in polynomial time and a **goal**, i.e., minimizing or maximizing the cost. For a given instance $I \in \mathcal{I}$, we write $\textsc{Opt}(I) := \min\{\ c(I,S) \mid S \in \mathcal{S}_I\}$ for the cost of an **optimum solution** in case of a minimization problem and $\textsc{Opt}(I) := \max\{c(I,S) \mid S \in \mathcal{S}_I\}$ for the cost of an optimum solution in case of a maximization problem. An algorithm for an optimization problem computes a feasable solution $S \in \mathcal{S}_I$ for every instance $I \in \mathcal{I}$ with $\mathcal{S}_I \neq \emptyset$. We write $\textsc{Alg}(I) := c(I,S)$ if the considered algorithm Alg computes solution $S \in \mathcal{S}_I$ on input I. If $\textsc{Alg}(I) = \textsc{Opt}(I)$ for all $I \in \mathcal{I}$ with $\mathcal{S}_I \neq \emptyset$, then Alg is an **exact algorithm**.

A decision problem or discrete optimization problem $\mathcal{P}_1$ **polynomially reduces** to an optimization $\mathcal{P}_2$ if there exists an exact polynomial algorithm for $\mathcal{P}_1$ using at most a polynomial number of calls to an exact algorithm for $\mathcal{P}_2$. This type of reduction is also referred to as **Turing reduction** and the algorithm for $\mathcal{P}_1$ using at most a polynomial number of calls to an exact algorithm for $\mathcal{P}_2$ is called a **polynomial time oracle algorithm**. A formal definition of this concept using oracle Turing ma-

chines can be found in [GJ79; KV18]. Moreover, an optimization problem or decision problem $\mathcal{P}$ is called **NP-hard** if all problems in NP polynomially reduce to $\mathcal{P}$.

Many interesting discrete optimization problems are NP-hard and there is thus no polynomial exact algorithm solving them under the assumption that NP $\neq$ P. In order to still find good (close to optimal) solutions for those problems in acceptable practical running time (e.g. polynomial), one can trade a loss in solution quality for a better running time, which leads to the concept of approximation algorithms. More precisely, an algorithm Alg is called an **asymptotic c-approximation algorithm** for a discrete optimization problem with the goal of minimization if

$$\textsc{Alg}(I) \leq c \cdot \textsc{Opt}(I) + \alpha \qquad \text{for all } I \in \mathcal{I}.$$

If $\alpha = 0$, we call Alg a **c-approximation algorithm**. For a maximization problem, an (asymptotic) c-approximation algorithm we require $\textsc{Alg}(I) \leq 1/c \cdot \textsc{Opt}(I) + \alpha$ for all $I \in \mathcal{I}$. In both cases the **approximation factor** or **approximation ratio** c satisfies $c \geq 1$ and the better the approximation, the closes the approximation factor c is to 1. A thorough introduction and study of approximation algorithms is for example given in [WS11].

In classical optimization problems the whole input is available to an algorithm at the beginning. There are many interesting problems, where this is not the case, and only a part of the input is received at a time and the algorithm already needs to output decisions only based on this partial input. Basically all graph exploration problems fall into this category. The graph to be explored is initially unknown and the algorithm, in this case the agents, need to make decision, e.g., which edges to traverse next, based on only the information they gathered so far, i.e., the part of the graph traversed so far. These type of problems are called **online problems** and an algorithm for such a problem is called an **online algorithm**. The classic optimization problems, where the whole input is known in advance, are in contrast to that refered to as **offline problems** and an algorithm which receives the complete input at the beginning an **offline algorithm**. An instance $I \in \mathcal{I}$ of an online problem is called an **input sequence** in order to emphazise that the input is received in many parts.

We measure the performance of an online algorithm using the concept of **competitive analysis** introduced by Sleator and Tarjan in [ST85]. In this framework, the cost of an online algorithm Alg on an instance $I \in \mathcal{I}$ is compared to the cost of an optimal offline solution $\textsc{Opt}(I)$, i.e., an optimal solution if the whole input is known in advance. An online algorithm Alg for a minimization problem is **c-competitive** if there is a constant α such that

$$\textsc{Alg}(I) \leq c \cdot \textsc{Opt}(I) + \alpha \qquad \text{for all finite input sequences } I \in \mathcal{I}.$$

For a maximization problem, a c-competitve algorithm Alg needs to satisfy $\textsc{Alg}(I) \leq 1/c \cdot \textsc{Opt}(I) + \alpha$ for all for all finite input sequences $I \in \mathcal{I}$. If $\alpha \leq 0$, then Alg is **strictly c-competitive**. We further call c the **competitive ratio** of the algorithm Alg. For further reading and a detailed introduction to online algorithms and competitive analysis the reader can refer to [BE98], for instance.

1.3 Related Work

The main aim of this section is to give a detailed systematic overview about the graph exploration literature and also cover the message delivery literature. We focus on the part of graph exploration literature most relevant for this thesis and give several pointers to other related work not covered in this section.

The vast amount of research on graph exploration and large number of different models makes it difficult to put the results in one general scheme. Nevertheless, we hope that our categorization of the results provides a fast and easy way to grasp the state-of-the-art of graph exploration and the main lines of research. Our main distinction is between single agent (Section 1.3.1) and collaborative (Section 1.3.2) exploration and undirected and directed graphs. We further distinguish between the objectives feasability, time, memory and energy. See also the Tables 1.1 and 1.2 for a concise overview of the related work.

1.3.1 Single Agent Exploration

Undirected Graphs. The exploration of plane labyrinths, i.e., finite connected subgraphs of the infinite 2-dimensional grid where edges are labeled with their cardinal direction, was the starting point of graph exploration research. Shannon [Sha51] constructed an actual physical device – Shannon's mouse – that could explore a 5×5 grid. Budach proved that one agent with constant memory and without any pebble cannot explore any plane labyrinth [Bud75; Bud78]. Later Hoffmann showed that also 1 pebble is not sufficient [Hof81]. On the positive side, Shah proposed an algorithm for an agent with 5 pebbles that can explore any plane labyrinth [Sha74]. This result was improved by Blum and Kozen who presented an algorithm using only 2 pebbles [BK78]. They also showed that exploration can be achieved utilizing a counter of size $O(\log n)$ instead of 2 pebbles.

For many years a central open problem in graph exploration was the question how much memory an agent needs to explore any undirected graph. It turned out that this problem is closely connected to the space complexity of the s-t-connectivity problem in undirected graphs, i.e., the problem of deciding if two vertices s and t are in the same connected component of a given graph. For instance, any exploration algorithm can be turned into an algorithm deciding s-t-connectivity by letting an agent start at s and returning yes if and only if the agent visits t during the exploration. The problem of undirected s-t connectivity is complete for the complexity class SL (see [LP82]), which was studied in an efford to answer the question whether the complexity classes NL and the class L are the same.

A big step towards understanding the space complexity of s-t-connectiviy and also graph exploration was the work by Aleliunas et al. [Ale+79], who showed that a random walk of length $O(\Delta^2 n^3 \log n)$ in an undirected graph with n vertices and maximum degree Δ visits all vertices with high probability. Moreover, the authors show the existence of a universal traversal sequence for all d-regular graphs on n vertices of length $O(d^2 n^3 \log n)$. Note that by adding a counter that keeps track of the number of edge traversals the first bound yields a randomized log-space exploration algorithm that terminates after a polynomial number of steps and explores an undirected graph with high prob-

ability if an upper bound on the number of vertices of the graph is known. In terms of complexity classes, the result by by Aleliunas et al. implies that s-t-connectiviy is contained in the class RL, the class of decision problems that can be solved by a randomized, log-space algorithm with one-sided error and the relationship of the complexity classes is

$$\mathrm{L} \subseteq \mathrm{SL} \subseteq \mathrm{RL} \subseteq \mathrm{NL}.$$

Finally, Reingold [Rei08] showed that s-t-connectiviy can be decided in log-space and therefore L = SL. His proof also yields a log-space constructible universal exploration sequence which can be used to devise a log-spacce exploration algorithm for undirected graphs [Rei08, Corollary 5.5]. As this algorithm utilizes an exploration sequence (and not a traversal sequence), it is essential that the agent can observe the label of the edge by which it enters a vertex. Universal traversal sequences of length $O(n^{\log n})$ can be constructed in $O(\log^2 n)$ space using Nisan's derandomization technique [Nis92]. Explicit construction of universal traversal sequences in log-space are only known for cycles [Ist88] and it remains an open problem whether universal traversal sequences of polynomial length can be constructed deterministically in log-space for general graphs.

Concerning a lower bound on the space complexity of graph exploration, the result by [Bud75; Bud78] already shows that constant memory is not sufficient to explore any graph. Later, Rollik constructed a trap for any set of k collaborating agents, i.e., a graph that the given set of agents do not explore [Rol80]. Although he never computes it explicitly, his work already implies a memory requirement of $\Omega(\log n)$ space for graph exploration. Finally, Fraigniaud et al. [Fra+05] show that for any agent with s states there exists a graph with $s + 1$ vertices which the agent does not explore. In terms of the memory in bits this result yields the same lower bound as the construction by Rollik, however, when considering the number of states of the agents the lower bound by Fraigniaud et al. is stronger.

For trees with maximum degree Δ, Diks et al. [Dik+04] gave a perpetual exploration algorithm that uses $O(\log \Delta)$ space, i.e., asymptotically not more than the space needed to store a single edge label. They showed that $\Omega(\log \log \log n)$ bits of memory are needed if the algorithm has to eventually terminate. If, in addition, the algorithm is required to terminate at the same vertex where it started, $\Omega(\log n)$ bits of memory are needed. A matching upper bound for the latter result was given by Ambuhl et al. [Amb+11].

Another natural objective for graph exploration is to minimize the exploration time, i.e., the number of edge traversals until the given graph is explored. In labeled graphs, depth first search can be used to explore an undirected graph with m edges in at most $2m$ steps. Note that m is a trivial lower bound for the problem, as every edge needs to be traversed before the agent can be sure that it explored the whole graph. In [PP99], Panaite and Pelc present an algorithm that requires $m+3n$ steps for exploring a graph of n nodes and m edges. This is an improvement over the depth-first search for dense graphs and shows that it is possible to exceed the lower bound m by a term depending only linearly on n.

If, however, the given graph is anonymous, minimizing the exploration time becomes consider-

ably harder. In a d-regular graph, for instance, an agent can gain no knowledge when traversing the graph and also has no way of recognizing when exploration is completed. If the number of vertices n or an upper bound on n is known, then it is possible to utilize universal traversal sequences or universal exploration sequences to completely explore the graph in this case. The length of a universal traversal sequence or universal exploration sequence for a d-regular graph is bounded by $O(dn^3 \log n)$ for $d \leq n/2-1$ [Kou03; Kah+89] and by $O(n^3 \log n)$ for $d \geq n/2$ [Kou03; Cha+97]. By using a transformation of a universal exploration sequence for 3-regular graphs to general undirected graphs (see [Kou03, Theorem 87] or Lemma 2.5 and its proof), we obtain a universal exploration sequence of length $O(n^4 \log n)$ for general graphs. Note that although the proof is not constructive, this bound already implies the existence of a polynomial space exploration algorithm that needs $O(n^4 \log n)$ edge traversals to explore any anonymous undirected graph because an agent can find a suitable exploration sequence in polynomial space by enumeration. There also is a lower bound of $\Omega(n^4)$ on the length of universal traversal sequences [BRT92]. However, this lower bound does not translate to a lower bound on the number of steps required for exploring an undirected graph as an agent can also make use of the fact that it observes the port number of the edge by which it enters a vertex. But for symmetric directed graphs this yields that $\Omega(n^4)$ steps are required for exploration in the worst case. For symmetric directed graphs the upper bound of $O(n^5 \log n)$ on the length of a universal traversal sequences by Aleliunas et al. [Ale+79] implies the existence of a $O(n^5 \log n)$ time algorithm for symmetric directed graphs.

A setting that is in between unlabeled and labeled graph exploration is to allow the agents to only distinguish certain vertices. Dudek et al. [Dud+91] showed that an agent provided with a pebble can explore and map an undirected graph in time $O(mn)$. For graphs with maximum degree Δ, Chalopin et al. [CDK10] showed that if the starting node can be recognized by the agent, then the graph can be explored and mapped in time $O(n^3\Delta)$ using $O(n\Delta \log n)$ bits of memory.

Another line of research, refered to as piecemeal exploration, focuses on minimizing the exploration time when the number of edge traversals an agent can do before returning to the starting vertex for refueling is bounded by $(2 + \alpha)r$, where α is some positive constant and r is the distance to the furthest node from the starting vertex. The problem was first considered by Betke et al. in [BRS95] and the authors presented an $O(m)$ algorithm for exploration of grid graphs with rectangular obstacles. In [Awe+99] an algorithm for piecemeal exploration of general graphs was proposed requiring $O(m + n^{1+o(1)})$ edge traversals. Finally, Ducan et al. gave an optimal algorithm for piecemeal exploration for generals graphs requiring only $\Theta(m)$ edge traversals [DKK06]. Their algorithm also extends to weighted graph and a similar model, where the agent is tethered by a rope of length $(2 + \alpha)r$ instead of requiring regular refueling.

Exploration of undirected weighted graphs was first considered by Kalyanasundaram and Pruhs in [KP94]. In their model, the graph is labeled and an agent arriving at a vertex v learns about all edge $\{v, w\} \in E$ incident to v including the edge weight $w(\{v, w\})$ and the vertex w at the other endpoint. Every time an agent traverses an edge e, it incurs a cost of $w(e)$, and the total exploration time of an agent is the sum over all edge weights (with multiplicities) traversed by the agent. Note

that it is important that an agent sees the neighbors of a vertex and thus does not need to traverse all edges of the graph as otherwise $\sum_{e \in E} w(\{v, w\})$ is a trivial lower bound on the exploration time and a depth-first search algorithm is already 2-competitive. The nearest neigbors greedy heuristic for the traveling salesman problem already yields a $\Theta(\log n)$-competitive algorithm for this problem [RSI77]. Kalyanasundaram and Pruhs propose a sophisticated algorithm which is 16-competitive on planar graphs [KP94]. Megow et al. show that the algorithm is in fact $16(1 + 2g)$ for graphs of genus at most g and provide a lower bound showing that it does not have a constant competitive ratio on general graphs [MMS12]. They also present an alternative $\Theta(\log n)$-competitive algorithm for the problem. It is an open problem, whether there exists a constant competitive algorithm in this model.

Directed Graphs. The main focus of research on exploration of directed graphs has been the exploration time. Deng and Papadimitriou considered the exploration of unknown labeled directed graphs, where the agent does not know the other endpoint of an edge that is has not traversed [DP99]. The offline version of the problem, i.e., traversing all edges of a given directed graph with the minimum number of edge traversals, is known as the Chinese postman problem and can be solved in polynomial time [EJ73]. Deng and Papadimitriou propose an online algorithm for the problem achieving a competitive ratio of $d^{O(d)}$, where d is the deficiency of the given graph G, i.e., the minimum number of edges that have to be added to make it Eulerian. They also show a lower bound of $\Omega(d)$ on the competitive ratio for deterministic algorithms and of $\Omega(d/\log d)$ for randomized algorithms. Note that there is a simple online algorithm that explores the graph in polynomial time $O(nm)$ by traversing the nearest edge, which has not been traversed, in every step. Albers and Henzinger propose the first algorithm with a subexponential competitive ratio of $d^{O(\log d)}$ for the problem [AH00]. Finally, Fleischer and Trippen give a deterministic exploration algorithm with a competitive ratio of $O(d^8)$, which is polynomial in d [FT05].

A variant of the above model is considered by Foerster and Wattenhofer in [FW16]. They consider weighted, labeled directed graphs and the main difference is that in their model the agent observes the vertex at the other endpoint of all outgoing edges at a vertex. This implies that an online algorithm does not necessarily have to traverse all edges to ensure that exploration is complete and the corresponding offline problem is the asymmetric traveling salesperson problem. They show that the competitive ratio is $\Theta(n)$ for this problem, even for euclidean planar graphs or unweighted graphs.

Exploration becomes considerably more difficult if the directed graph is unlabeled. In this case it is possible to construct a graph given an arbitrary agent such that the agent needs an exponential number of steps in n to visit all vertices, see the combination lock graph presented in [BS94] for details. Note that this holds even if we allow the agent to use randomization. If the number of vertices n or a bound on n is known, the exploration of a directed graph is still feasable by using a brute-force approach for instance: Iterate over all directed graphs G on n vertices and possible start positions v_0 and in every iteration first compute the current position v reached in G when following the edge labels traversed so far and then follow a sequence of edge labels exploring G.

If we allow the agent to utilize indistinguishable pebbles, the exploration time can be reduced to

Graph				Agent	Goal			Result	Reference
class	V	E	know		task	ter.	obj.		
laby	a	u	-	$O(1)$ memory, dist. pebbles	expl	y	feas	2 pebbles necessary and sufficient	[BK78] [Hof81]
laby	a	u	-	$O(1)$ memory, counter	expl	y	feas	exploration in $O(n^2)$ steps counter of size $O(\log n)$	[BK78]
tree	a	u	-		expl	n	mem	$O(\log \Delta)$ memory sufficient	[Dik+04]
tree	a	u	-		expl	y	mem	need $\Omega(\log \log \log n)$ memory, $O(\log n)$ sufficient	[Dik+04] [Amb+11]
graph	a	u	n		expl	y	mem	$\Theta(\log n)$ memory necessary and sufficient	[Rol80] [Rei08]
graph	a	u	-	$O(1)$ memory	expl	y	mem	$\Theta(\log \log n)$ pebbles necessary and sufficient	Cor. 2.9 Cor. 2.26
graph	a	u	Δ, n	$O(\log n)$ memory	expl	n	time	rand. walk explores graph in $O(n^3\Delta^2 \log n)$ steps whp.	[Ale+79]
graph	a	u	n	poly memory	expl	n	time	$O(n^4 \log n)$ steps sufficient	[Kou03]
graph	a	u	-	$l = (1+\alpha)r$ rope or $2l$ fuel	map	y	time	$\Theta(m + n/\alpha) = \Theta(m)$	[DKK06]
graph	l	u	-	poly mem	expl	n	time	at most $m + 3n$ steps	[PP99]
graph	l	wu	-	poly mem	expl	n	time	$O(\log n)$-competitive alg., $O(g)$-competitive alg. for graphs of genus g	[RSI77] [MMS12]
graph	a	d	n	indist. pebbles	map	y	time	need 1 pebbles for expl. in poly time	[Ben+02]
graph	a	d	-	indist. pebbles	map	y	time	need $\Theta(\log \log n)$ pebbles, $\Omega(n \log \Delta)$ memory for expl. in poly time	[Ben+02] [FI04]
graph	l	d	-	sees labels of neighbors	expl	y	time	$O(d^8)$ competitive on graphs with defficiency d	[FT05]
graph	l	wd	-	unaware of neighb. labels	expl	y	time	$\Theta(n)$ competitive for weighted graphs	[FW16]

a=anonymous, l=labeled, (w)u=(weighted) undirected, (w)d=(weighted) directed, y=yes, n=no

Table 1.1: Summary of main results for single agent exploration.

polynomial time, as shown by Bender et al. [Ben+02]. The authors gave an $O(n^8\Delta^2)$-time algorithm that uses one pebble and explores (and maps) a directed graph with maximum degree Δ, when n or an upper bound on n is known. For the case that such an upper bound is not available, they proved that $\Theta(\log\log n)$ pebbles are both necessary and sufficient to explore the graph in polynomial time. Concerning the space complexity of directed graph exploration in the same model, Fraigniaud and Ilcinkas [FI04] showed that $\Omega(n\log\Delta)$ bits of memory are necessary to explore any directed graph with n vertices and maximum degree Δ, even with a linear number of pebbles. As an upper bound on space complexity, they presented an algorithm requiring $O(n\Delta\log n)$ bits of memory that explore a graph in exponential time with a single pebble and terminates. They also gave an $O(n^2\Delta\log n)$-space algorithm running in polynomial time and using $O(\log\log n)$ indistinguishable pebbles for the case that n is not known.

Further related work. A lot of research has been done in more geometric and applied exploration settings, see the survey in [Rao+93] and [DS17].

Search problems, i.e., problems where a specific target t needs to be located in an unknown environment, are quite similar to exploration problems. In the worst case, for instance, the whole environment needs to be searched in order to locate the target t. If the target is found earlier, however, the algorithm can already terminate whereas in exploration we typically require the whole environment to be always visited. This fact leads to a different notion of (offline) optimum solution that a solution for a search problem is compared to. For a detailed introduction to search algorithms the reader can refer to the textbook [AG03]. A survey covering both search and exploration problems is given in [Ber98]. Another survey with the focus of exploration or search on the plane is given in [GK10].

Randomized graph exploration and the study of memory efficient graph exploration if the environment can be manipulated by providing a suitable labeling of the graph, for instance, is further considered in the survey [GR08]. Another line of research is the study of exploration of graphs that change over time as studied in [FMS09; EHK15].

1.3.2 Collaborative Exploration

Undirected Graphs. The first main focus of research for collaborative exploration was the feasibility and memory requirement for exploration of mazes and planar graphs. Blum and Kozen [BK78] showed that any maze 2-dimensional maze can be explored by two agents with constant memory. In the same work, the authors also show that 3-regular graphs are more difficult to explore than 2-dimensional mazes by exhibiting that no 3 agents with constant memory can explore all 3-regular graphs. In [BS77], Blum and Sakoda showed that no finite set of finite agents can explore any finite 3-dimensional maze, i.e., finite subgraphs of the 3-dimensional lattice graph $\mathbb{Z}^3$. A similar result was later obtained by Rollik [Rol80] for 3-regular graphs. He showed that for any set of k agents with s states each there is a planar graph that cannot be explored by the agents. Fraigniaud et al. [Fra+06b] revisited his construction and bounded the order of his trap for k agents with s states

by $\tilde{O}(s \uparrow\uparrow (2k+1))$, where $a \uparrow\uparrow b := a^{a^{\cdot^{\cdot^{a}}}}$ with b levels in the exponent. They further improve the bound on the order of the trap to $\tilde{O}(s \uparrow\uparrow (k+1))$. Note that the order of the barrier directly implies a lower bound on the memory requirement for k agents given a graph of size n. Concerning an upper bound for the memory requirement of collaborative graph exploration, we are not aware of any previous work that improves the $O(\log n)$ bits of memory algorithm for one agent. In Chapter 2 of this thesis, we present an algorithm that breaks the barrier of construct $O(\log n)$ bits and also construct a dramatically smaller trap. We thereby establish that $\Theta(\log \log n)$ agents can explore every graph on n vertices in polynomial time and terminate.

Collaborative exploration with the objective of minimizing the exploration time was first considered by Fraigniaud et al. in [Fra+06a]. The authors present an algorithm for agents using whiteboard communication that explores any tree in time $O(D + n/log k)$, where D is the diameter of the tree and k is is the number of collaborating agents. They also that the the offline problem of minimizing the exploration time of k collaborating agents is NP-hard even for tree topologies. Note that an optimal offline algorithm can explore a tree in time $\Theta(D + n/k)$ by dividing a depth-first traversal of the tree in k equal parts. Thus the algorithm in [Fra+06a] achieves a competitive ratio of $O(k/\log k)$. The authors also give a lower bound of $\Omega(2 - 1/k)$ on the competitive ratio. Later, Dynia et al. proposed a different algorithm, which is $O(D^{1-1/p})$ competitive, where p is the density of the tree, i.e., the minimum number $p \in \mathbb{N}$ which satisfies $|V'| \leq 4|h(T')|^p$ for all induced subtrees $T' = (V', E')$ of T. As $p \leq \log n$, the algorithm is $O(D)$ in general. Their algorithm only requires local communication, that is, agents can exchange information if they are at distance at most 1. For $k \leq \sqrt{n}$ agents, Dynia et al. constructed an improved lower bound of $\Omega(\log k/\log \log k)$ in [DŁS07]. Later, Disser et. al construct a different a different family of trees showing that the same lower bound on the competitive ratio also holds for $k \leq nlog^c n$ agents for any $c \in \mathbb{N}$ [Dis+17]. Another algorithm is presented by Brass et al. in [Bra+11] achieving a competitive ratio of $O(n/k + (k+D)^{k-1})$, which is an improved over the algorithm by Fraigniaud et al. for small values of k and D compared to n. They algorithm also relies on whiteboards communication of the agents. The special case of grid graphs is considered by Ortolf and Schindelhauer in [OS12]. They present an algorithm for exploring grid graphs which obtains a competitive ratio of $O(\log^2 n)$ and also show a general lower bound of $\Omega(\log k/\log \log k)$ on the competitive ratio if $k \leq n$. In [OS14], Ortolf and Schindelhauer adopted a recursive approach using global communication between the agents to improve the upper bound on the competitive ratio for certain values of the parameters n, k and D. In [Hig+14], they authors introduce a class of algorithm called greedy algorithms for the collaborative exploration of trees and show a lower bound of $O(k/\log k)$ on the competitive ratio of any greedy algorithm for weighted trees. Surprisingly, Dereniowski et al. show in [Der+15] that for $k \geq Dn^c$ agents for some constant $c > 1$, any graph can be explored in time $\Theta(D)$ using only local communication where agents can only exchange information at the same vertex. This means that for a large number of agents the competitive ratio is $O(1)$.

Another line of research in collaborative graph exploration is energy efficient graph exploration, where the number of edge traversals of an agent is bounded or the maximum number of edge traversals of an agents is to be minimized. Dynia et al. [DKS06] consider the problem of collaborative

exploration with a fixed number of agents while minimizing the maximum number of edges traversed by an agent. The agents can communicate at distance at most one and have to return to the starting vertex at the end. They presented an 8-competitive algorithm for trees and showed a lower bound of 1.5 on the competitive ratio for any deterministic algorithm. The upper bound was later improved to $4 - 2/k$ in [DŁS07]. The authors in [DDK15] considered tree exploration with no return for the case where the amount of energy B available to the agents is fixed and the goal is to minimize the number of agents used. They presented an algorithm with a competitive ratio of $O(\log B)$ for the case that the agents need to meet in order to communicate and showed that this is best possible. In our model considered in Chapter 3 the number of agents as well as the bound on the energy is fixed and we do not require the agents to explore the whole graph. Instead, we measure the performance of an online algorithm by the number of vertices explored by it compared to an optimal offline algorithm.

A very different variant of collaborative exploration, in which the agents are identical and initially dispered amoung the vertices of the graph, is considered in [Das+06; Das+07]. The agents further move asynchronously and can communicate by writing to whiteboards at every node. As the agents follow exactly the same protocol, exploration with termination is not always feasable because of symmetries (e.g., consider two agents starting on opposite vertices of an even length cicle). In [Das+07], the authors show that the problem of exploration with termination, leader election (i.e. selecting a leader among the agents) and rendezvous (i.e. gathering all agents at one vertex) are equivalent in this setting. present an algorithm achieving exploration with termination if k and n are coprime and using at most $O(m \cdot k)$ edge traversals and at most $O(\log n)$ bits of whiteboard memory at every node. The cases where exploration is possible are characterizied in [Das+06] including an algorithm that achieves leader election and thus also exploration with termination in all solvable cases. There are different variants of the algorithm with a different tradeoff between the number of edge traversals and whiteboard memory.

Directed Graphs. There has been only little research on collaborative exploration of directed graphs that we are aware of. For unlabeled, directed graphs Bender and Slonim show that two randomized agents can explore and map the given graph in expected polynomial time when global communication is allowed and n is not known [BS94]. Recall that in [Ben+02] the authors show that the same task can be achieved by one agent with $O(\log \log n)$ instinguishable pebbles.

Further related work. A survey covering both single agent and multi agent exploration and covers similar topics as this related work is given in [Das13].

A lot of research also has been done on collaborative exploration involving malacious software or a malacious environment that can destroy agents. The task is to explore the graph while removing the malacious software or locating malacious vertices that destroy agents. Surveys for collaborative exploration in unsafe environments are given in [FS06; Mar12].

The rendezvous problem, i.e., the task of gathering multiple, often identical agents at one location

Graph				Agent	Goal			Result	Reference
class	V	E	know		task	ter.	obj.		
laby	a	u	-	$O(1)$ mem.	expl	y	feas	2 agents necessary and sufficient	[BK78]
graph	a	u	-	identical, local com. via whiteboards	expl	y	feas	algorithm using $O(m \cdot k)$ edge traversals in all feasable cases	[Das+06] [Das+07]
graph	a	u	-	$O(1)$ mem. local com.	expl	y	feas	$\Theta(\log \log n)$ agents necessary and sufficient	Cor. 2.10 Cor. 2.25
tree	a	u	-		expl	y	time	CR $\leq O(k/\log k)$, CR $\geq \Omega(\log k/\log \log k)$ for $k \leq nlog^c n$ agents, $c \in \mathbb{N}$	[Fra+06a] [DŁS07] [Dis+17]
tree	a	u	-		expl	y	time	CR $= O(1)$ for $k \geq Dn^c$ agents, $c \in \mathbb{N}$, tree with diameter D	[Der+15]
tree	a	u	-	global com., fixed # agents	expl	y	energy	$3/2 \leq$ CR $\leq 4 - 2/k$ for minimizing max. # edges traversed by an agent	[DKS06] [DŁS07]
tree	a	u	-	local com., fixed energy B	expl	y	energy	CR $= \Theta(\log B)$ for minimizing # agents	[DDK15]
tree	a	u	-	global com., fixed energy B fixed # agents	expl	y	energy	$2.17 \leq$ CR ≤ 3 for maximizing total # vertices visited	Theo. 3.2 Theo. 3.6
graph	a	d	-	randomized, global com.	map	y	time	2 agents can map graph in polynomial time	[BS94]

a=anonymous, l=labeled, u= undirected, d=directed, com.=communication, y=yes, n=no

Table 1.2: Summary of main results for collaborative exploration.

of the environment, is closely related to collaborative graph exploration. Connections betweeb graph exploration and rendezvous were already mentioned in the related work above, see [Das+06; Das+07] for an example. A detailed introduction to rendezvous problems is given in the textbook by Gal and Alpern [AG03]. Surveys about rendezvous research are further given in [Pel12] and [Alp+13].

1.3.3 Message Delivery

The problem of transporting goods between sources and destinations has many real-world applications in logistics and has been studied in a lot of different variants.

In some cases the transportation of goods can be modeled as a network flow problem. Two prominent well-studied models are the minimum-cost flow problem for a single good or more generally the multi-commodity flow problem for multiple goods [KV18, Chapter 9 and 19]. While the first problem admits a polynomial time algorithm [EK72], the latter problem is known to be NP-hard [EIS76]. In contrast to our model, where the agents transporting the messages have capacity limits and transporting messages together does not incur additional costs, in these models there is a capacity limit on the edges and the cost of transportation grows linearly with the amount of goods transported.

More closely related to our problem is the *point-to-point delivery problem* studied in [LMS92]. In their model, a set of items have to be transported from different sources to different destinations and up to κ items can be transported together on an edge by an agent while the costs increase linearly in the number of agents used. The main difference to our model is that in this model there is an infinite supply of agents and agents can move for free if they do no transport any item. The authors show that the problem is NP-hard for $\kappa \geq 2$ and moreover give a polynomial algorithm for the case the number of items is constant.

In the *vehicle routing problem*, introduced by Dantzig and Ramser in [DR59], a set of items have to be delivered from a common source called depot to different destinations in a network by fleet of vehicles that all start at the depot. The number of items transported by a vehicle is further bounded by a capacity limit κ. For the special case of unbounded capacity, the vehicle routing problem corresponds to the *traveling salesman problem*, which is known to be NP-hard [GJ79]. A large number of variants of the vehicle routing problem have been considered since, differing in whether the vehicles start at a single depot or at different locations, the item sources are all at the depot or at different locations, the vehicles are identical or have different capacities or speeds. Moreover, variants with additional constraints motivated by applications have been considered such as a time window until deliveries must be made. Almost all variants of the vehicle routing problem are also NP-hard and most research focuses on integer programming techniques and heuristics. A survey of many types of vehicle routing problems is given int he book [TV02]. A survey about several vehicle routing problems with a hetereogeneous fleet of vehicles is further given in [BBV08]. The class of vehicle routing problem with pickup and delivery as well as time window contraints is referred to as *dial-a-ride* problems and covered in the survey [CL07].

The *Chinese postman problem*, i.e., the problem of finding the shortest tour traversing all edges of

a given undirected or directed graph, can also be viewed as a delivery problem and it can be solved in polynomial time [EJ73]. A generalization of this problem is the *stacker crane problem* introduced in [FHK78], which requires the tour to only traverse a given set of arcs of a mixed graph. The authors show that the stacker crane problem is NP-hard and also consider the k-person variants of the traveling salesman problem, Chinese postman problem and stacker crane problem. In the k-person variant, the goal is to find k tours starting and ending at the same vertex while minimizing the maximum cost of the k tours. This objective function is one of the main differences to the class of vehicle routing problems and our problem considered in Chapter 4, where we minimize the overall cost. In [FHK78] the authors show that all three k-person variants are NP-hard and they further present approximation algorithms for the problems.

Another related problem is the study of how to move a set of identical agents in a graph from a starting configuration to a desired final configuration while minimizing the overall or maximum movement of the agents. Demaine et al. [Dem+09] gave several approximation algorithms and inapproximability results for this problem on graphs. Moverover, for agents on a simple polygons several algorithms and inapproximability are presented in [Bil+13].

The delivery of multiple pieces of data or messages from different sources to different destinations by collaborating agents with different energy budgets, i.e., bounds on the distance they can travel, is called the *budgeted delivery problem*. The problem was first considered in [Cha+13] under the additional assumption that all destinations are the same. The authors show that the problem is strongly NP-hard even for a single source and uniform energy budgets and further present approximation and resource augmentation algorithms for the problem. In [Cha+14], it is shown that the problem is already weakly NP-hard for transporting a single piece of data from a source to a destination on the line. The general budgeted delivery problem with different sources and sinks is considered in [Bär+16]. The authors provide both hardness results and resource augmentation algorithm for the general budgeted delivery problem as well as a returning variant, where the agents additionally need to return to their starting vertex. Another variant of the problem where robots can share energy was considered in [Bam+17].

In the *weighted delivery problem* considered in Chapter 4, the agents can travel an arbitrary distance, but every agent has a different energy efficiency, which is the rate of energy consumption per unit distance traveled by the agent. The goal is to delivery all messages while minimizing the total energy consumption. A variant of this problem is considered in [BT17], where instead both the energy consumption as well as the delivery time is supposed to be minimized.

Further related is the problem of *convergecast*, i.e., in which every agent initially has a piece of information and one agent has to collect the information of all agents, and *broadcast*, i.e., in which the information of one agent has to be transferred to all other agents, as considered in [Ana+16; Czy+17].

Chapter 2

Space Efficient Graph Exploration

The space complexity of undirected graph exploration for one agent has received a lot of attention in the literature as it is closely related to the problem of undirected s-t-connectivity, which is complete for the complexity class SL. In his breakthrough result [Rei08], Reingold showed that undirected s-t-connectivity lies in L and therefore L = SL. His result also gave rise to a deterministic exploration algorithm that explores any anonymous undirected graph of size n in polynomial time and $O(\log n)$ space. Logarithmic memory is in fact necessary to explore all anonymous graphs with n vertices, see Fraigniaud et al. [Fra+05].

Already the early literature on graph exploration problems is rich with examples where exploration is made feasible or the time or space complexity of exploration by a single agent can be decreased substantially by either allowing the agent to mark vertices with pebbles or by cooperating with other agents. For instance, two-dimensional mazes can be explored by a single agent with finite memory using two pebbles [Sha74; BS77; BK78], or by two cooperating agents with finite memory [BK78], while a single agent with finite memory (and even a single agent with finite memory and a single pebble) does not suffice [Bud78; Hof81]. Directed anonymous graphs can be explored in polynomial time by two cooperating agents [BS94] or by a single agent with $\Theta(\log \log n)$ indistinguishable pebbles and $O(n^2 \Delta \log n)$ bits of memory [Ben+02; FI04], where Δ is the maximum out-degree in the graph. Note that a single agent needs at least $\Omega(n \log \Delta)$ bits of memory in this setting even if it is equipped with a linear number of indistinguishable pebbles [FI04] and it needs exponential time for exploration if it only has a constant number of pebbles and no upper bound on the number of vertices is known [Ben+02].

Less is known regarding the complexity of general undirected graph exploration by more than one agent or an agent equipped with pebbles, which we study in this chapter. The only result in this direction is due to Rollik [Rol80] showing that there are finite graphs, henceforth called **traps**, that a finite set of k agents each with a finite number s of states cannot explore. Fraigniaud et al. [Fra+06b] revisited Rollik's construction and observed that the traps have $\tilde{O}(s \uparrow\uparrow (2k+1))$ vertices, where $a \uparrow\uparrow b := a^{a^{\cdot^{\cdot^{a}}}}$ with b levels in the exponent and $\tilde{O}$ suppresses lower order terms. Fraigniaud et

al. also gave an improved upper bound of $\tilde{O}(s \uparrow\uparrow (k+1))$. While it is a rather straightforward observation that an agent with s states and p pebbles is less powerful than a set of $p+1$ agents with s states each, no better bounds for a single agent with pebbles were known. Even more striking is the lack of any non-trivial upper bounds for the exploration with several agents or the single agent exploration with pebbles for undirected graphs. Specifically, there was no algorithm known that explores an undirected graph with sublogarithmic space when more than one agent and/or pebbles are allowed.

We first give a formal introduction of the agent models for an agent with pebbles and multiple collaborating agent in Section 2.1. We further prove that an additional agent is more powerful than a pebble and a pebble is more powerful than a bit of memory.

Afterwards, in Section 2.2, we develop an algorithm that explores any graph with n vertices using $O(\log\log n)$ pebbles. Our algorithm terminates after having explored the graph and returns to the starting vertex. We further show that the exploration time, i.e., the number of edge traversals of the agent, is polynomial in the size of the graph. Our algorithm does not require n to be known and gradually increases the number of used pebbles during the course of the algorithm such that for any n-vertex graph at most $f(n)$ pebbles are used where $f(n) \in O(\log\log n)$. The fact that an additional agent is more powerful than a pebble allows to rephrase our single agent exploration algorithm with $O(\log\log n)$ pebbles as a multi-agent exploration algorithm with $O(\log\log n)$ agents and constant memory each.

As a perhaps surprising result, we show in Section 2.3 that this is optimal in terms of the asymptotic number of agents. To prove this lower bound, we construct a family of graphs with $O(s^{2^{5k}})$ vertices that trap any set of k agents with s states each. Our construction exhibits dramatically smaller traps with only a doubly exponential number of vertices compared to the traps of size $\tilde{O}(s \uparrow\uparrow (2k+1))$ and $\tilde{O}(s \uparrow\uparrow (k+1))$ due to Rollik [Rol80] and Fraigniaud et al. [Fra+06b], respectively. As a consequence of our improved bound on the size of the trap, we are able to show that, even if we allow $O((\log n)^{1-\epsilon})$ bits of memory for an arbitrary constant $\epsilon > 0$ for every agent, the number of agents needed for exploration is at least $\Omega(\log\log n)$. This construction also yields the lower bound for a single agent with pebbles, as $p+1$ agents with $O((\log n)^{1-\epsilon})$ bits of memory each are more powerful than one agent with $O((\log n)^{1-\epsilon})$ bits of memory and p pebbles. Our results allow to fully describe the tradeoff between the number of agents and the memory of each agent. When agents have $\Omega(\log n)$ memory, a single agent without pebbles explores all n-vertex graphs. For agents with $O((\log n)^{1-\epsilon})$ memory, $\Omega(\log\log n)$ agents are needed. On the other hand, when $\Omega(\log\log n)$ agents are available it is sufficient that each of them has only constant memory. In fact, already one agent with constant memory and $\Omega(\log\log n)$ pebbles are sufficient.

Bibliographic Information The results presented in this chapter are joint work with Yann Disser and Max Klimm. Parts of the results appeared in [DHK16], a more extensive version was published in [DHK18].

2.1 Agent Models

In this section, we formally introduce the agent model. We further give proofs of the intuitive facts that for undirected graph exploration an additional agent (with two states) is more powerful than a pebble, by showing that one of the agents can replicate the moves of the single agent while the others do not move independently and simply act as pebbles (Lemma 2.2). Moreover, we show that a pebble is more powerful than a bit of memory as one bit of memory can basically be encoded by either dropping or picking up a pebble Lemma 2.1. Note that all graphs considered in this chapter are undirected, anonymous, locally edge-labeled and connected.

We model an agent as a tuple $A = (\Sigma, \bar{\Sigma}, \delta, \sigma^*)$, where Σ is its set of states, $\bar{\Sigma} \subseteq \Sigma$ is its set of halting states, $\sigma^* \in \Sigma$ is its starting state, and δ is its transition function. The transition function governs the actions of the agent and its transitions between states based on its local observations. Its exact specifics depend on the problem considered, i.e., whether we consider a single agent or a group of agents and whether we allow the agents to use pebbles. Exploration terminates when a halting state is reached by all agents. Our model for an agent is based on a Mealy automaton. In particular this means that the output, i.e., the actions of the agent, can depend on the current state of the agent and the input, i.e., the local environment. This allows for a more memory efficient representation of the agents in contrast to a Moore automaton, whose output online depends on its current state.

2.1.1 Single Agent without Pebbles

The most basic model is that of a single agent A without any pebbles. In each step, the agent observes its current state $\sigma \in \Sigma$, the degree d_v of the current vertex v and the port number at v of the edge from which v was entered. The transition function δ then specifies a new state $\sigma' \in \Sigma$ of the agent and a move $l' \in \{0, \dots, d_v - 1\} \cup \{\perp\}$. If $l' \in \{0, \dots, d_v - 1\}$ the agent enters the edge with the local port number l', whereas for $l' = \perp$ it stays at v. Formally, the transition function is a partial function

$$\begin{aligned} \delta \colon \Sigma \times \mathbb{N} \times \mathbb{N} &\to \Sigma \times (\mathbb{N} \cup \{\perp\}), \\ (\sigma, d_v, l) &\mapsto (\sigma', l'). \end{aligned}$$

Note that the transition function only needs to be defined for l with $l < d_v$ and degrees d_v that actually appear in the class of graphs considered. It is standard to define the space requirement of an an agent with states Σ as $\log |\Sigma|$ as this is the number of bits needed to encode every state, see, e.g., Cook and Rackoff [CR80].

2.1.2 Single Agent with Pebbles

We may equip the agent A with a set $P = \{1, \dots, p\}$ of unique and distinguishable pebbles. At the start of the exploration the agent is carrying all of its pebbles. As before, the agent observes in each step the degree d_v of the current vertex v and the port number from which v was entered. In addition, the agent has the ability to observe the set of pebbles P_A that it carries and the set of pebbles P_v

present at the current vertex v. The transition function δ then specifies the new state $\sigma' \in \Sigma$ of the agent, and a move $l' \in \{0, \dots, d_v - 1\} \cup \{\perp\}$ as before. In addition, the agent may drop any subset $P_{\text{drop}} \subseteq P_A$ of carried pebbles and pick up any subset of pebbles $P_{\text{pick}} \subseteq P_v$ that were located at v, so that after the transition the set of carried pebbles is $P'_A = \big(P_A \setminus P_{\text{drop}}\big) \cup P_{\text{pick}}$ and the set of pebbles present at v is $P'_v = \big(P_v \setminus P_{\text{pick}}\big) \cup P_{\text{drop}}$. Formally, we have

$$\delta \colon \Sigma \times \mathbb{N} \times \mathbb{N} \times 2^P \times 2^P \to \Sigma \times (\mathbb{N} \cup \{\perp\}) \times 2^P \times 2^P,$$
$$(\sigma, d_v, l, P_A, P_v) \mapsto (\sigma', l', P'_A, P'_v).$$

The transition function δ is partial as it is only defined for $P_A \cap P_v = \emptyset$. We assume that the pebbles are actual physical devices dropped at the vertices so that no space is needed to manage the pebbles, thus, the space requirement of the agent is again $\log |\Sigma|$.

2.1.3 Collaborating Agents without Pebbles

Consider a set of k cooperative agents $A_1 = (\Sigma_1, \bar{\Sigma}_1, \delta_1, \sigma_1^*), \dots, A_k = (\Sigma_k, \bar{\Sigma}_k, \delta_k, \sigma_k^*)$ jointly exploring the graph. We assume that all agents start at the same vertex v_0 of the given graph G. In each step, all agents synchronously determine the set of agents they share a location with, as well as the states of these agents. Then, all agents move and alter their states synchronously according to their transition functions $\delta_1, \dots, \delta_k$. The transition function of agent i determines a new state σ' and a move l' as before. Formally, let

$$\Sigma_{-i} = (\Sigma_1 \cup \{\perp\}) \times \cdots \times (\Sigma_{i-1} \cup \{\perp\}) \times (\Sigma_{i+1} \cup \{\perp\}) \times \cdots \times (\Sigma_k \cup \{\perp\})$$

denote the states of all agents potentially visible to agent A_i where a $\perp$ at position j (or $(j-1)$ if $j \geq i$) stands for the event that agent A_i and agent A_j are located on different vertices. Then, the transition function δ_i of agent A_i is a partial function

$$\delta_i : \Sigma_i \times \Sigma_{-i} \times \mathbb{N} \times \mathbb{N} \to \Sigma_i \times (\mathbb{N} \cup \{\perp\}),$$
$$(\sigma_i, \boldsymbol{\sigma}_{-i}, d_v, l) \mapsto (\sigma'_i, l'_i).$$

The overall memory requirement is $\sum_{i=1}^{k} \log |\Sigma_i|$.

2.1.4 Relationship between Agent Models

In order to compare the capability of an agent A with s states and p pebbles to another agent A' with s' states and p' pebbles or a set of agents $\mathcal{A}$, we use the following notion: We say that the walk of an agent A is **reproduced** by an agent A' in a graph G, if the sequence of edges traversed by A is a subsequence of the edges visited by A' in G. Put differently, A traverses the same edges as A' in the same order, but for every edge traversal of A the agent A' can do an arbitrary number of intermediate edge traversals. Similarly, we say that a set of agents $\mathcal{A}$ reproduces the walk of an agent A in G, if there is an agent $A' \in \mathcal{A}$ such that A' reproduces the walk of A in G.

We first formally show the intuitive fact that pebbles are more powerful than memory bits.

Lemma 2.1. *Let A be an agent with s states and p pebbles exploring a set of graphs $\mathcal{G}$. Then there is an agent A' with six states and $p + \lceil \log s \rceil$ pebbles that reproduces the walk of A on every $G \in \mathcal{G}$ and performs at most three edge traversals for every edge traversal of A.*

Proof. As the set of graphs $\mathcal{G}$ that can be explored by an agent with s states and p pebbles is non-decreasing in s, it suffices to show the claimed result for the case that s is an integer power of two. Let $A = (\Sigma, \bar{\Sigma}, \delta, \sigma^*)$ be an agent with a set of p pebbles P and $s = |\Sigma| = 2^r$, $r \in \mathbb{N}$ states exploring all graphs $G \in \mathcal{G}$. In the following, we construct an agent $A' = (\Sigma', \bar{\Sigma}', \delta', \sigma^{*\prime})$ with six states $\Sigma' = \{\sigma^{*\prime}, \sigma_{\text{comp}}, \bar{\sigma}_{\text{halt}}, \sigma_{\text{back}-1}, \sigma_{\text{back}-2}, \sigma_{\text{swap}}\}$, one halting state $\bar{\Sigma}' = \{\bar{\sigma}_{\text{halt}}\}$, and a set P' of $|P'| = p+r$ pebbles. The general idea is to let A' store the state of A by dropping and retrieving the additional r pebbles. To this end, we identify p of the pebbles of A' with the p pebbles of A and call the additional set of r pebbles P'_Σ, i.e., $P' = P \cup P'_\Sigma$ with $|P| = p$ and $|P'_\Sigma| = r$, respectively. Since $|P'_\Sigma| = r$ and $|\Sigma| = s = 2^r$, there is a canonical bijection $f : \Sigma \to 2^{P'_\Sigma}$. Every edge traversal of agent A in a state σ, will be simulated by agent A' in the computation state σ_{comp} while carrying the set of pebbles $f(\sigma)$ plus the additional pebbles that A is carrying. We need the additional states $\sigma_{\text{back}-1}$, $\sigma_{\text{back}-2}$, σ_{swap} to move all pebbles in P'_Σ encoding the state of A to the next vertex in some intermediate steps.

At the start of the exploration, A' remains at the starting vertex and stores the starting state σ^* of agent A by dropping the set of pebbles $(P'_\Sigma \setminus f(\sigma^*))$. Formally, we define the transition from the starting state $\sigma^{*\prime}$ of agent A' as

$$\delta'(\sigma^{*\prime}, d_v, l, P', \emptyset) = (\sigma_{\text{comp}}, \bot, f(\sigma^*) \cup P, (P'_\Sigma \setminus f(\sigma^*)).$$

for all $d_v, l \in \mathbb{N}$.

Next, we define the transition function δ' of A' for the case that A' is in its computing state σ_{comp}, i.e., we want to simulate the change of state of A and traverse the same edge. If $\sigma = f^{-1}(P_{A'} \cap P'_\Sigma)$ is the current state of agent A and agent A transitions according to

$$\delta(\sigma, d_v, l, P_{A'} \cap P, P_v \cap P) = (\sigma', l', P'_A, P'_v) \tag{2.1}$$

with $\sigma' \in \Sigma$, $l' \in \mathbb{N}$ and $P'_A, P'_v \in 2^{P'}$, then we define

$$\delta'(\sigma_{\text{comp}}, d_v, l, P_{A'}, P_v) = \begin{cases} (\sigma_{\text{comp}}, \quad l', P'_A \cup f(\sigma'), P'_v \cup (P'_\Sigma \setminus f(\sigma')) & \text{if } l' = \bot \text{ and } \sigma' \notin \bar{\Sigma}, \\ (\sigma_{\text{back}-1}, l', P'_A \cup f(\sigma'), P'_v \cup (P'_\Sigma \setminus f(\sigma')) & \text{if } l' \neq \bot \text{ and } \sigma' \notin \bar{\Sigma}, \\ (\sigma_{\text{halt}}, \quad l', P'_A \cup f(\sigma'), P'_v \cup (P'_\Sigma \setminus f(\sigma')) & \text{else.} \end{cases} \tag{2.2}$$

Note that before and after this transition the subset of pebbles from P'_Σ carried by A' encodes the state of A via the bijection f. However, if A traverses an edge without entering a halting state, we also need to fetch the remaining pebbles from P'_Σ from the previous vertex to be able to encode the state of A in the future. To this end, A' switches to the state $\sigma_{\text{back}-1}$. The fetching will be done in three steps: First, A' drops all pebbles in $f(\sigma')$, moves to the previous vertex and changes its state to $\sigma_{\text{back}-2}$. Formally, this means

$$\delta'(\sigma_{\text{back}-1}, d_v, l, P_{A'}, P_v) = \left(\sigma_{\text{back}-2}, l, P_{A'} \setminus P'_\Sigma, P_v \cup \left(P'_\Sigma \cap P_{A'}\right)\right)$$

for all $d_v, l \in \mathbb{N}$ and $P_{A'}, P_v \in 2^{P'}$ with $P_{A'} \cap P_v = \emptyset$. Then it picks up the pebbles in $\left(P'_\Sigma \setminus f(\sigma')\right)$, returns to the current vertex of A and changes its state to σ_{swap}, i.e.,

$$\delta'(\sigma_{\text{back}-2}, d_v, l, P_{A'}, P_v) = \left(\sigma_{\text{swap}}, l, P_{A'} \cup \left(P'_\Sigma \cap P_v\right), P_v \setminus P'_\Sigma\right)$$

for all $d_v, l \in \mathbb{N}$ and $P_{A'}, P_v \in 2^{P'}$ with $P_{A'} \cap P_v = \emptyset$. Lastly, agent A' swaps the set of carried pebbles $P'_\Sigma \setminus f(\sigma')$ and the set $f(\sigma')$ of pebbles on the current vertex by performing the transition

$$\delta'(\sigma_{\text{swap}}, d_v, l, P_{A'}, P_v) = \left(\sigma_{\text{comp}}, \bot, P_{A'} \cup \left(P'_\Sigma \cap P_v\right), P_v \cup \left(P'_\Sigma \cap P_{A'}\right)\right)$$

for all $d_v, l \in \mathbb{N}$ and $P_{A'}, P_v \in 2^{P'}$ with $P_{A'} \cap P_v = \emptyset$.

A simple inductive proof establishes that the state σ of A in every step of the exploration of a graph $G \in \mathcal{G}$ corresponds to the set of pebbles in P'_Σ carried by A' in its computation state σ_{comp}, i.e., $\sigma = f^{-1}\left(P_{A'} \cap P'_\Sigma\right)$. Moreover, if agent A in state σ traverses an edge $\{v, w\}$ from a vertex v to a vertex w and does not move to a halting state, then A' will traverse the edge $\{v, w\}$ three times and afterwards again the set of pebbles carried by A will correspond to $P_{A'} \cap P$ and the state of A to $\sigma = f^{-1}\left(P_{A'} \cap P'_\Sigma\right)$. If A remains at the same vertex or moves to a halting state then this transition is mirrored by a single transition of agent A'. In particular, agent A' visits exactly the same vertices as A in every graph $G \in \mathcal{G}$ while performing at most three times the number of edge traversals. □

Next, we show the intuitive result that an additional agent is more powerful than a pebble.

Lemma 2.2. *Let A be an agent with s states and p pebbles exploring a set $\mathcal{G}$ of graphs. Then, there is a set $\mathcal{A} = (A_0, \ldots, A_p)$ of $p+1$ agents, where A_0 has s states and all other agents have two states, that reproduce the walk of A in every graph $G \in \mathcal{G}$. Moreover, for every edge traversal of A each agent in $\mathcal{A}$ performs at most one edge traversal.*

Proof. Let $A = (\Sigma, \bar{\Sigma}, \delta, \sigma^*)$ be an agent with $|\Sigma| = s$ and a set $P = \{1, \ldots, p\}$ of p pebbles exploring all graphs $G \in \mathcal{G}$. We proceed to construct a set $\mathcal{A} = \{A_0, \ldots, A_p\}$ of $p+1$ agents $A_i = (\Sigma_i, \bar{\Sigma}_i, \delta_i, \sigma_i^*)$, $i \in \{0, \ldots, p\}$ that reproduces the walk of A on all graphs $G \in \mathcal{G}$. In this construction, agent A_0 represents the original agent A while every agent A_i for $i > 0$ represents a pebble.

For agent A_0, we set $\Sigma_0 = \Sigma$, $\bar{\Sigma}_0 = \bar{\Sigma}$, and $\sigma_0^* = \sigma^*$. For every agent A_i with $i \in P$, we set $\Sigma_i = \{c_i, d_i\}$, $\bar{\Sigma}_i = \Sigma_i$, and $\sigma_i^* = c_i$. Intuitively, the state c_i simulates that pebble i is carried and d_i simulates that the pebble is dropped. In every step, we let agent A_0 and the agents A_i corresponding to a carried pebble do the same transitions as agent A. Agents that are not sharing their current vertex with A_0 remain at their vertex and in their state. Let $\sigma_{-i,j}$ for $i, j \in \{0, \ldots, p\}$ with $i \neq j$ denote the state of agent A_j visible to agent A_i, i.e., $\sigma_{-i,j} = \sigma_j$ if A_i and A_j share the same vertex and $\sigma_{-i,j} = \bot$ otherwise. Specifically, to define the transition functions $\delta_i(\sigma_i, \boldsymbol{\sigma}_{-i}, d_v, l)$ for $i \in \{0, \ldots, p\}$, $\sigma_i \in \Sigma_i$, $\boldsymbol{\sigma}_{-i} \in \Sigma_{-i}$ and $d_v, l \in \mathbb{N}$, we first compute

$$\delta(\sigma_0, d_v, l, \{i \in P : \sigma_{-0,i} = c_i\}, \{i \in P : \sigma_{-0,i} = d_i\}) = (\sigma'_0, l', P'_A, P'_v)$$

with $\sigma'_0 \in \Sigma$, $l' \in \mathbb{N}$, and $P'_A, P'_v \in 2^P$. We then set

$$\delta_0(\sigma_0, \boldsymbol{\sigma}_{-0}, d_v, l) = (\sigma'_0, l')$$

and

$$\delta_j(\sigma_j, \boldsymbol{\sigma}_{-j}, d_v, l) = \begin{cases} (\sigma_j, \bot) & \text{if } \sigma_0 = \bot \\ (c_j, l') & \text{if } \sigma_0 \neq \bot \text{ and } j \in P'_A \\ (d_j, \bot) & \text{if } \sigma_0 \neq \bot \text{ and } j \in P'_v \end{cases}$$

for all $j \in P$.

To finish the proof, fix a graph $G \in \mathcal{G}$ and consider the transitions of agent A and the set of agents $\mathcal{A}$ in G. A simple inductive proof shows that after i transitions, the state and position of agent A equals the state and position of agent A_0, the position of agent A_j equals the position of pebble j and $\sigma_j = c_j$ if and only if pebble j is carried by A for all $j \in P$. This implies the claim. □

Note that, for ease of presentation, we allow agents to make transitions even when they are in one of their halting states. We need this property in the proof above to show that two-state agents are more powerful than pebbles (cf. Lemma 2.2) in general. However, this reduction only needs agents to make transitions from their halting states to other halting states, and only when colocated with another agent that has not yet reached a halting state. Furthermore, our main algorithm for single-agent exploration with pebbles that we devise in Section 2.2 has the special property that the agent A_0 returns to the starting vertex carrying all pebbles after having explored the graph. Thus, for our algorithm it is not necessary that agents can make transitions from halting states as we could add an additional halting state to the two-state agents to which they transition once exploration is complete and A_0 has returned to the starting vertex.

2.2 Exploration Algorithms

In this section, we devise an agent exploring any graph on at most n vertices with $O(\log \log n)$ pebbles and $O(\log \log n)$ memory. By the reductions between the agents' models given in Section 2.1.4 this implies that an agent with $O(\log \log n)$ pebbles and constant memory can explore any n-vertex graph and that a set of $O(\log \log n)$ agents with constant memory each can explore any n-vertex graph.

For the algorithm, we use the concept of universal exploration sequences due to Koucký [Kou02], see Section 1.2.2. One of our main building blocks is the algorithm of Reingold [Rei08] that takes n and d as input and deterministically constructs an exploration sequence universal to all d-regular graphs using $O(\log n)$ bits of memory. The general idea of our algorithm is to run Reingold's algorithm with a smaller amount of seed memory a. As the seed memory is substantially less than $O(\log n)$, the algorithm will, in general, fail to explore the whole graph. We show in Lemma 2.5, however, that the algorithm will visit $2^{\Omega(a)}$ distinct vertices. Reinvoking Reingold's algorithm allows us to deterministically walk along these vertices in the order of exploration of Reingold's algorithm. Using this traversal, we encode additional memory by placing a subset of pebbles on the vertices along the walk as explained formally in Theorem 2.7. Having boosted our memory this way, we again run Reingold's algorithm, this time with more memory, and recurse. At some recursion depth, running

Reingold's algorithm with a^* bits of memory will visit less than $2^{\Omega(a^*)}$ distinct vertices. In the proof of Theorem 2.8, we show that this can only happen when the graph is fully explored which allows to terminate the algorithm when this event occurs and return to the starting vertex . The ability of our algorithm to terminate and return to the starting vertex after successful exploration, stands in contrast to Reingold's algorithm that is only able to terminate when being given the number n of vertices as input.

There are a couple of technical difficulties to make these ideas work. The main challenge is that the memory generated by placing pebbles along a walk in the graph is implicit and can only be accessed and altered locally. To still make use of the memory, we do not work with Reingold's algorithm directly but consider an implementation of Reingold's algorithm on a Turing machine with logarithmically bounded working tape. We show that the tape operations on the working tape can be reproduced by the agent by placing and retrieving the pebbles on the walk as explained in detail in the proof of n Theorem 2.7. This allows to use the memory encoded by the pebble positions for further runs of Reingold's algorithm. In each recursion, we only need a constant number of pebbles and additional states. We further show in Theorem 2.8 that $O(\log \log n)$ recursive calls are sufficient to explore an n-vertex graph so that the total number of pebbles needed is $O(\log \log n)$.

A second challenge is that Reingold's algorithm produces a universal exploration sequence for regular graphs which our graph need not be. A natural approach to circumvent this issue is to apply the technique of Koucký [Kou03] that allows to locally view vertices with degree d as cycles of $3d$ subvertices with degree 3 each. Unfortunately, this approach requires $O(\log d)$ bits of memory if we keep track of the current subvertex which may exceed the memory of our agent. To circumvent this issue, we store the current subvertex only implicitly and navigate the graph in terms of subvertex index offsets instead of the actual subvertex indices. This technique is explain in detail in the proof of Lemma 2.5

The following fundamental result of Reingold [Rei08] establishes that universal exploration sequences can be constructed in logarithmic space.

Theorem 2.3 ([Rei08, Corollary 5.5]). *There exists an algorithm taking n and d as input and producing in $O(\log n)$ space an exploration sequence universal for all connected d-regular graphs on n vertices.*

Reingold's result implies in particular that there is an agent without pebbles and $O(n^c)$ states for some constant c that explores any d-regular graph with n vertices when both n and d are known. We further note that Reingold's algorithm can be implemented on a Turing machine that has a read/write tape of length $O(\log n)$ as work tape and writes the exploration sequence to a write only output tape, see [Rei08, Section 5] for details. For formal reasons the Turing machine in [Rei08] additionally has a read-only input tape from which it reads the values of n and d encoded in unary so that the space complexity of the algorithm is actually logarithmic in the input length. For our setting, it is sufficient to assume that n and d are given as binary encoded numbers on the working tape of length $O(\log n)$, as we care only about the space complexity of exploration in terms of the number of vertices n.

As a first step, we show in Lemma 2.4 how to modify Reingold's algorithm for 3-regular graphs

Algorithm 2.1: Turing machine M computing exploration sequence for 3-regular graphs.

Input: $z \in \mathbb{N}$

Output: exploration sequences $w \in \{0,1,2\}^*$

for $t \in \{1, \dots, 2a\}$ **do**
  if $t \leq a$ **then**
    run M_0 for t steps to obtain element e_t of the exploration sequence generated by M_0
    output e_t
  else if $t = a + 1$ **then**
    output 0
  else if $t \geq a + 2$ **then**
    run M_0 for $2a + 2 - t$ steps to obtain element e_{2a+2-t} of exploration sequence of M_0
    output $-e_{2s+2-t} \bmod 3$

to yield a closed walk containing an exponential number of vertices in terms of the memory used. Afterwards, we extend this result to general graphs in Lemma 2.5.

Lemma 2.4. *For any $z \in \mathbb{N}$, there exists a $O(\log z)$-space algorithm producing an exploration sequence $w \in \{0,1,2\}^*$ such that for all connected 3-regular graphs G with n vertices the following hold:*

(a) an agent following w in G explores at least $\min\{z, n\}$ distinct vertices,

(b) w yields a closed walk in G,

(c) the length of w is bounded by $z^{O(1)}$.

Proof. By Theorem 2.3, there is a Turing machine M_0 with a tape of length $O(\log z)$ producing a universal exploration sequence $e_1, e_2, \dots$ for any 3-regular graph on exactly $4z$ vertices. Let c_{M_0} be the number of configurations of M_0 and $a := 12zc_{M_0} + 1$. Here the number of configurations of M_0 is the number of possible combinations of Turing state, tape contents and head position of M_0.

The Turing machine M producing an exploration sequence w with the desired properties is given in Algorithm 2.1. By construction, the sequence w produced by M is

$$e_1, e_2, \dots, e_a, 0, (-e_a \bmod 3), (-e_{a-1} \bmod 3), \dots, (-e_2 \bmod 3).$$

We first show that this sequence corresponds to a closed walk in any 3-regular graph. Let an agent A start at a vertex v_0 in some graph 3-regular G, follow the exploration sequence w, and, for $i \in \{1, \dots, a\}$, let v_i be the vertex reached after following w up to e_i. Then the offset 0 takes the agent back from v_a to v_{a-1} and afterwards $-e_i \bmod 3$ takes agent A from v_{i-1} to v_{i-2}. Thus, at the end the agent returns to v_0, which yields property (b).

Moreover, the number of configurations c_{M_0} of the Turing machine M_0, i.e., the number of possible combinations of state, head position, and tape contents, is bounded by $z^{O(1)}$, because the working tape has length $O(\log z)$. Hence, the length of w, i.e., $2a = 2 \cdot (12zc_{M_0} + 1)$, is also bounded by $z^{O(1)}$,

which yields property (c). As the auxiliary variable t ranges from 1 to $2a$ and running the Turing machine M_0 for t steps can be implemented in $O(\log z)$ space, the Turing machine M can be implemented to run in $O(\log z)$ space.

It is left to show is that an agent following w in an arbitrary connected 3-regular graph with n vertices explores at least $\min\{z, n\}$ vertices. For the sake of contradiction, assume there exists some 3-regular graph G on n vertices so that an agent A starting in a vertex v_0 and following the exploration sequence w produced by M only visits a set of vertices V_0 with $|V_0| < \min\{z, n\}$. Let G_0 be the subgraph of G induced by V_0. Note that, since $|V_0| < n$ by assumption, at least one vertex in G_0 has degree less than 3. We now extend G_0 to a connected 3-regular graph with $4z$ vertices as follows. First, we let G_1 be the graph G_0 after adding an isolated vertex if V_0 is odd and we let V_1 be the vertex set of G_1. We further let G_2 be a cycle of length $4z - |V_1|$ with opposite vertices connected by an edge. Note that $4z$ and $|V_1|$ are even and G_2 is 3-regular. As long as G_1 contains at least one vertex of degree less than 3, we delete an edge $\{w, w'\}$ connecting opposite vertices in the cycle in G_2 and for w and then w' add an edge from this vertex to a vertex of degree less than 3 in G_1 (possibly the same). This procedure terminates when all vertices in G_1 have degree 3, since G_2 contains $4z - |V_1| \geq 3z \geq 3|V_1|$ vertices and there cannot be a single vertex of degree 2 left in G_1, as this would mean that the sum of all vertex degrees in G_1 is odd. The labels in $\{0, 1, 2\}$ at both endpoints of every edge not in G_0 are chosen arbitrarily. Let H be the resulting 3-regular graph with $4z$ vertices containing G_0 as induced subgraph.

By construction, the walk of an agent A starting in H at v_0 and following w is the same as the walk in G starting in v_0 and following w. In particular, the agent A does not explore H. Let now A_0 be an agent following the exploration sequence w_0 produced by M_0 starting in vertex v_0 in H. As the first a values of w and w_0 coincide, the walk of agent A_0 in H up to step a is the same as that of agent A. Recall that $a = 3 \cdot 4zc_{M_0} + 1$. This implies that in the first a steps there must be a vertex v in H visited twice by agent A_0 (there are $4z$ vertices in H) and in both visits, the label to the previous vertex (there are 3 possible labels) is the same and the Turing machine M_0 producing the exploration sequence w_0 is in the same configuration (there are c_{M_0} possible configurations) in both visits. But this implies that the behaviour of A_0 in H becomes periodic and it only visits the set of vertices already visited in the first a steps, i.e., the set of vertices V_0. We conclude that A_0 does not explore H, contradicting that w_0 is a universal exploration sequence for all 3-regular connected graphs on $4z$ vertices. □

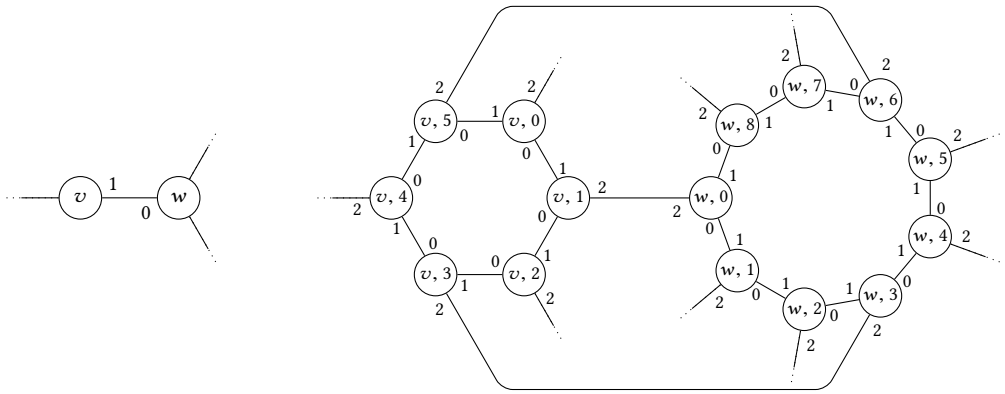
We proceed to give a similar result for non-regular graphs.

Lemma 2.5. *For any $z \in \mathbb{N}$, there exists a $O(\log z)$-space algorithm producing an exploration sequence $w \in \{-1, 0, 1\}^*$ such that for all connected graphs G with n vertices the following hold:*

(a) an agent following w in G explores at least $\min\{z, n\}$ distinct vertices,

(b) w yields a closed walk in G,

(c) the length of w is bounded by $z^{O(1)}$.



(a) Original graph G. *(b)* 3-regular graph G_{reg}.

Figure 2.1: Example for the transformation of a graph G to a 3-regular graph G_{reg}. A vertex v of degree 2 is transformed to a cycle containing 6 vertices and for the edge $\{v, w\}$, three edges are added to the graph.

Proof. Let M_{reg} be the Turing machine of Lemma 2.4 with a tape of length bounded by $O(\log z)$ producing a universal exploration sequence $w_{\text{reg}} \in \{0, 1, 2\}^*$ such that an agent following w_{reg} in some 3-regular graph with n vertices visits at least $\min\{3z^2, n\}$ distinct vertices.

To prove the statement, we transform this universal exploration sequence for 3-regular graphs to a universal exploration sequence universal for general graphs by using a construction taken from Koucký [Kou03, Theorem 87]. In this construction, an arbitrary graph G with n vertices is transformed into a 3-regular graph G_{reg} as follows: We replace every vertex v of degree d_v by a circle of $3d_v$ vertices $(v, 0), \ldots, (v, 3d_v - 1)$, where the edge $\{(v, i), (v, i + 1 \bmod 3d_v)\}$ has port number 0 at (v, i) and port number 1 at $(v, i + 1 \bmod 3d_v)$, see also Figure 2.1 for an example of this construction. For any edge $\{v, w\}$ in G with port number i at v and j at w, we add the three edges $\{(v, i), (w, j)\}, \{(v, i + d_v), (w, j + d_w)\}, \{(v, i + 2d_v), (w, j + 2d_w)\}$ with port numbers 2 at both endpoints to G_{reg}.

Observe that there are only two labelings of edges in G_{reg}, edges with port number 2 at both endpoints and edges with port numbers 0 and 1. In particular, one port number of an edge can be deduced from the other port number. As a consequence, given the previous edge label and the edge offsets from the exploration sequence w_{reg} produced by M_{reg}, the next edge label can be computed without knowing the edge label of the edge by which the vertex was entered. In other words, we can transform the sequence of edge label offsets given by w_{reg} to a traversal sequence, i.e., a sequence of absolute edge labels $l_0, l_1, \ldots$ of G_{reg}.

We proceed to define the Turing machine M producing an exploration sequence $w \in \{-1, 0, 1\}^*$ with the desired properties as shown in Algorithm 2.2. First of all, note that the next edge label l_i in G_{reg} can be computed from the last edge label in G_{reg} and the offset $w_{\text{reg}}(i)$ in constant space (line 5 of Algorithm 2.1). Thus, M can be implemented in $O(\log z)$ space. By assumption, the length of the

Algorithm 2.2: Turing machine M computing exploration sequence for arbitrary graphs.

Input: $z \in \mathbb{N}$

Output: exploration sequences $w \in \{-1, 0, 1\}^*$

```
1  i := 0
2  output 0, 0
3  while M_reg has not terminated do
4      obtain next offset w_reg(i) from M_reg
5      compute edge label l_i in G_reg
6      if l_i = 0 then
7          output 1, 0
8      else if l_i = 1 then
9          output −1, 0
10     else if l_i = 2 then
11         output 0
12     i := i + 1
```

exploration sequence produced by M_{reg} is bounded by $z^{O(1)}$. Hence, also the length of the exploration sequence produced by M is bounded by $z^{O(1)}$ showing (c).

Let A_{reg} be an agent following w_{reg} in G_{reg} and A be an agent following the exploration sequence w produced by M in G. What is left to show is that A traverses G in a closed walk and visits at least $\min\{z, n\}$ distinct vertices. In order to show this, we first establish the following invariants that hold after every iteration i of the while-loop in Algorithm 2.2:

1. If agent A_{reg} is at vertex (v_i, a_i) in G_{reg} after i steps, then after following the exploration sequence output by M up to the end of iteration i agent A is at v_i and $a_i \bmod d_{v_i}$ is the label of the edge to the previous vertex.
2. If (v_i, a_i) is visited by A_{reg} in G_{reg}, then in G both v_i and the neighbor incident to the edge with label $(a_i \bmod d_{v_i})$ are visited by A.

We show the invariants by induction. The starting vertex of A_{reg} in G_{reg} is $(v_0, 0)$ and the starting vertex of A in G is v_0. Note that at the beginning the Turing machine M outputs 0,0 so that in G agent A visits the neighbor of v_0 incident to the edge 0 and then returns to v_0. Thus, both invariants hold before the first iteration of the while-loop.

Now assume that before iteration i both invariants hold. We show that then they also hold after iteration i. If agent A_{reg} is at the vertex (v, a) after $i - 1$ steps and the edge traversed by A_{reg} in step i has label 0, i.e., $l_i = 0$, then A_{reg} moves to vertex $(v, (a + 1) \bmod 3d_v)$ by the definition of G_{reg}, see also Figure 2.1. By assumption, agent A is at vertex v in G and the last edge label is $a \bmod d_v$. Thus, if agent A follows the exploration sequence $1, 0$ output by M in iteration i (line 7 of Algorithm 2.2), then it first traverses the edge labeled $(a + 1) \bmod d_v$ and then returns to v. This

means that after iteration i, the current vertex of A in G is v and the edge label to the previous vertex is $(a+1) \bmod d_v = ((a+1) \bmod 3d_v) \bmod d_v$. Moreover, agent A visited both v and the neighbor of v incident to the edge with label $(a+1) \bmod d_v$. Thus, both invariants hold after iteration i in this case.

The case that $l_i = 1$ is analogous except that edges with label $l_i = 1$ in G_{reg} lead from a vertex (v, a) to a vertex $(v, (a-1) \bmod 3d_v)$. The equivalent movement of A in G is achieved by the sequence $-1, 0$ (line 9 in Algorithm 2.1).

So assume that agent A_{reg} in step i traverses an edge with label $l_i = 2$ from a vertex (v, a) to a vertex (v', a'). This means that there is an edge $\{v, v'\}$ in G with port number $a \bmod d_v$ at v and port number $a' \bmod d_{v'}$ at v'. By assumption, at the beginning of iteration i agent A is at v and $a \bmod d_v$ is the label of the edge to the previous vertex. So if A follows the exploration sequence 0 output in iteration i (line 11 of Algorithm 2.2), then it moves to v'. Now the label to the previous vertex at v' is $a' \bmod d_{v'}$ and A visited both v and v' so that both invariants hold again.

Finally, for property (c) in the lemma, we know that the traversal of agent A_{reg} in G_{reg} is a closed walk by Lemma 2.4 and hence the traversal of A in G also is a closed walk by the first invariant.

What is left to show is that A visits at least $\min\{z, n\}$ distinct vertices in G. If G_{reg} has at most $3z^2$ vertices, then A_{reg} visits all vertices in G_{reg} by assumption and thus A also visits all vertices in G by the second invariant. Otherwise, we know that A_{reg} visits at least $3z^2$ distinct vertices in G_{reg}. Note that this implies $z < n$ as G_{reg} contains at most $3n(n-1)$ vertices.

Assume, for the sake of contradiction, that A visits less than z vertices in G. Let $\bar{V}_{\text{reg}}$ be the set of vertices visited by A_{reg} in G_{reg}. As $|\bar{V}_{\text{reg}}| \geq 3z^2$ by assumption, at least one of the two following cases occurs:

1. The cardinality of $\bar{V} := \{ v \mid (v, j) \in \bar{V}_{\text{reg}} \text{ for some } j \}$ is at least z.
2. There is a vertex $\bar{v}$ in G such that $M_{\bar{v}} := \{ j \mid (\bar{v}, j) \in \bar{V}_{\text{reg}} \}$ has cardinality $\geq 3z$.

We show that both cases lead to a contradiction.

Note that by the second invariant agent A visits all vertices in $\bar{V}$. Thus, if $|\bar{V}| \geq z$, then A visits at least z distinct vertices in G, a contradiction.

Assume the second case occurs and let $\bar{v}$ in G be a vertex such that $|M_{\bar{v}}| \geq 3z$. Then we have $|\{j \mod d_{\bar{v}} \mid j \in M_{\bar{v}}| \geq z$ implying that agent A visits at least z neighbors of $\bar{v}$ in G by the second invariant. This again is a contradiction. □

To make the results above usable for our agents with pebbles, we need more structure regarding the memory usage of the agent. To this end, we formally define a walking Turing machine with access to pebbles which we will refer to as a **pebble machine**. Formally, we can view such a walking Turing machine as a specification of the general agent model with pebbles described in Section 2.1.2, where the states of the agent correspond the state of the working tape, the position of the head, and the state of the Turing machine.

Definition 2.6. *Let $s, p, m \in \mathbb{N}$. An* ***(s, p, m)-pebble machine*** *$T = (Q, \bar{Q}, P, m, \delta_{in}, \delta_{TM}, \delta_{out}, q^*)$ is an agent $A = (\Sigma, \bar{\Sigma}, \delta, \sigma^*)$ with a set $P = \{1, \dots, p\}$ of p pebbles and the following properties:*

(a) The set of states is $\Sigma = Q \times \{0,1\}^m \times \{0, \dots, m-1\}$, where each state consists of a Turing state, the state of the working tape of length m, and a head position on the tape.

(b) In the initial state σ^* the Turing state is q^*, the head position is 0, and the tape has 0 at every position.

(c) The agent's transition function $\delta \colon \Sigma \times \mathbb{N} \times \mathbb{N} \times 2^P \times 2^P \to \Sigma \times (\mathbb{N} \cup \{\perp\}) \times 2^P \times 2^P$ is computed as follows:

(i) The agent first observes its local environment according to the function $\delta_{in} \colon Q \times \mathbb{N} \times \mathbb{N} \times 2^P \times 2^P \to Q$ that maps a vector (q, d_v, l, P_A, P_v) consisting of the current Turing state, the degree d_v of the current vertex, the label l of the edge leading back to the vertex last visited, the set P_A of carried pebbles and the set P_v of pebbles located at the current vertex to a new Turing state q'.

(ii) The agent does computations on the working tape like a regular Turing machine according to the function $\delta_{TM} \colon Q \times \{0,1\} \to Q \times \{0,1\} \times \{\text{left}, \text{right}\}$ that maps the tuple consisting of the current Turing state and the symbol at the current head position (q, a) to a tuple (q', a', d) meaning that the machine transitions to the new state q', writes a' at the current position of the head and moves the head in direction d; this process is repeated until a halting state $\bar{q} \in \bar{Q}$ is reached (note that we only consider Turing machines that eventually halt).

(iii) It performs actions according to the function $\delta_{out} \colon \bar{Q} \times 2^P \times 2^P \times \mathbb{N} \times \mathbb{N} \to 2^P \times 2^P \times \mathbb{N}$ that maps a tuple (q, P_A, P_v) containing the current Turing state q, the set of carried pebbles P_A and the set of pebbles P_v at the current vertex v to a tuple (P'_A, P'_v, l') meaning that it drops and retrieves pebbles such that it carries P'_A, leaves P'_v at v and takes the edge locally labeled by l'.

When considering a pebble machine $T = (Q, \bar{Q}, P, m, \delta_{\text{in}}, \delta_{\text{TM}}, \delta_{\text{out}})$ we will call the Turing states Q simply **states** and we will call the set of states Σ of the underlying agent model **configurations**. As the configuration of a pebble machine is fully described by the (Turing) state $q \in Q_T$, the head position, and the state of the working tape, it has $sm2^m$ configurations. We further call a transition of the agent according to the transition function δ_{TM} a **computation step**. Note that an agent remains at the same vertex and only changes its configuration when performing a computation step.

In the following theorem, we explain how to place pebbles on a closed walk and use them as additional memory.

Theorem 2.7. *There are constants $c, c' \in \mathbb{N}$, such that for every $(s, p, 2m)$-pebble machine T there exists a $(cs, p + c, m)$-pebble machine T' with the following properties:*

(a) For every graph G with $n < 2^{m/c'}$ vertices, the pebble machine T' explores G in a closed walk, collects all pebbles, returns to the starting vertex and terminates. The overall number of edge traversals and computation steps needed by the pebble machine T' is bounded by $2^{O(m)}$.

(b) *For every graph G with $n \geq 2^{m/c'}$ vertices, T' reproduces the walk of T in G while the positions of p of the $p+c$ pebbles correspond to the positions of the p pebbles of T. For the initialization, T' needs $2^{O(m)}$ edge traversals and computations steps. Afterwards, the number of edge traversals and computation steps needed by the pebble machine T' to reproduce one edge traversal or computation step of T is bounded by $2^{O(m)}$.*

Proof. The general idea of the proof is that T' places the constant number of additional pebbles on a closed walk ω in order to encode the tape content of the pebble machine T. Using these pebbles, T' can also count the number of distinct vertices on the closed walk ω. If the closed walk is too short, then T' already explored the graph and the condition for (a) is satisfied. Otherwise, the closed walk is long enough to allow for a sufficient number of distinct positions of the pebble and we are in part (b) of the statement of the theorem.

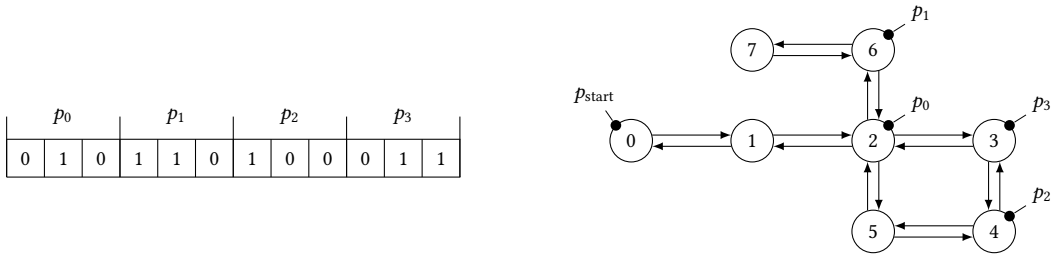
Let Q be the set of states of T. We define the set of states of T' to be $Q \times Q'$ for a set Q', i.e., every state of T' is a tuple (q, q'), where q corresponds to the state of T in the current step of the traversal. The pebble machine T' observes the input according to δ_{in} and performs actions according to δ_{out} just as T, while only changing the first component of the current state. T' uses p pebbles in the same way as T and possesses a set $\{p_{\text{start}}, p_{\text{temp}}, p_{\text{next}}, p_0, p_1, \ldots, p_{c-4}\}$ of additional pebbles. The pebble p_{start} is dropped by T' right after observing the input according to δ_{in} in order to mark the current location of T during the traversal. The purpose of the pebbles p_{temp} and p_{next} will be explained later. The other pebbles $\{p_0, p_1, \ldots, p_{c-4}\}$ are placed along a closed walk ω to simulate the memory of T, while the states Q' and the tape of T' are used to manage this memory.

To this end, we divide the tape of T' into a constant number c_0 of blocks of size m/c_0 each. In the course of the proof, we will introduce a constant number of variables to manage the simulation of the memory of T with pebbles. Each of these variables is stored in a constant number of blocks. The constant c_0 is chosen large enough to accommodate all variables on the tape of T'. By Lemma 2.5, there is a constant c_1 such that for any $r \in \mathbb{N}$ there is a Turing machine M with at most c_1 states and a tape of length $c_1 \cdot r$ outputting an exploration sequence that gives a closed walk of length at most $2^{c_1 \cdot r}$ visiting at least $\min\{2^r, n\}$ vertices in any graph with n vertices. Let $m_1 := m/(c_0 c_1)$ and let $m_0 \in \mathbb{N}$ be such that for all $m' \in \mathbb{N}$ with $m' \geq m_0$ we have $c_1 \leq 2^{m'/c_0}$ and $2^{m'/c_0} > 2m'$.

In the following, we show how the simulated memory is managed by providing algorithms in pseudocode (see **??** 2.3–2.8). These can be implemented on a Turing machine with a constant number of states c_{Alg}. Let $c = \max\{2^{2m_0}, 2c_0c_1 + 3, c_{\text{Alg}}\}$ and $c' := c_0c_1$. Note that c only depends on the constants c_0, c_1 and c_{Alg}, but not on m or p. It is without loss of generality to assume $m \geq m_0$, because, for $m < m_0$, we can store the configuration of the tape of T in the states Q' of T', since $c \geq 2^{2m_0}$.

We proceed to show that the computations on the tape of length $2m$ performed by T according to the transition function δ_{TM} can be simulated using the pebbles $\{p_{\text{start}}, p_{\text{temp}}, p_0, p_1, \ldots, p_{c-4}\}$. The proof of this result proceeds along the following key claims.

1. We can find a closed walk ω containing 2^{m_1} distinct vertices so that $c-3$ pebbles placed along



(a) Tape memory *(b) Memory encoded by pebbles*

Figure 2.2: Memory encoding by pebbles on a closed walk. The state of the tape of length $2m = 12$ in (a) is encoded by the position of the $c - 3 = 4$ pebbles in (b). The number of the vertices corresponds to the order of first traversal by the closed walk ω starting in 0. The position of each pebble encodes $m_1 = 3$ bits.

this walk can encode all configurations of the tape of T.

2. We can move along ω while keeping track of the number of steps and counting the number of distinct vertices until we have seen 2^{m_1} distinct vertices.
3. We can read from and write to the memory encoded by the placement of the pebbles along ω.
4. If the closed walk ω starts at vertex v and T moves from vertex v to vertex v', we can move all pebbles to a closed walk ω' starting in v' while preserving the content of the memory.

1. Finding a closed walk ω. Lemma 2.5 yields a Turing machine M_{walk} with c_1 states and a tape of length m/c_0 that produces an exploration sequence corresponding to a closed walk ω that contains at least $\min\{n, 2^{m_1}\}$ distinct vertices and has length at most $2^{c_1 m_1} = 2^{m/c_0}$. We use a variable R_{walk} of size m/c_0 for the memory of M_{walk}, which is initially assumed to have all bits set to 0. If $2^{m_1} > n$, then the exploration sequence produced by M_{walk} is a walk exploring G. Note that by definition we have $m/c' = m_1$. So this happens exactly when the condition for (a) in the theorem is satisfied. Below we will show how to count the number of unique vertices on the closed walk of M_{walk}. Hence, the pebble machine T' can initially walk along the closed walk ω counting the number of distinct vertices. If this number is smaller than 2^{m_1}, we know that we have visited all vertices of G so that we can collect all pebbles and return to the pebble p_{start}, which has not been moved and therefore marks the starting vertex of T. We show at the end of the proof that this takes at most $2^{O(m)}$ edge traversals and computation steps.

From now on, we can therefore assume that ω contains at least 2^{m_1} distinct vertices. We need to show that $c - 3$ pebbles placed along the walk ω can be used to encode all of the 2^{2m} configurations of the tape of T. Figure 2.2 shows how each pebble encodes a certain part of the tape of T. The idea is that each pebble can be placed on one of 2^{m_1} different vertices, thus encoding exactly m_1 bits. We divide the tape of length $2m$ into $2m/m_1 = 2c_0c_1$ parts of size m_1 each, such that the position of pebble p_i encodes the bits $\{im_1, \ldots, (i+1)m_1 - 1\}$, where we assume the bits of the tape of T to be numbered $0, 1 \ldots, 2m - 1$. As $c \geq 2c_0c_1 + 3$, we have enough pebbles to encode the configuration of

Algorithms 2.3: Auxiliary functions for moving along the closed walk ω.

```
1  function STEP()
2      traverse edge according to value of exploration sequence output by M_walk
3      R_steps ← R_steps + 1
4  function FINDPEBBLE(p_i)
5      while not OBSERVE(p_i) do
6          STEP()
7  function RESTART()
8      FINDPEBBLE(p_start)
9      R_steps ← 0
10     R_id ← 0
11     R_walk ← 0
```

the tape of T.

2. Navigating ω. Let R_{steps} be a variable counting the number of steps along ω and R_{id} be a variable for counting the number of unique vertices visited along ω after starting in the vertex marked by p_{start}. Note that R_{id} gives a way of associating a unique identifier to the first 2^{m_1} distinct vertices along ω. As $m_1 \leq m/c_0$ holds, m/c_0 tape cells suffice for counting the first 2^{m_1} distinct vertices along ω. The overall number of steps along the closed walk is bounded by $2^{m/c_0}$ and therefore m/c_0 tape cells also suffice for counting the steps along ω.

It remains to show that we can move along the closed walk ω while updating R_{steps} and R_{id}, such that, starting from the vertex marked by p_{start}, the variable R_{steps} contains the number of steps taken and R_{id} contains the number of distinct vertices visited. Let DROP(p_i) denote the operation of dropping pebble p_i at the current vertex, PICKUP(p_i) the operation of picking up p_i from the current vertex if possible, and let OBSERVE(p_i) be "true" if and only if pebble p_i is located at the current vertex. Consider the auxiliary functions shown in Algorithms 2.3. The function STEP() moves one step along ω and updates R_{steps} accordingly. The function FINDPEBBLE(p_i) moves along ω until it finds pebble p_i. The function RESTART() goes back to the starting vertex marked by p_{start}, sets both variables R_{steps} and R_{id} to 0, and restarts M_{walk} by setting the variable R_{walk} to 0. Finally, the function NEXTDISTINCTVERTEX() in Algorithm 2.4 does the following: If the number of distinct vertices visited is already 2^{m_1}, then we go back to the start. Otherwise, we continue along ω until we encounter a vertex we have not visited before. We repeatedly traverse an edge, drop the pebble p_{temp}, store the number of steps until reaching that vertex, then we restart from the beginning and check if we can reach that vertex with fewer steps. If not, we found a new distinct vertex. Note that we use the auxiliary variables R'_{steps} and R'_{walk}, which both need a constant number of blocks of size m_0/c_0.

3. Reading from and writing to simulated memory. We show how to simulate the changes to the tape of T by changing the positions of the pebbles along ω. The transition function δ_{TM} of T

Algorithm 2.4: Moving along the closed walk ω while updating R_{steps} and R_{id}.

Input: local environment observed by pebble machine T'

```
1  function nextDistinctVertex()
2      if R_id ≡ 2^{m_1} − 1 then
3          restart()
4          return
5      R_id ← R_id + 1
6      R'_steps ← R_steps
7      repeat
8          step()
9          R'_steps ← R'_steps + 1
10         drop(p_temp)
11         R'_walk ← R_walk
12         restart()
13         findPebble(p_temp)
14         pickup(p_temp)
15         R_walk ← R'_walk
16     until R_steps ≡ R'_steps
```

determines how T does computations on its tape and, in particular, how T changes its head position. We use a variable R_{head} of size m/c_0 to store the head position. By assumption, $m \geq m_0$ and therefore $2^{m/c_0} > 2m$, i.e., the size of R_{head} is sufficient to store the head position. In order to simulate one transition of T according to δ_{TM}, we need to read the bit at the current head position and then write to the simulated memory and change the head position accordingly. Reading from the simulated memory is done by the function readBit() in Algorithm 2.7 and writing of a bit b to the simulated memory is performed by the function writeBit(b) in Algorithm 2.8.

First, let us consider the two auxiliary functions getPebbleId(p_i) and putPebbleAtId(p_i, id) (cf. Algorithms 2.5 and 2.6). As the name suggests, the function getPebbleId(p_i) returns the unique identifier associated to the vertex marked by p_i. Recall that vertices are indistinguishable. Here, unique identifier refers to the number of distinct vertices on the walk ω before reaching the vertex marked with p_i for the first time. Given an identifier id, we can use the function putPebbleAtId(p_i,id) for placing pebble p_i at the unique vertex corresponding to id. By the choice of our encoding, if $R_{\text{head}} = i \cdot m_1 + j$ with $j \in \{0, \ldots, m_1 - 1\}$, then the j-bit of the binary encoding of the position of pebble p_i encodes the contents of the tape cell specified by R_{head}. Thus, for reading from the simulated memory, we have to compute i and j and determine the position of the corresponding pebble in the function readBit(). For the function writeBit(b), we also compute i and j. Then, we move the pebble p_i by 2^j unique vertices forward if the bit flips to 1 or by 2^j unique vertices backward if the bit flips to 0.

Algorithm 2.5: Reading position of a pebble on the closed walk ω.

Input: pebble p_i

Output: identifier id $\in \{0, \dots, 2^{m_1} - 1\}$ corresponding to position of pebble p_i

```
1 function getPebbleId(p_i)
2     restart()
3     while not observe(p_i) do
4         nextDistinctVertex()
5     return R_id
```

Algorithm 2.6: Putting a pebble at a specific position on the closed walk ω.

Input: pebble p_i, identifier id $\in \{0, \dots, 2^{m_1} - 1\}$

```
1 function putPebbleAtId(p_i, id)
2     findPebble(p_i)
3     pickup(p_i)
4     restart()
5     while id>0 do
6         id ← id − 1
7         nextDistinctVertex()
8     drop(p_i)
```

4. Relocating ω. When T moves from a vertex v to another vertex v', the walk ω and the pebbles on it need to be relocated. Recall that T' marked the current vertex v with the pebble p_{start}. After having computed the label of the edge to v', T' drops the pebble p_{next} at v'. Then T' moves the pebbles placed along the walk ω to the corresponding positions along a new walk ω' starting at v' in the following way. We iterate over all $c-3$ pebbles and for each pebble p_i we start in v, determine the identifier id of the vertex marked by p_i via getPebbleId(p_i), pick up p_i, move to p_{next} and place p_i on ω' using the function putPebbleAtId(p_i, id). In this call of putPebbleAtId(p_i, id) all occurrences of p_{start} are replaced by p_{next}. This way, we can carry the memory simulated by the pebbles along during the graph traversal.

Thus, we have shown the first part of (a) and (b), i.e., T' either explores G or it can simulate the traversal of T in G while using a tape with half the length, but c additional pebbles and a factor of c additional states.

Finally, we bound the number of edge traversals and computation steps in both (a) and (b). First, we bound the number of edge traversals that T' needs for simulating one computation step of T. Recall that T' needs at most $2^{m/c_0} \leq 2^m$ edge traversals for moving once along the whole closed walk ω. A call of the function step() corresponds to one edge traversal, a call of findPebble(p_i) thus corresponds to at most 2^m edge traversals and also a call of restart() corresponds

Algorithm 2.7: Reading the bit at the current head position from the simulated memory.

Output: bit $b \in \{0,1\}$ at current head position of the simulated memory

```
1 function readBit()
2     i ← ⌊R_head / m_1⌋
3     j ← R_head − m_1 · i
4     id ← getPebbleId(p_i)
5     return j-th bit of id (in binary)
```

Algorithm 2.8: Writing the bit b to the simulated memory at the current head position.

Input: bit $b \in \{0,1\}$ to be written to simulated memory at current head position

```
1 function writeBit(b)
2     i ← ⌊R_head / m_1⌋
3     j ← R_head − m_1 · i
4     id ← getPebbleId(p_i)
5     if b ≡ 1 and readBit() ≡ 0 then
6         id ← id + 2^j
7     else if b ≡ 0 and readBit() ≡ 1 then
8         id ← id − 2^j
9     putPebbleAtId(p_i,id)
```

to at most 2^m edge traversals. Moreover, one iteration of the loop in nextDistinctVertex() accounts for at most 2^m edge traversals and therefore executing the whole function results in at most $2^m \cdot 2^m = 2^{2m}$ edge traversals. This means that one call of getPebbleId(p_i) or putPebbleAtId(p_i,id) incur at most $2^{O(m)}$ edge traversals and this also holds for readBit() and writeBit(b). Hence, for every computation step performed by T according to δ_{TM}, the pebble machine T' performs actions according to readBit() and writeBit(b) and overall does at most $2^{O(m)}$ edge traversals. The above argument also shows that at most $2^{O(m)}$ edge traversals are necessary to count the number of distinct vertices on the closed walk ω at the beginning.

Next, let us bound the number of edge traversals that T' needs for reproducing one edge traversal of T. This means that we need to count how many edge traversals are necessary to relocate all pebbles placed along the walk ω to the new walk ω'. For every pebble p_i, we call getPebbleId(p_i) which results in at most 2^m edge traversals, we pick up p_i and move to p_{next} which again needs at most 2^m edge traversals, and place p_i on ω' using the function putPebbleAtId(p_i, id) which also needs 2^m edge traversals. Overall, this procedure is done for a constant number of pebbles and hence requires at most $2^{O(m)}$ edge traversals.

Next we bound the number of computation steps of T' by using the bounds on the number of edge traversals. Recall that the state of T' is a tuple (q, q'), where q corresponds to the state of T.

In the computation only the second component of the state of T' changes and therefore there are only at most c possible states. The tape length and number of possible head positions of the Turing machine is m. Since we may assume without loss of generality that $m \geq 2$, we can bound the number of distinct configurations of T' in each computation by $2^{O(m)}$. Hence, after every edge traversal T' does at most $2^{O(m)}$ computation steps. This implies that in part (a) of the theorem, the number of computation steps is bounded by $2^{O(m)}$ because the number of edge traversals is bounded by $2^{O(m)}$ as shown above. Similarly, in part (b) of the theorem the total number of computation steps after $2^{O(m)}$ edge traversals is bounded by $2^{O(m)}$. Since $m \geq 2$ this means that also the sum of computation steps and edge traversals can be bounded by $2^{O(m)}$ both for one computation step and one edge traversal of T. □

Finally, we show that by recursively simulating a pebble machine by another pebble machine with half the memory but a constant number of additional pebbles we can explore any graph with at most n vertices while using $O(\log \log n)$ pebbles and only $O(\log \log n)$ bits of memory.

Theorem 2.8. *Any connected undirected graph on at most n vertices can be explored by an agent in a polynomial number of steps using $O(\log \log n)$ pebbles and $O(\log \log n)$ bits of memory. The agent does not require n as input and terminates at the starting vertex with all pebbles after exploring the graph.*

Proof. Let $c, c' \in \mathbb{N}$ be the constants from Theorem 2.7. Let $r \in \mathbb{N}$ be arbitrary and consider a $(c, 0, c'2^{r+1})$-pebble machine $T^{(r)}$ that simply terminates without making a computation step or edge traversal. Applying Theorem 2.7 for the pebble machine $T^{(r)}$ gives a $(c^2, c, c'2^r)$-pebble machine $T_r^{(r)}$ with the following properties. If $n < 2^{2^r}$, then $T_r^{(r)}$ explores the graph and returns to the starting vertex. If, on the other hand, $n \geq 2^{2^r}$, then $T_r^{(r)}$ reproduces the walk of $T^{(r)}$ (which in this case is of course trivial). Note that these properties hold even though the number n of vertices is unknown and, in particular, not given as input to $T_r^{(r)}$.

Applying Theorem 2.7 iteratively, we obtain a $(c^{r+2-i}, (r+1-i)c, c'2^i)$-pebble machine $T_i^{(r)}$ for all $i \in \{0, \dots, r-1\}$ that reproduces the walk of $T_{i+1}^{(r)}$ or it already explores the given graph and returns to the start vertex. For a graph G with $n < 2^{2^r}$, $T_r^{(r)}$ explores G and returns to the start with all pebbles and terminates. Thus for such a graph G it does not matter which case occurs when applying Theorem 2.7, as in both cases we can conclude that $T_i^{(r)}$ for $i \in \{0, \dots, r-1\}$ explores the graph, returns with all pebbles to the start vertex and terminates.

If we have $n \geq 2^{2^r}$, then $n \geq 2^{2^i}$ holds for all $i \in \{0, \dots, r-1\}$ and in particular $T_0^{(r)}$ reproduces the walk of $T^{(r)}$ in G, i.e., remains at the starting vertex and terminates.

The desired pebble machine T exploring any graph G with $O(\log \log n)$ pebbles and $O(\log \log n)$ bits of memory works as follows: We have a counter r, which is initially 1 and is increased by one after each iteration until the given graph G is explored. In iteration r, pebble machine T does the same as the $(c^{r+2}, (r+1)c, c')$-pebble machine $T_0^{(r)}$ until it terminates. The pebble machine T terminates as soon as for some $r \in \mathbb{N}$ the pebble machine $T_0^{(r)}$ recognizes that it explored the whole graph. This happens when $r = \lceil \log \log n \rceil + 1$. Hence, T uses at most $O(\log \log n)$ pebbles.

Concerning the memory requirement of T, note that T needs to store the state of $T_0^{(r)}$, the tape content of $T_0^{(r)}$ and the current value of r. There are c^{r+2} states of the pebble machine $T_0^{(r)}$, its tape length is c' and $r \leq \lceil \log \log n \rceil + 1$ in every iteration, so that T can be implemented with $O(\log \log n)$ bits of memory.

It is left to show is that the number of edge traversals of T in the exploration of a given graph G with n vertices is polynomial in n. To this end, we first show that the number of edge traversals of the pebble machine $T_0^{(r)}$ is bounded by $n^{O(1)}$ for all $r \in \{1, \ldots, \lceil \log \log n \rceil + 1\}$. Let $r \in \{1, \ldots, \lceil \log \log n \rceil + 1\}$ be arbitrary and let t_i denote the sum of the number of edge traversals and computation steps of $T_i^{(r)}$ in the given graph G. The pebble machine $T_r^{(r)}$ has a tape of length of $m = c'2^r$. Applying Theorem 2.7, we get that either $T_r^{(r)}$ explores G and uses at most $2^{O(m)}$ edge traversals and computation steps or $T_r^{(r)}$ simulates the walk of a pebble machine that does not make a single edge transition and uses at most $2^{O(m)}$ edges traversals and computation steps. In both cases, we obtain

$$t_r \leq 2^{O(2^r)} \leq 2^{O(2^{\log \log n})} = 2^{O(\log n)} = n^{O(1)}.$$

This shows the desired bound for t_r. Furthermore, one computation step or one edge traversal of $T_i^{(r)}$ leads to at most $2^{O(c' \cdot 2^i)} = 2^{O(1)2^i}$ edge traversals and computation steps of $T_{i-1}^{(r)}$ by Theorem 2.7. Hence, we obtain

$$t_{i-1} \leq 2^{O(1)2^i} t_i \qquad \forall\, i \in \{1, \ldots, \lceil \log \log n \rceil + 1\}. \tag{2.3}$$

By iterative application of (2.3), we obtain

$$t_0 \leq 2^{O(1)2^i} t_1 \leq \ldots \leq 2^{O(1)\sum_{i=1}^{\lceil \log \log n \rceil + 1} 2^i} \cdot t_{\lceil \log \log n \rceil + 1} \leq 2^{O(1)2^{\lceil \log \log n \rceil}} \cdot n^{O(1)} \leq n^{O(1)}.$$

Thus, the number of edge traversals t_0 of $T_0^{(r)}$ is polynomial in n. As T performs at most $n^{O(1)}$ edge traversals according to $T_0^{(r)}$ for at most $\lceil \log \log n \rceil + 1$ distinct values of r, the overall number of edge traversals of T is also bounded by $n^{O(1)}$. □

Since an additional pebble is more powerful than a bit of memory (Lemma 2.1), we obtain the following direct corollary of Theorem 2.8.

Corollary 2.9. *Any connected undirected graph on at most n vertices can be explored by an agent in a polynomial number of steps using $O(\log \log n)$ pebbles and constant memory. The agent does not require n as input and terminates at the starting vertex with all pebbles after exploring the graph.*

Since an additional agent is more powerful than a pebble (Lemma 2.2), we obtain the following direct corollary of Theorem 2.8 and Corollary 2.9.

Corollary 2.10. *Any connected undirected graph on at most n vertices can be explored in polynomial time by a set of $O(\log \log n)$ agents with constant memory each. The agents do not require n as input and terminate at the starting vertex after exploring the graph.*

Remark 2.11 The agent in Theorem 2.8 requires $O(\log \log n)$ bits of memory and the agents in Corollary 2.9 and Corollary 2.10 only $O(1)$ bits of memory. An interesting question is how much memory is necessary to fully encode the transition function

$$\delta : \Sigma \times \mathbb{N} \times \mathbb{N} \times 2^P \times 2^P \to \Sigma \times (\mathbb{N} \cup \{\perp\}) \times 2^P \times 2^P,$$

of an agent (see Section 2.1.2). Naively encoding it as a table with a row for every possible state, vertex degree, previous edge label and possible combination of $O(\log \log n)$ pebbles/agents at the current vertex takes $(\log n)^{O(1)}$ bits of memory.

However, we can obtain a much more compact encoding by exploiting the specific structure of our algorithm: First of all, we never explicitly use the degree of the current vertex. Moreover, the Turing machine from Lemma 2.5 that we internally use produces an exploration sequence of the form $\{-1, 0, 1\}^*$. This means that our transition function can be expressed more concisely if we would allow in our model to specify transitions relative to the label of the previous edge.

Furthermore, our algorithm only interacts with a constant number of pebbles in every level of the recursion (cf. Theorem 2.7). We can express the state of T in the proof of Theorem 2.8 as a vector, where each component encodes the state in a different level of the recursion. In every transition, only two consecutive entries of this vector can change, as one level of recursion only interacts with the level of recursion below to access the simulated memory.

Since there are only a constant number of states per recursive level, and only a constant number of pebbles involved, all transitions regarding two consecutive levels can be encoded in constant memory. If we therefore explicitly encode all $O(\log \log n)$ levels of recursion and additionally allow to only give the edge label offset in the transition function, the entire transition function can be encoded with $O(\log \log n)$ bits of memory.

2.3 Lower Bounds

In this section, we present a general lower bound relating the memory requirement and number of collaborating agent needed for collaborative exploration. Specifically, we show that for a set of cooperative agents with sublogarithmic memory of $O((\log n)^{1-\varepsilon})$ for some constant $\varepsilon > 0$, $\Omega(\log \log n)$ agents are needed to explore any undirected graph with n vertices. In light of our reduction presented in Section 2.1.4, this implies that an agent with sublogarithmic memory needs $\Omega(\log \log n)$ pebbles to explore any n-vertex graph.

To prove the lower bound, we use the concept of an r-barrier introduced in Definition 2.12. Informally, an r-barrier is a graph with two special entry points such that any subset of up to r agents with s states cannot reach one entry point when starting from the other. Moreover, a set of $r+1$ agents can explore an r-barrier, but the agents can only leave the barrier via the same entry point. We construct an r-barrier by replacing every edge of a graph G by a $(r-1)$-barrier. The resulting graph has the property that a set of r agents traversing this graph needs to stay close to each other to be able to traverse the barriers and make progress, as shown in Lemma 2.18. However, if the agents

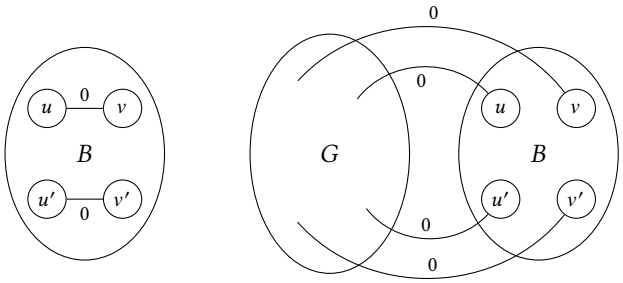


Figure 2.3: The r-barrier B on the left with two distinguished edges $\{u,v\}$, $\{u',v'\}$ can be connected to an arbitrary graph G, as shown on the right.

stay close to each other, the states and relative positions of the agents repeat and their behaviour becomes periodic. This property is formally expressed in Lemma Lemma 2.19. In Theorem 2.20, we then show how to use these two key properties in order to construct an r-barrier for a set of k agents given an $(r-1)$-barrier.

By carefully bounding the size of the r-barriers in our recursive construction via Lemma 2.22, we obtain a trap of size $O(s^{2^{5k}})$ for any given set of k agents with at most s states each (Theorem 2.24). In Theorem 2.25, we show that the size of the trap directly implies that the number of agents with at most $O((\log n)^{1-\varepsilon})$ bits of memory needed for exploring any graph of size n is at least $\Omega(\log\log n)$.

The graphs involved in our construction are 3-regular and allow a labeling such that the two port numbers at both endpoints of any edge coincide. We therefore speak of the label of an edge and assume the set of labels to be $\{0,1,2\}$.

The most important building block for our construction are **barriers**. Intuitively, a barrier is a subgraph that cannot be crossed by a subset of the given set of agents. To define barriers formally, we need to describe how to connect two 3-regular graphs. Let B be a 3-regular graph with two distinguished edges $\{u,v\}$ and $\{u',v'\}$ both labeled 0, as shown in Figure 2.3. An arbitrary 3-regular graph G with at least two edges labeled 0 can be connected to B as follows: We remove the edges $\{u,v\}$ and $\{u',v'\}$ from B and two edges labeled 0 from G. We then connect each vertex of degree 2 in G with a vertex of degree 2 in B via an edge labeled 0.

Definition 2.12. *For $1 \le r \le k$, the graph B is an **r-barrier** for a set of k s-state agents $\mathcal{A}$ if for all graphs G connected to B as above, the following two properties hold:*

(a) For all subsets of agents $\mathcal{A}' \subseteq \mathcal{A}$ with $|\mathcal{A}'| \le r$ and every pair (a,b) in $\{u,v\} \times \{u',v'\}$ the following holds: If initially all agents $\mathcal{A}$ are at vertices of G, then no agent in the set $\mathcal{A}'$ can traverse B from a to b or vice versa when only agents in $\mathcal{A}'$ enter the subgraph B at any time during the traversal. We equivalently say that no subset of r agents can traverse B from a to b or vice versa.

(b) Whenever a subset of agents $\mathcal{A}' \subseteq \mathcal{A}$ with $|\mathcal{A}'| = r+1$ enters the subgraph B during the traversal, all agents in $\mathcal{A}'$ leave B either via u and v or via u' and v' if no other agents visit B during this

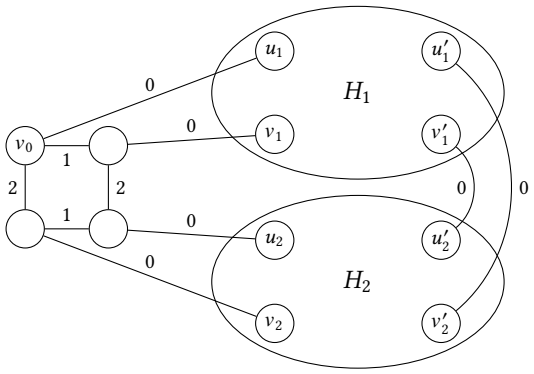


Figure 2.4: Constructing a trap given two k-barriers H_1 and H_2.

traversal. In other words, the set of agents $\mathcal{A}'$ cannot split up such that a part of the agents leaves B via u or v and the other part via u' or v'.

A k-barrier immediately yields a trap for a set of agents.

Lemma 2.13. *Given a k-barrier with n vertices for a set of k agents $\mathcal{A}$, we can construct a trap with $2n + 4$ vertices for $\mathcal{A}$.*

Proof. Let H_1 and H_2 be two copies of a k-barrier for the set of agents $\mathcal{A}$ with distinguished edges $\{u_i, v_i\}$, $\{u'_i, v'_i\}$ of H_i. We connect the two graphs and four additional vertices, as shown in Figure 2.4. If the agents start in the vertex v_0, then none of the agents can reach u'_1 or v'_1 via the k-barrier H_1 or via the k-barrier H_2. Thus the agents $\mathcal{A}$ do not explore the graph. The constructed trap for the set of agents $\mathcal{A}$ contains $2n + 4$ vertices. □

Our goal for the remainder of the section is to construct a k-barrier for a given set of k agents $\mathcal{A}$ and to give a good upper bound on the number of vertices it contains. This will give an upper bound on the number of vertices of a trap by Lemma 2.13. The construction of the k-barrier is recursive. We start with a 1-barrier which builds on the following useful result by Fraigniaud et al. [Fra+06b] stating that, for any set of non-cooperative agents, there is a graph containing an edge which is not traversed by any of them. A set of agents is **non-cooperative** if the transition function δ_i of every agent A_i is completely independent of the state and location of the other agents, i.e., δ_i is independent of $\boldsymbol{\sigma}_{-i}$, see Section 2.1.3.

Theorem 2.14 ([Fra+06b, Theorem 4])**.** *For any k non-cooperative s-state agents, there exists a 3-regular graph G on $O(ks)$ vertices with the following property: There are two edges $\{v_1, v_2\}$ and $\{v_3, v_4\}$ in G, the former labeled 0, such that none of the k agents traverses the edge $\{v_3, v_4\}$ when starting in v_1 or v_2.*

We proceed to generalize this construction towards arbitrary starting states and collaborating agents.

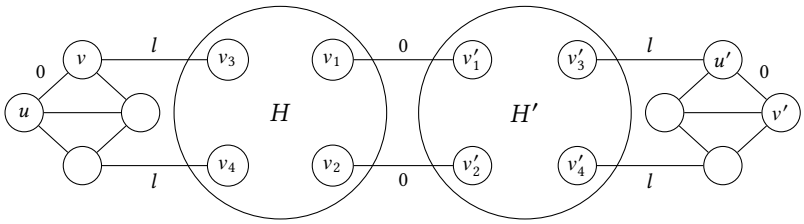


Figure 2.5: A 1-barrier B for $\mathcal{A}$ for the case that $l \in \{1, 2\}$.

Lemma 2.15. *For every set of k collaborating s-state agents $\mathcal{A}$, there exists a 1-barrier B with $O(ks^2)$ vertices. Moreover, B remains a 1-barrier even if for all $i \in \{1, \dots, k\}$ agent A_i starts in an arbitrary state $\sigma \in \Sigma_i$ instead of the starting state σ_i^*.*

Proof. Let $\mathcal{A} = \{A_1, \dots, A_k\}$, let Σ_i be the set of states of A_i and let σ_i^* be its starting state. For all $i \in \{1, \dots, k\}$ and all $\sigma \in \Sigma_i$, we define agent $A_i^{(\sigma)}$ to be the agent with the same behavior as A_i, but starting in state σ instead of σ_i^*. That is, $A_i^{(\sigma)}$ has the same set of states Σ_i as A_i and it transitions according to the function δ_i of A_i. Moreover, let $S := \{A_i^{(\sigma)} \mid i \in \{1, \dots, k\},\ \sigma \in \Sigma_i\}$.

Applying Theorem 2.14 for the set of agents S yields a graph H with an edge $\{v_1, v_2\}$ labeled 0 and an edge $\{v_3, v_4\}$ labeled $l \in \{0, 1, 2\}$ so that any agent $A_i^{(\sigma)}$ that starts in v_1 or v_2 does not traverse the edge $\{v_3, v_4\}$. Let B be the graph consisting of two connected copies of H and 8 additional vertices, as illustrated in Figure 2.5. The edges $\{v_1, v_2\}$ and $\{v_1', v_2'\}$ are replaced by $\{v_1, v_1'\}$ and $\{v_2, v_2'\}$, which are also labeled 0. The edges $\{v_3, v_4\}$ and $\{v_3', v_4'\}$ with label l are deleted and v_3 and v_4 are connected each to one of the two two-degree vertices of a diamond graph by an edge with label l. The same connection to a diamond graph is added for v_3' and v_4' as shown in Figure 2.5. The edge labels of the two diamond graphs are arbitrary. Since each diamond graph has two vertices of degree three, each diamond graph has at least one edge with label 0. We choose one edge with label 0 and call the end vertices u and v (resp. u', v'). Note that in Figure 2.5 we have $l \in \{1, 2\}$; for the case that $l = 0$ the edge $\{u, v\}$ is the unique edge between the two vertices that are not adjacent to v_3 or v_4.

We claim that B is a 1-barrier for $\mathcal{A}$ with the distinguished edges $\{u, v\}$ and $\{u', v'\}$. Assume for the sake of contradiction, that property (a) of the 1-barrier does not hold, i.e., there is a graph G that can be connected to B via the pairs of vertices $\{u, v\}$ and $\{u', v'\}$ so that if the agents $\mathcal{A}$ start in G in an arbitrary state, there is an agent A_j that walks (without loss of generality) from u to u' in B while there are no other agents in B. Then A_j in particular walks from v_1' or v_2' to v_3' or v_4' in H' and starts this walk in a state $\sigma \in \Sigma_j$. But the traversal sequence of A_j in H' is the same as that of $A_j^{(\sigma)}$ that starts at v_1' or v_2'. This would imply that $A_i^{(\sigma)}$ traverses the edge $\{v_3, v_4\}$ in the original graph H when starting in v_1 or v_2, which contradicts Theorem 2.14.

To prove property (b) of a 1-barrier, assume that there is a set of two agents, such that both enter B during the traversal and one of them exits B via u or v and the other via u' or v'. But then again one of the agents must have traversed H starting in v_1 or v_2 in a state σ and finally traversed the edge with label l incident to v_3 or v_4 or similarly in H' with v_1', v_2', v_3', v_4'. This leads to the same

contradiction as above.

The whole proof does not use the specific starting states of the agents $\mathcal{A}$ and, in particular, the definition of S is independent of the starting states of the agents. Consequently, B is a 1-barrier for $\mathcal{A}$ even if we change the starting states of the agents.

Since every agent has s states, we obtain that the cardinality of S is bounded by $O(ks)$ and, hence, the graph B has $O(ks^2)$ vertices by Theorem 2.14. □

The proof of Theorem 2.14 in [Fra+06b] uses the fact that when traversing a 3-regular graph the next state of an s-state agent only depends on the previous state and the label $l \in \{0, 1, 2\}$ of the edge leading back to the previous vertex. Thus, after at most $3s$ steps, the state of the agent and therefore also the next label chosen need to repeat with a period of length at most $3s$. For cooperative agents, however, the next state and label that are chosen may also depend on the positions and states of the other agents. We therefore need to account for the positions of all agents when forcing them into a periodic behavior. To this end, we will consider the relative positions of the agents with respect to a given vertex v. For our purposes, it is sufficient to define the relative position of an agent A_i by the shortest traversal sequence, i.e., the traversal sequence corresponding to a shorted path, leading from v to the location of A_i. This motivates the following definition.

Definition 2.16. *The **configuration** of a set of k agents $\mathcal{A} = \{A_1, \ldots, A_k\}$ in a graph G with respect to a vertex v is a $(3k)$-tuple $(\sigma_1, l_1, r_1, \sigma_2, l_2, r_2, \ldots, \sigma_k, l_k, r_k)$, where σ_i is the current state of A_i, l_i is the label of the edge leading back to the previous vertex visited by A_i and r_i is the shortest traversal sequence from v to A_i, where ties are broken in favor of lexicographically smaller sequences and where we set $r_i = \perp$ if the location of A_i is v.*

In order to limit the number of possible configurations, we will force the agents to stay close together. Intuitively, we can achieve this for any graph G by replacing all edges with $(k-1)$-barriers. This way, only all agents together can move between neighboring vertices of the original graph G. To formalize this, we first need to explain how edges of a graph can be replaced by barriers. Since our construction may not be 3-regular, we need a way to extend it to a 3-regular graph.

Definition 2.17. *Given a graph G, with vertices of degrees 2 and 3, we define the **3-regular extension** $\overline{G}$ as the graph resulting from copying G and connecting every vertex v of degree 2 to its copy v'. As the edges incident to v and v' have the same labels, it is possible to label the new edge $\{v, v'\}$ with a locally unique label in $\{0, 1, 2\}$.*

Note that the 3-regular extension only increases the number of vertices of the graph by a factor of 2. Given a 3-regular graph G and an r-barrier B for a set of k agents $\mathcal{A}$ with $k \geq r$, we replace edges of G using the following construction. First, for every $l \in \{0, 1, 2\}$ we replace every edge $\{a, b\}$ labeled l with the gadget $B(l)$ shown in Figure 2.6, and we call the resulting graph $G_1(B)$. By construction, the labels of the edges incident to the same vertex in $G_1(B)$ are distinct. However, certain vertices only have degree 2. We take the 3-regular extension of $G_1(B)$ and define the resulting graph as $G(B) := \overline{G_1(B)}$.

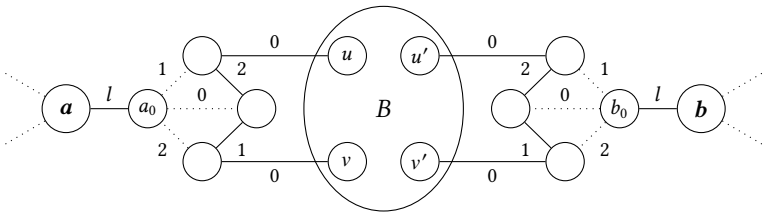


Figure 2.6: An edge $\{a, b\}$ labeled l is replaced with the gadget $B(l)$ containing an r-barrier B. Only the dotted edges incident to a_0 and b_0 that are not labeled l are part of the gadget. Consequently, the gadget contains two vertices of degree 2. The vertices a and b are macro vertices of the graph $G(B)$.

The graph $G(B)$ contains two copies of $G_1(B)$. To simplify exposition, we identify each vertex v with its copy v' in $G(B)$. Then, there is a canonical bijection between the vertices in G and the vertices in $G(B)$ which are not part of a gadget $B(l)$. These vertices can be thought of as the original vertices of G, and we call them **macro vertices**.

We now establish that the agents always stay close to each other in the graph $G(B)$.

Lemma 2.18. *Let G be a connected 3-regular graph and let B be a $(k-1)$-barrier for a set of k agents $\mathcal{A}$ with s states each. Then, the following statements hold for the graph $G(B)$:*

(a) For all edges $\{v, v'\}$ in G no strict subset $\mathcal{A}' \subsetneq \mathcal{A}$ of the agents can get from macro vertex v to macro vertex v' in $G(B)$ without all other agents also entering the gadget $B(l)$ between v and v', where $l \in \{0, 1, 2\}$.

(b) At each step of the walk of $\mathcal{A}$ in G, there is some macro vertex v such that all agents are at v or in one of the surrounding gadgets $B(0)$, $B(1)$ and $B(2)$.

Proof. For the sake of contradiction, assume that there is a strict subset of agents $\mathcal{A}' \subsetneq \mathcal{A}$ that walks from a macro vertex v in $G(B)$ to a distinct macro vertex v' without the other agents entering the gadget between v and v' at any time during the traversal. The graph $G(B)$ contains two copies of $G_1(B)$, but all vertices in the $(k-1)$-barriers within $G_1(B)$ have degree 3. Thus, $\mathcal{A}'$ must have traversed some $(k-1)$-barrier B while only agents in $\mathcal{A}'$ enter B at any time of the traversal. This is a contradiction, as $|\mathcal{A}'| \leq k-1$ and B is a $(k-1)$-barrier. Therefore, the agents $\mathcal{A}$ need to all enter the gadget between v and v' to to get from a macro vertex v to a distinct macro vertex v'. This shows the first part of the claim.

For the second part of the claim, note that because of property (b) of the barrier B the agents cannot split up into two groups such that after the traversal of the gadget between v and v' one group is at v (or one of the vertices at distance at most 4 from v which are not part of the barrier B) and the other group is at v' (or one of the vertices at distance 4 from v' which are not part of the barrier B). This implies that if we consider the positions of the agents after an arbitrary number of steps and let v be the macro vertex last visited by an agent in $\mathcal{A}$, then all agents must be located at v or one of the three surrounding gadgets. □

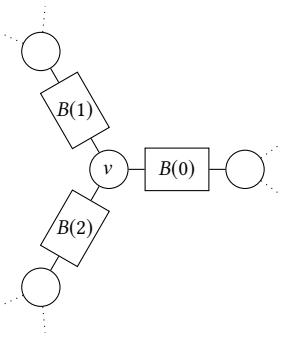


Figure 2.7: A macro vertex v in a graph $G(B)$ surrounded by the three gadgets $B(0)$, $B(1)$ and $B(2)$.

We will frequently consider the configuration of $\mathcal{A}$ in a graph of the from $G(B)$ with respect to some macro vertex v. Recall from the definition that the graph $G(B)$ contains two copies of the graph $G_1(B)$ and actually there exists a macro vertex v and a copy v'. Thus, when we talk about configurations of $\mathcal{A}$ in $G(B)$ with respect to some macro vertex v, we mean that we consistently choose one of the copies $G_1(B)$ and consider the configuration of $\mathcal{A}$ with respect to the macro vertex in this copy.

Let B be a $(k-1)$-barrier for a set of k cooperative s-state agents $\mathcal{A} = \{A_1, \ldots, A_k\}$ that all start in some macro vertex v_0 of $G(B)$. Iteratively, define $t_0 = 0$ and t_i to be the first point in time after t_{i-1}, when one of the agents in $\mathcal{A}$ visits a macro vertex v_i distinct from v_{i-1}. Then v_i is a neighbor of v_{i-1} in G and by Lemma 2.18, all agents are at v_i or one of the incident gadgets. The sequence of macro vertices $v_0, v_1, \ldots$, which is a sequence of neighboring vertices in G, yields a unique sequence of labels $l_0, l_1, \ldots$ of the edges between the neighboring vertices in G, which we call the **macro traversal sequence** of $\mathcal{A}$ starting in vertex v_0 in $G(B)$. Note that the macro traversal sequence may be finite.

Consider the traversal sequence $l_0, l_1, \ldots$ of a single agent in a 3-regular graph G and the traversal sequence $l'_0, l'_1, \ldots$ of the same agent in another 3-regular graph G'. If the state of the agent and label of the edge to the previous vertex in G after i steps is the same as the state in G' after j steps, then the traversal sequences coincide from that point on, i.e., $l_{i+h} = l'_{j+h}$ holds for all $h \in \mathbb{N}$. The reason is that the graphs we consider are 3-regular and the label of every edge $\{u, v\}$ is the same at u and at v. Therefore, once the state and label to the previous vertex are the same, the agent makes the same transitions as it can gain no new information while traversing the graph. We want to obtain a similar result for a set of agents. However, in general it is not true that if the configuration of a set of agents in a graph G after i steps is the same as after j steps in G', then the next configurations and chosen labels of each agent coincide. This is because an agent can be used to mark a particular vertex and this can be used to detect differences in two 3-regular graphs G and G'. For instance, one agent could remain at a vertex v while the other one walks in a loop that is only part of one of the graphs and this may lead to different configurations. That is why we consider graphs of the form $G(B)$. In

these graphs, all macro vertices look the same, as they are surrounded by the same gadgets, and the agents have to stay close together, making it impossible for the agents to detect a loop that is part of one of the graphs, but not the other. This intuition is formally expressed in the following technical lemma.

Lemma 2.19. *Let B be a $(k-1)$-barrier for a set of k s-state agents $\mathcal{A}$, and let G and G' be two 3-regular graphs. Let $v_0, v_1, \ldots$ be the sequence of macro vertices visited by $\mathcal{A}$ in $G(B)$, let $l_0, l_1, \ldots$ be the corresponding macro traversal sequence, let $t_0 = 0$, and let t_i be the first time after t_{i-1} that an agent in $\mathcal{A}$ visits v_i. Let $v'_0, v'_1, \ldots$ and $l'_0, l'_1, \ldots$ and t'_i be defined analogously with respect to $G'(B)$. If there are $t \in \{t_i, \ldots, t_{i+1} - 1\}$ and $t' \in \{t'_j, \ldots, t'_{j+1} - 1\}$ for some $i, j \in \mathbb{N}$, such that after t steps in $G(B)$ the configuration of $\mathcal{A}$ with respect to v_i is the same as after t' steps in $G'(B)$ with respect to v'_j, then:*

(a) We have $l_{i+h} = l'_{j+h}$ for all $h \in \mathbb{N}$.

(b) The configuration of $\mathcal{A}$ in $G(B)$ after t_{i+h} steps with respect to v_{i+h} is the same as configuration of $\mathcal{A}$ in $G'(B)$ after t'_{j+h} steps with respect to v'_{j+h} for all $h \in \mathbb{N}$, $h > 0$.

Proof. In order to simplify the notation of the proof, we abuse notation and overwrite the definition of t_i and t'_j by setting $t_i := t$, $t'_j := t'$. By induction on $h \in \mathbb{N}$, we show that the configuration of $\mathcal{A}$ after t_{i+h} steps in $G(B)$ with respect to v_{i+h} is the same as the configuration of $\mathcal{A}$ after t'_{j+h} steps in $G'(B)$ with respect to v'_{j+h}. The induction step also shows that we have $l_{i+h} = l'_{j+h}$ for all $h \in \mathbb{N}$.

For $h = 0$ we have by assumption (and as we redefined t_i and t'_j) that after t_i steps in $G(B)$ the configuration of $\mathcal{A}$ with respect to v_i is the same as after t'_j steps in $G'(B)$ with respect to v'_j.

Now, assume that the statement holds for some $h \in \mathbb{N}$. The idea of the proof is that, in between visits to macro vertices, the agents behave the same in the two graphs and, in particular, they traverse the same gadget $B(l)$ in both settings in such that $l_{i+h} = l'_{j+h}$.

The graphs $G(B)$ and $G'(B)$ locally look the same to the agents in v_{i+h} and v'_{j+h} as both macro vertices are surrounded by the same gadgets, as shown in Figure 2.7. Formally, there is a canonical graph isomorphism γ from the induced subgraph of $G(B)$ containing v_{i+h} and all surrounding gadgets to the induced subgraph of $G'(B)$ containing v'_{j+h} and all surrounding gadgets. Moreover, γ respects the labeling and maps v_{i+h} to v'_{j+h}. As the configuration of $\mathcal{A}$ after t_{i+h} steps with respect to v_{i+h} is the same as the configuration of A after t'_{j+h} steps with respect to v'_{j+h}, the isomorphism also respects the positions of all the agents. As v_{i+h+1} is the first macro vertex visited after v_{i+h}, all agents are at v_{i+h} or any of the surrounding gadgets until the agents $\mathcal{A}$ reach v_{i+h+1} by Lemma 2.18. The same holds for v'_{j+h} and v'_{j+h+1}. Iteratively, for $c = 0, 1, \ldots$ the following holds until the agents reach the next macro vertex v_{i+h+1} or v'_{j+h+1}:

1. For every agent $A \in \mathcal{A}$, the state of A and the edge label to the previous vertex after $t_{i+h} + c$ steps in $G(B)$ is the same as the state of A and the edge label to the previous vertex after $t'_{j+h} + c$ steps in $G'(B)$.
2. The isomorphism γ maps the position of every agent $A \in \mathcal{A}$ after $t_{i+h} + c$ steps in $G(B)$ to the position of A after $t'_{j+h} + c$ steps in $G'(B)$.

This implies that macro vertices v_{i+h} and v_{i+h+1} are connected with the same gadget as v'_{j+h} and v'_{j+h+1}, i.e., $l_{i+h} = l'_{j+h}$. Furthermore, there is $\bar{c}$ such that $t_{i+h+1} = t_{i+h} + \bar{c}$ and $t'_{j+h+1} = t'_{j+h} + \bar{c}$. Moreover, the configuration of $\mathcal{A}$ with respect to v_{i+h+1} after t_{i+h+1} steps is the same as with respect to v'_{j+h+1} after t'_{j+h+1} steps. □

Let $2 \leq r \leq k$. In order to construct an r-barrier B' for a set $\mathcal{A}$ of k cooperative s-state agents given an $(r-1)$-barrier B, we need to examine the behavior of all subsets of r agents. There are $\binom{k}{r}$ subsets of r agents and the behavior of two different subsets of r agents may be completely different. We denote these $\binom{k}{r}$ subsets of r agents by $\mathcal{A}_1^{(r)}, \ldots, \mathcal{A}_{\binom{k}{r}}^{(r)}$.

Assume, we have an $(r-1)$-barrier B for a set of k agents $\mathcal{A}$. For $1 \leq j \leq \binom{k}{r}$, consider the behavior of only the subset of agents $\mathcal{A}_j^{(r)}$ in a graph of the form $G(B)$. Let $v_0, v_1, \ldots$ be the sequence of macro vertices, $l_0, l_1, \ldots$ the corresponding macro label sequence, $t_0 = 0$, and t_i be the first time after t_{i-1} that an agent in $\mathcal{A}_j^{(r)}$ visits v_i. Between steps t_{i-1} and t_i all agents are located at v_{i-1} or one of the surrounding gadgets $B(0), B(1), B(2)$ by Lemma 2.18. Thus, the number of possible locations of the agents can be bounded in terms of the size of the gadgets $B(0)$, $B(1)$, and $B(2)$. In addition, every agent has at most s states. Therefore the number of configurations of $\mathcal{A}_j^{(r)}$ with respect to v_i between steps t_{i-1} and t_i can also be bounded in terms of s and the size of the gadgets. In particular, this bound is independent of the specific subset of agents $\mathcal{A}_j^{(r)}$. For a sufficiently large number of steps, a configuration must repeat and, by applying Lemma 2.19 for $G = G'$, the macro label sequence becomes periodic. The other crucial property that follows from Lemma 2.19 is that the macro label sequence is independent of the underlying 3-regular graph G. As a consequence, we may denote by α_B the maximum over all $j \in \left\{1 \ldots \binom{k}{r}\right\}$ of the number of steps in the macro label sequence until $\mathcal{A}_j^{(r)}$ is twice in the same configuration in $G(B)$ with respect to two macro vertices, i.e., there are $a,\ b \leq \alpha_B$ such that the configuration of $\mathcal{A}_j^{(r)}$ at t_a with respect to v_a is the same as at t_b with respect to v_b. Note that the value of α_B depends on the size of the barrier B and thus also on the values of s and r.

Given the definition of α_B, we are now in position to present the construction of an r-barrier given an $(r-1)$-barrier. We will later bound α_B and, thus, the size of the r-barrier in Lemma 2.22.

Theorem 2.20. *Given an $(r-1)$-barrier B with n vertices for a set $\mathcal{A}$ of k agents with s states each, we can construct an r-barrier B' for $\mathcal{A}$ with the following properties:*

(a) We have $B' = H(B)$ for a suitable 3-regular graph H.

(b) If $\{u, v\}$ and $\{u', v'\}$ are the two distinguished edges of B', then any path from u or v to u' or v' contains at least 3 distinct barriers B.

(c) The r-barrier B' contains at most $O\left(\binom{k}{r} \cdot n \cdot \alpha_B^2\right)$ vertices.

Proof. For $j \in \left\{1, 2, \ldots, \binom{k}{r}\right\}$, consider a subset of r agents $\mathcal{A}_j^{(r)}$ starting at a vertex v_0 in a graph $G(B)$. Let $t_0 = 0$ and for $i = 1, 2, \ldots$ iteratively define t_i to be the first point in time after t_{i-1}, when an agent in $\mathcal{A}_j^{(r)}$ visits a macro vertex v_i distinct from v_{i-1}. Then $v_0, v_1, \ldots$ is the macro label sequence of $\mathcal{A}_j^{(r)}$

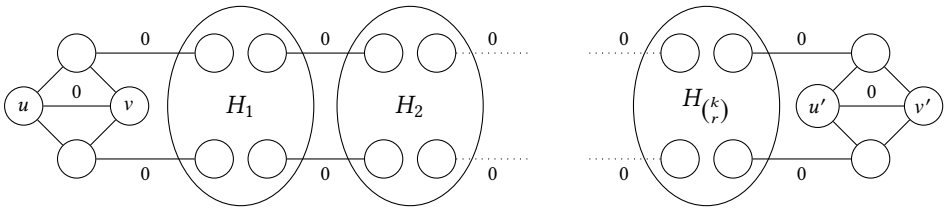


Figure 2.8: Connecting the graphs $H_1, H_2, \ldots, H_{\binom{k}{r}}$ to a graph H, yields the r-barrier $H(B)$.

in $G(B)$ with a corresponding macro label sequence $l_0, l_1, \ldots$. After at most α_B steps, the agents in $\mathcal{A}_j^{(r)}$ are twice in the same configuration with respect to two macro vertices, i.e., there are $a, b \in \mathbb{N}$ with $a < b \leq \alpha_B$ such that after t_a steps the configuration of $\mathcal{A}_j^{(r)}$ with respect to v_a is the same as after t_b steps with respect to v_b. Note that α_B is a bound on the maximum possible number of steps until the configuration repeats and therefore independent of the specific subset of agents $\mathcal{A}_j^{(r)}$. The possible configurations of $\mathcal{A}$ at times $t_0, t_1, \ldots$ can hence be enumerated $x_1, \ldots, x_{\alpha_B}$.

By Lemma 2.19, the configuration of the set of agents $A_j^{(r)}$ uniquely determines the next label in the macro label sequence of $A_j^{(r)}$, independently of the underlying graph G. We can therefore define a single agent $\bar{A}_j$ whose state corresponds to the configuration of the set of agents $A_j^{(r)}$ and whose label sequence is the macro label sequence of $A_j^{(r)}$. More precisely we define $\bar{A}_j$ as follows: The set of states of $\bar{A}_j$ is $\{\sigma_1, \ldots, \sigma_{\alpha_B}\}$. Moreover, in state σ_h the agent $\bar{A}_j$ traverses the edge labeled l and transitions to $\sigma_{h'}$ if the set of agents $\mathcal{A}_j^{(r)}$ in configuration x_h at a time t_i will traverse the gadget $B(l)$ to the next vertex v_{i+1} in the macro vertex sequence where it arrives in configuration $x_{h'}$ at time t_{i+1} (this means that $l = l_i$ is the next label in the macro label sequence of $\mathcal{A}_j^{(r)}$ in configuration x_h). The starting state of $\bar{A}_j$ corresponds to the configuration, where all the agents in $\mathcal{A}_j^{(r)}$ are in their starting states and located at the same vertex. Note that the transition function $\bar{\delta}$ of $\bar{A}_j$ described above is well-defined because, by Lemma 2.19, the next label l_i in the macro label sequence of $\mathcal{A}_j^{(r)}$ only depends on the configuration of $\mathcal{A}_j^{(r)}$ at t_i and is independent of the underlying graph G. By construction, the macro traversal sequence of $\mathcal{A}_j^{(r)}$ in $G(B)$ is exactly the same as the traversal sequence of $\bar{A}_j$ in G, independently of the graph G. Applying Lemma 2.15 for the single agent $\bar{A}_j$, we obtain a 1-barrier H_j with $O(\alpha_B^2)$ vertices that cannot be traversed by $\bar{A}_j$, irrespective of its starting state.

We now connect the graphs $H_1, \ldots, H_{\binom{k}{r}}$ as shown in Figure 2.8, and we let H denote the resulting graph. We first show that the graph $B' := H(B)$ is an r-barrier for $\mathcal{A}$ and and then show the three additional properties in the claim.

For property (a) of an r-barrier, assume, for the sake of contradiction, that there is a subset of r agents $\mathcal{A}_j^{(r)}$ and some graph G connected to $H(B)$ such that the agents $\mathcal{A}_j^{(r)}$ can traverse $H(B)$ from u to u'. Then there must be a consecutive subsequence $w_0, w_1, \ldots, w_h$ of the macro vertex sequence of $\mathcal{A}_j^{(r)}$ during the traversal of $H(B)$ with the following properties: The vertices $w_1, \ldots, w_{h-1}$ are contained in $H_j(B)$, w_0 and w_h are not contained in $H_j(B)$, w_1 and w_{h-1} (as vertices in the 1-barrier H_j) are incident to different distinguished edges (i.e., $\{u, v\}$ or $\{u', v'\}$ in Figure 2.5) of the 1-barrier

H_j. Thus, the set of agents $\mathcal{A}_j^{(r)}$ starting in w_0 in a suitable configuration x_i traverses the graph $H_j(B)$ from w_1 to w_{h-1}. This means that for a suitable graph G' and starting state σ_i the agent $\bar{A}_j$ can traverse H_j. But this is a contradiction as we constructed H_j as a 1-barrier for $\bar{A}_j$ using Lemma 2.15 and the 1-barrier H_j is independent of the starting state of $\bar{A}_j$.

For property (b) of an r-barrier, let $\mathcal{A}' \subseteq \mathcal{A}$ be a set of agents with $|\mathcal{A}'| = r + 1$. Assume, for the sake of contradiction, that there is some graph G connected to $H(B)$ such that after the agents of $\mathcal{A}'$ (and no other agents) enter $H(B)$ a subset $\emptyset \neq \mathcal{A}'_1 \subsetneq \mathcal{A}'$ leaves $H(B)$ via u or v and the other agents $\mathcal{A}'_2 := \mathcal{A}' \setminus \mathcal{A}'_1$ via u' or v'. Since B is an $(r-1)$-barrier, no set of at most $r-1$ agents can get from a macro vertex to a distinct macro vertex in $H(B)$. Thus, we must have $|\mathcal{A}'_1| \geq r$ or $|\mathcal{A}'_2| \geq r$. Without loss of generality, we assume that the first case occurs, which implies $|\mathcal{A}'_1| = r$ and $|\mathcal{A}'_2| = 1$. For the single agent in $\mathcal{A}'_2$ to leave $H(B)$ via u' or v' at least $r-1$ agents from $\mathcal{A}'_1$ must be in a gadget adjacent to u' or v'. But all these $r-1$ agents afterwards leave $H(B)$ via u or v and they need the remaining agent in $\mathcal{A}'_1$ to even get to a distinct macro vertex. But then the set of r agents $\mathcal{A}'_1$ traverses the subgraphs $H_j(B)$ for all $j \in \{1, \dots, \binom{k}{r}\}$, which again leads to a contradiction as in the proof for the first property (for j such that $\mathcal{A}'_1 = \mathcal{A}_j^{(r)}$).

Finally, we obviously have $B' = H(B)$ for a 3-regular graph H by construction and the second additional property follows from the fact that any path from u or v to u' or v' in H has length at least 3. Further, each H_j contains $O(\alpha_B^2)$ vertices and therefore H has at most $O(\binom{k}{r} \cdot \alpha_B^2)$ vertices. As B has n vertices, the number of vertices of $B' = H(B)$ is at most $O(\binom{k}{r} \cdot n \cdot \alpha_B^2)$, where we use that H is 3-regular and therefore the number of edges of H that are replaced by a copy of B is $3/2$ times the number of its vertices. □

We now fix a set of k agents $\mathcal{A}$ with s states each and let B_1 be the 1-barrier given by Lemma 2.15 and B_r for $1 < r \leq k$ be the r-barrier constructed recursively using Theorem 2.20. Moreover, we let n_r be the number of vertices of B_r and $\alpha_r := \alpha_{B_{r-1}}$ be the maximum number of steps in the macro label sequence that a set of r agents from $\mathcal{A}$ can execute in a graph of the form $G(B_{r-1})$ until their configuration repeats.

We want to bound the number of vertices n_k of B_k and thus, according to Lemma 2.13, also the number of vertices of the trap for $\mathcal{A}$. By Theorem 2.20, there is a constant $c \in \mathbb{N}$ such that $n_r \leq c\binom{k}{r} n_{r-1} \alpha_r^2$. In order to bound n_r, we therefore need to bound α_r.

One possible way to obtain an upper bound on α_r is to use Lemma 2.18 stating that there always is a macro vertex v such that all agents are located at v or inside one of the surrounding gadgets. Counting the number of possible positions within these three gadgets and states of the agents then gives an upper bound on α_r. For the tight bound in our main result, however, we need a more careful analysis of the recursive structure of our construction and also need to consider the configurations of the agents at specific times. We start with the following definition and a technical lemma.

For $j \in \{1, \dots, r-1\}$, we say that a vertex w' is ***j*-adjacent** to some other vertex w if there is a path P from w to w' that does not traverse a j-barrier B_j, i.e., P does not contain a subpath leading from one vertex of the distinguished edge $\{u, v\}$ to a vertex of the other distinguished edge $\{u', v'\}$

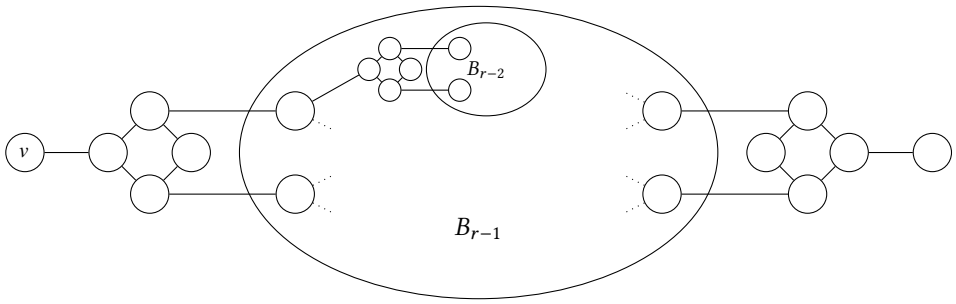


Figure 2.9: Recursive structure of $B(l)$ containing i-barriers for $i \in \{1, \dots, r-1\}$.

in B_j. As a convention, every vertex w is j-adjacent to itself for all $j \in \{1, \dots, r-1\}$. Note that a vertex w' contained inside a j-barrier may be j-adjacent to some vertex w outside the barrier if there is a path from w to w' that does not traverse a distinct j-barrier.

Lemma 2.21. *Let v be a macro vertex in $G(B_{r-1})$. Then for $j \in \{1, \dots, r-1\}$ the number of vertices that are j-adjacent to v is bounded by $2^{4(r-j)}n_j$.*

Proof. In order to bound the number of j-adjacent vertices, we examine the recursive structure of one of the gadgets $B(l)$ incident to v, as shown in Figure 2.9. By Theorem 2.20 an $(r-1)$-barrier B' for $r \geq 3$ is constructed from a 3-regular graph H and an $(r-2)$-barrier B such that $B' = H(B)$. Hence, the gadget $B(l)$, which contains the barrier B_{r-1}, also contains many copies of the barrier B_{r-2}, which again contain many copies of the barrier B_{r-3} (if $r \geq 4$) and so on.

We first observe that the distance from v to any j-adjacent vertex, which is not contained in a barrier B_j, is at most $3(r-j)+1$. This observation is clear for $j = r-1$ and follows for $r-2, r-3, \dots$ by examining the recursive structure given in Figure 2.9. As $G(B_{r-1})$ is 3-regular, there are at most $2^{3(r-j)+1}$ such vertices. Moreover, any j-barrier B_j containing vertices that are j-adjacent to v, in particular contains a vertex with a distance of exactly $3(r-j)$ to v. As $G(B_{r-1})$ is 3-regular, there are at most $2^{3(r-j)}$ vertices of distance exactly $3(r-j)$ from v and therefore at most $2^{3(r-j)}$ different j-barriers, with n_j vertices each, containing j-adjacent vertices. Thus, there are at most $2^{3(r-j)}n_j$ vertices that are j-adjacent to v and contained in a barrier B_j. Overall, the number of j-adjacent vertices to v can therefore be bounded by

$$2^{3(r-j)}n_j + 2^{3(r-j)+1} \leq 2^{4(r-j)}n_j,$$

where we used $n_j \geq 2$ and $j \leq r-1$. □

The idea now is to consider the configuration of the agents with respect to a macro vertex v_i exactly at the time t when at least $\lceil r/2 \rceil + 1$ agents are $\lceil r/2 \rceil$-adjacent to v_i. We then further use the fact that it is not possible to partition the agents $\mathcal{A}$ into two groups $\mathcal{A}'$ and $\mathcal{A}''$ with at most $i \geq \lceil r/2 \rceil$ agents each that are separated on any path by at least two i-barriers. This yields the following bound on α_r.

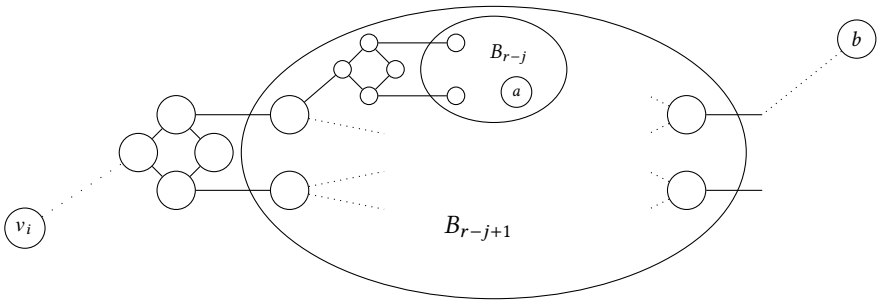


Figure 2.10: An $(r-j+1)$-barrier adjacent to v_i containing $(r-j)$-barriers.

Lemma 2.22. *Let $\mathcal{A}$ be a set of k agents, $s \geq 2$ and $r \in \{2, \ldots, k\}$. We then have*

$$\alpha_r \leq s^{7r^2} \cdot n_{\lceil r/2 \rceil}^{\lceil r/2 \rceil} \cdot n_{r-1} \cdot \prod_{j=\lceil r/2 \rceil+1}^{r-1} n_j.$$

Proof. Let $\mathcal{A}^{(r)} \subseteq \mathcal{A}$ be an arbitrary subset of r agents. In order to bound α_r, we consider the behaviour of this subset $\mathcal{A}^{(r)}$ of agents in a graph of the form $G(B_{r-1})$. We let v_0 be the starting vertex of the set of agents $\mathcal{A}^{(r)}$ in $G(B_{r-1})$ and let $t_0 = 0$. Again, we iteratively define t_i be the first point in time after t_{i-1}, when an agent in $\mathcal{A}^{(r)}$ visits a macro vertex v_i distinct from v_{i-1}.

Because of the recursive structure of the barriers, see Figure 2.9, every macro vertex is surrounded by $\lceil r/2 \rceil$-barriers and any path between two consecutive macro vertices v_{i-1} and v_i contains at least one barrier $B_{\lceil r/2 \rceil}$ (note that $r \geq 2$ by assumption). In order to reach the vertex v_i after visiting v_{i-1}, at least $\lceil r/2 \rceil + 1$ agents from $\mathcal{A}^{(r)}$ are necessary to traverse such an $\lceil r/2 \rceil$-barrier. Thus, at some time $t \in \{t_{i-1}, \ldots, t_i - 1\}$ at least $\lceil r/2 \rceil + 1$ agents must be at a vertex that is $\lceil r/2 \rceil$-adjacent to v_i, as otherwise the agents would not be able to reach v_i.

The crucial observation at this point is that by Lemma 2.19 the number of possible configurations at this time t also bounds α_r, the number of possible steps in the macro label sequence after which a configuration of $\mathcal{A}^{(r)}$ with respect to a macro vertex must repeat. The reason is that whenever the set of agents $\mathcal{A}^{(r)}$ traverse a gadget $B(l)$ there has to be a time t with the properties described above.

Let $\mathcal{A}_1$ denote the set of agents that are at a vertex that is $\lceil r/2 \rceil$-adjacent to v_i at time t, and let $\mathcal{A}_2 := \mathcal{A}^{(r)} \setminus \mathcal{A}_1$. We claim the following: For $j \in \{1, \ldots, |\mathcal{A}_2|\}$, there are at least $(r-j)$ agents that are located at a vertex which is $(r-j)$ adjacent to v_i.

For $j = |\mathcal{A}_2|$, we have $r - j = |\mathcal{A}_1| > \lceil r/2 \rceil$. Thus, the claim holds by definition of $\mathcal{A}_1$, since there are $r-j$ agents, namely the set of agents $\mathcal{A}_1$, which are located at vertices which are $\lceil r/2 \rceil$-adjacent to v_i and thus also $(r-j)$-adjacent to v_i because $r - j > \lceil r/2 \rceil$.

Now, assume for the sake of contradiction that the claim holds for j, but not for $j-1$. This means that there is a subset of agents $\mathcal{A}' \subset \mathcal{A}^{(r)}$ with $|\mathcal{A}'| = r - j$ such that all agents in $\mathcal{A}'$ are located at vertices which are $(r-j)$-adjacent to v_i. But for $j-1$ the claim does not hold, which implies that all other agents $\mathcal{A}'' := \mathcal{A}^{(r)} \setminus \mathcal{A}'$ are at vertices which are not $(r-j+1)$-adjacent: If there was an agent $A \in \mathcal{A}''$ at a vertex which is $(r-j+1)$-adjacent, then $\mathcal{A}' \cup \{A\}$ would be a set of $(r-j+1)$

agents which are all at $(r-j+1)$-adjacent vertices, which is a contradiction to the choice of j.

But the path between any pair of vertices (a, b), such that a is $(r-j)$-adjacent to v_i and b is not $(r-j+1)$-adjacent to v_i, contains at least two $(r-j)$-barriers, see also Figure 2.10. The reason is that $r-j+1 > \lceil r/2 \rceil \geq 1$ and, by Theorem 2.20, any path from u or v to u' or v' contains at least three $(r-j)$ barriers. Thus the set of agents $\mathcal{A}'$ and $\mathcal{A}''$ are separated by at least two $(r-j)$-barriers on any path and $|\mathcal{A}'| \leq r-j$ as well as $|\mathcal{A}''| = j < r-j$ since $j \leq \lceil r/2 \rceil - 1$. But then a set of at most $r-j$ agents must have traversed a barrier B_{r-j} or a set of at most $r-j-1$ agents must have traversed the gadget between two macro vertices in B_{r-j}, which both is a contradiction.

We now use the bound on the number of j-adjacent vertices from Lemma 2.21 together with the claims to bound α_r. By the claim above, we can enumerate the agents in $\mathcal{A}^{(r)}$ as $A_1, A_2, \ldots, A_r$ so that:

1. For $j \in \{1, \ldots, |\mathcal{A}_1|\}$, $A_j \in \mathcal{A}_1$ and the location of A_j is $\lceil r/2 \rceil$-adjacent to v_i.
2. For $j \in \{|\mathcal{A}_1|+1, \ldots, r-1\}$, $A_j \in \mathcal{A}_2$ and the location of A_j is j-adjacent to v_i.
3. Agent $A_r \in \mathcal{A}_2$ is at v_i or one of the surrounding gadgets by Lemma 2.18.

There are $r!$ possible permutations of the agents and each agent has s possible states. Using Lemma 2.21, we can bound the number of possible locations at time t of the agents in $\mathcal{A}_1$ by $(2^{4(r-\lceil r/2 \rceil)} n_{\lceil r/2 \rceil})^{|\mathcal{A}_1|}$, the number of possible locations of the agents $\{A_{|\mathcal{A}_1|+1}, \ldots, A_{r-1}\}$ by $\prod_{j=|\mathcal{A}_1|+1}^{r-1} 2^{4(r-j)} n_j$ and the number of possible locations of A_r by $2^4 n_{r-1}$. Overall, we can thus bound the number of possible locations of the agents $\mathcal{A}^{(r)}$ at t with respect to v_i by

$$r! \cdot \left(2^{4(r-\lceil r/2 \rceil)} n_{\lceil r/2 \rceil}\right)^{|\mathcal{A}_1|} \left(\prod_{j=|\mathcal{A}_1|+1}^{r-1} 2^{4(r-j)} n_j \right) 2^4 n_{r-1}$$

$$\leq r! \cdot \left(2^{4r}\right)^r \cdot n_{\lceil r/2 \rceil}^{|\mathcal{A}_1|} \cdot n_{r-1} \cdot \prod_{j=|\mathcal{A}_1|+1}^{r-1} n_j \leq 2^{5r^2} \cdot n_{\lceil r/2 \rceil}^{\lceil r/2 \rceil} \cdot n_{r-1} \cdot \prod_{j=\lceil r/2 \rceil+1}^{r-1} n_j,$$

where we used $r! \leq r^r \leq 2^{r^2}$ and $n_{j-1} \leq n_j$ for all $j \in \{2, \ldots, r-1\}$.

In order to bound the number of configurations of the agents $\mathcal{A}^{(r)}$ note that there are s^r possible states of the agents and for each agent 3 possible edge labels to the previous vertex. Combining these bounds with the above bound on the number of locations of the agents, we obtain the following bound on the number of configurations of $\mathcal{A}^{(r)}$ at t with respect to v_i:

$$s^r \cdot 3^r \cdot 2^{5r^2} \cdot n_{\lceil r/2 \rceil}^{\lceil r/2 \rceil} \cdot n_{r-1} \cdot \prod_{j=\lceil r/2 \rceil+1}^{r-1} n_j \leq s^{7r^2} \cdot n_{\lceil r/2 \rceil}^{\lceil r/2 \rceil} \cdot n_{r-1} \cdot \prod_{j=\lceil r/2 \rceil+1}^{r-1} n_j.$$

Here we used $s \geq 2$ and $r \geq 2$. By the observation at the beginning of the proof, the number of possible configurations of $\mathcal{A}^{(r)}$ at t with respect to v_i also bounds α_r. □

Using the bound on α_r from Lemma 2.22, we can bound the number of vertices of the barriers.

Theorem 2.23. *For every set of k agents $\mathcal{A}$ with s states each and every $r \leq k$, there is an r-barrier with at most $O(s^{k \cdot 2^{4 \cdot r}})$ vertices.*

Proof. The existence of an r-barrier follows from Lemma 2.15 and Theorem 2.20 and we further have the following bound on the number of vertices n_r of B_r for a sufficiently large constant $c \in \mathbb{N}$:

$$n_1 \leq cks^2 \quad \text{and} \quad n_r \leq c\binom{k}{r} n_{r-1} \alpha_r^2.$$

It is without loss of generality to assume $s \geq 2$ since otherwise a trap of constant size can trivially be found. Hence, we can plug in the bound on α_r from Lemma 2.22. For the asymptotic bound, we may assume $c \leq s^k$ and we further have $\binom{k}{r} \leq 2^k$. We therefore get

$$\begin{aligned} n_r &\leq s^k \cdot 2^k \cdot n_{r-1} \cdot \left(s^{7 \cdot r^2}\right)^2 \cdot \left(n_{\lceil r/2 \rceil}^{\lceil r/2 \rceil}\right)^2 \cdot n_{r-1}^2 \prod_{j=\lceil r/2 \rceil + 1}^{r-1} n_j^2 \\ &\leq s^{2k+14r^2} \cdot n_{\lceil r/2 \rceil}^{(r+1)} \cdot n_{r-1}^3 \prod_{j=\lceil r/2 \rceil + 1}^{r-1} n_j^2. \end{aligned} \tag{2.4}$$

We proceed to show inductively that $n_r \leq s^{k \cdot 2^{4 \cdot r}}$ holds for all $r \in \{1, \ldots, k\}$. For $r = 1$, we have $n_1 \leq cks^2 \leq s^{4k} \leq s^{k \cdot 2^4}$. Let us assume the claim holds for $1, \ldots, r-1$. From Inequality (2.4) we obtain

$$\begin{aligned} n_r &\leq s^{2k+14r^2} \cdot \left(s^{k \cdot 2^{4 \cdot \lceil r/2 \rceil}}\right)^{r+1} \cdot \left(s^{k \cdot 2^{4(r-1)}}\right)^3 \cdot \prod_{j=\lceil r/2 \rceil}^{r-1} \left(s^{k \cdot 2^{4 \cdot j}}\right)^2 \\ &= s^{2k+14r^2+k \cdot (r+1) \cdot 2^{4 \cdot \lceil r/2 \rceil} + 3 \cdot k \cdot 2^{4(r-1)} + 2k \sum_{j=\lceil r/2 \rceil + 1}^{r-1} 2^{4 \cdot j}}. \end{aligned}$$

Thus, it is sufficient to bound the exponent. As $r \geq 2$, we have $\sum_{i=0}^{r-1} 2^{4 \cdot i} = (2^{4r} - 1)/(2^4 - 1) \leq 2 \cdot 2^{4(r-1)}$ as well as $(r+1) \cdot 2^{4 \lceil r/2 \rceil} \leq 4 \cdot 2^{4(r-1)}$ and $2k + 14r^2 \leq 2 \cdot k \cdot 2^{4(r-1)}$. Hence, we obtain

$$\begin{aligned} &2k + 14r^2 + k \cdot (r+1) \cdot 2^{4 \cdot \lceil r/2 \rceil} + 3 \cdot k \cdot 2^{4(r-1)} + 2k \sum_{j=\lceil r/2 \rceil + 1}^{r-1} 2^{4 \cdot j} \\ &\leq k \cdot \left(2 \cdot 2^{4(r-1)} + 4 \cdot 2^{4(r-1)} + 3 \cdot 2^{4(r-1)} + 4 \cdot 2^{4(r-1)}\right) \leq k \cdot 2^{4 \cdot r}. \end{aligned}$$

This shows $n_r \leq s^{k \cdot 2^{4r}}$, as desired. □

The bound for the barriers above immediately yields the bound for the trap for k agents.

Theorem 2.24. *For any set $\mathcal{A}$ of k agents with at most s states each, there is a trap with at most $O(s^{2^{5k}})$ vertices.*

Proof. We can always add additional unreachable states to all agents so that all of them have s states. Theorem 2.23 yields a k-barrier for a given set of k agents $\mathcal{A}$ with $O(s^{k \cdot 2^{4 \cdot k}})$ vertices. The claim follows from the fact that $k \cdot 2^{4 \cdot k} \leq 2^{5 \cdot k}$ and that a k-barrier with n vertices yields a trap with $O(n)$ vertices for $\mathcal{A}$ by Lemma 2.13. □

Finally, we derive a bound on the number of agents k that are needed for exploring every graph on at most n vertices.

Theorem 2.25. *The number of agents needed to explore every graph on at most n vertices is at least $\Omega(\log \log n)$, if we allow $O((\log n)^{1-\varepsilon})$ bits of memory for an arbitrary constant $\varepsilon > 0$ for every agent.*

Proof. Let $\mathcal{A}$ be a set of k agents with $O((\log n)^{1-\varepsilon})$ bits of memory that explores any graph on at most n vertices. By otherwise adding some unused memory, we may assume that $0 < \varepsilon < 1$ and that there is a constant $c \in \mathbb{N}$ such that all agents in $\mathcal{A}$ have $s := 2^{c \cdot (\log n)^{1-\varepsilon}}$ states. We apply Theorem 2.24 and obtain a trap for $\mathcal{A}$ containing $O(s^{2^{5 \cdot k}})$ vertices. As the set of agents $\mathcal{A}$ explore any graph on at most n vertices, we have $n \leq O(1)s^{2^{5 \cdot k}}$. By taking logarithms on both sides of this inequality, we obtain

$$\log n \leq O(1) + 2^{5k} \log s = O(1) + 2^{5k} \cdot c \cdot (\log n)^{1-\varepsilon}.$$

Multiplication by $(\log n)^{\varepsilon - 1}$ on both sides and taking logarithms yields the claim. □

As an additional agent is more powerful than a pebble (Lemma 2.2), we obtain the following result as a direct corollary of Theorem 2.25.

Corollary 2.26. *An agent with $O((\log n)^{1-\varepsilon})$ bits of memory for an arbitrary constant $\varepsilon > 0$ needs $\Omega(\log \log n)$ pebbles to explore every graph with at most n vertices.*

Chapter 3

Energy Efficient Tree Exploration

In this chapter, we study the exploration of trees under the natural constraint that agents have limited energy resources and movement consumes energy. We model this constraint by bounding the number of edges that an agent can traverse by an integer B, which we call the **energy budget** of the agent. A similar restriction was considered in the piecemeal exploration problem [BRS95; Awe+99; DKK06], where also the number of edge traversals of the agent is bounded and it can refuel by going back to its starting location. A different approach is to consider multiple agents instead of allowing refueling and minimize the energy budget per agent [DKS06; DŁS07] or the number of agents needed for a fixed budget [DDK15]. In our model, we drop the requirement that the tree needs to be completely explored by the agents and focus on exploring the maximum number of vertices with a fixed given number k of initially colocated agents with fixed energy budgets B.

We start by giving a formal introduction of the model and introducing some specific notation in Section 3.1. In Section 3.2, we present a collaborative exploration algorithm for the problem that utilizes global communication between the agents. The challenge is to balance between sending agents in a depth-first manner to avoid visiting the same set of vertices too often and exploring the tree in a breadth-first manner to make sure that there is no large set of vertices close to the root that was missed by the online algorithm. We achieve this by maintaining a set of edge-disjoint subtrees of the part of the tree that is already explored and by iteratively sending an agent from the root to the subtree with the highest root. We show that our algorithm is 3-competitive, i.e., an optimal offline algorithm that knows the tree in advance can explore at most three times as many vertices as our algorithm. We also show that our analysis is tight by giving a sequence of instances showing that the algorithm is not better than 3-competitive. In Section 3.3, we complement this result by showing that no online algorithm can be better than 2.17-competitive. The proof of this general lower bound is based on an adaptive adversary that forces the online algorithm to spend a lot of energy if it completely wants to explore certain subtrees while preventing it from discovering some vertices close to the root.

Bibliographic Information The results presented in this chapter are joint work with Evangelos Bampas, Jérémie Chalopin, Shantanu Das and Christina Karousatou and were published in [Bam+18].

3.1 Terminology and Model

We consider a set $\mathcal{A}$ of k distinct agents initially located at the root v_0 of an undirected, initially unknown, locally edge-labeled tree T. We assume, without loss of generality, that the local port number of the edge leading back to the root r is 0 for any vertex $v \neq v_0$ in T. Otherwise, every agent internally swaps the labels of the edge leading back to the root and the label 0 for every vertex $v \neq r$. Note that in our setting, it does not make a difference if we assume that the vertices are labeled or not because we can uniquely identify every vertex with the sequence of port numbers leading to it from the root v_0. For any vertex v in T, we let $d(v)$ be the depth of v in T. The induced subtree with root v containing v and all vertices below v in T is further denoted by $T(v)$. For a subtree S of T, we write r_S to denote the root of S, i.e., the unique vertex contained in S having the smallest depth in T. Moreover, $|S|$ denotes the number of vertices in S.

The tree is initially unknown to the agents, but they learn the map of the tree as they traverse new edges. Each time an agent arrives at a new vertex, it learns the local port number of the edge through which it arrived, as well as the degree of the vertex. We assume that agents can communicate at arbitrary distances, so the updated map of the tree, including all agent positions, is instantaneously available to all agents (global communication). Each agent has limited energy B and it consumes one unit of energy for every edge that it traverses.

The goal is to design an algorithm Alg that maximizes the total number of distinct vertices visited by the agents. For a given instance $I = \langle T, v_0, k, B \rangle$, where T is a tree, v_0 is the starting vertex of the agents, k is the number of agents, and B is the energy budget of each agent, let $\textsc{Alg}(I)$ denote the total number of distinct vertices visited by the agents using algorithm Alg on the instance I. Similarly, $\textsc{Opt}(I)$ denotes the maximum number of distinct vertices of T that can be explored by the agents using an optimal offline algorithm Opt, i.e., an algorithm with full initial knowledge of the instance I. We measure the performance of an algorithm for this problem by the standard tool of competitive analysis, i.e., we compare a given online algorithm to an optimal offline algorithm which has a complete map of the tree in advance.

3.2 An Algorithm for Maximal Tree Exploration

This section is divided into three parts. First, we present the idea and intution behind our algorithm in Section 3.2.1. Afterwards, we analyze the algorithm and show that it is 3-competitive in Section 3.2.2. Finally, we construct an instance showing that the analysis of the algorithm is tight in Section 3.2.3.

Algorithm 3.1: L-DFS traversal of a tree T starting in a vertex u.

```
Input: tree T, starting vertex u in T
function L-DFS(T,u)
    move on a shortest path to u
    while agent A has energy left and T is not completely explored do
        if the subtree below the current node is completely explored then
            traverse the edge with label 0
        else
            traverse the unexplored edge with the smallest label l > 0
```

Algorithm 3.2: R-DFS traversal of a tree T starting in a vertex u.

```
Input: tree T, starting vertex u in T
function L-DFS(T,u)
    move on a shortest path to u
    while agent A has energy left and T is not completely explored do
        if the subtree below the current node is completely explored then
            traverse the edge with label 0
        else
            traverse the unexplored edge with the largest label l > 0
```

3.2.1 Algorithm Divide & Explore and Intuition

Let us assume that we do a depth-first search of the whole tree T and always choose the smallest label $l > 0$ to an unexplored vertex, as describend in Algorithm 3.1. We call this algorithm L-DFS. We further denote the sequence $(v_0, v_1), (v_1, v_2) \ldots, (v_m, v_0)$ of directed edges obtained by directing every undirected edge of T that the agent traversed in the direction in which the agent traversed the edge in the L-DFS traversal the **L-DFS sequence** of T. Note that every undirected edge $\{v, w\}$ of the tree T appears as (v, w) and (w, v) in this sequence. Similarly, we call a depth-first search of T that always chooses the largest label $l > 0$ to an unexplored vertex an R-DFS and the corresponding sequence of directed edges an **R-DFS sequence**. An implementation of the algorithm R-DFS is given in Algorithm 3.2. Note that the R-DFS sequence of the edges in T is obtained by reversing the order of edges of the L-DFS sequence and changing every edge (v, w) to (w, v).

We call a consecutive subsequence of an L-DFS or R-DFS sequence a **substring**. For an induced subtree $T(v)$ of T, the L-DFS sequence of $T(v)$ is simply a substring of the L-DFS sequence of T. For a subtree S we define the **leftmost** unexplored vertex as the unexplored vertex in S which is incident to the first edge in the L-DFS sequence of S leading to an unexplored vertex and the **rightmost** unexplored vertex as the unexplored vertex in S which is incident to the first edge in the R-DFS

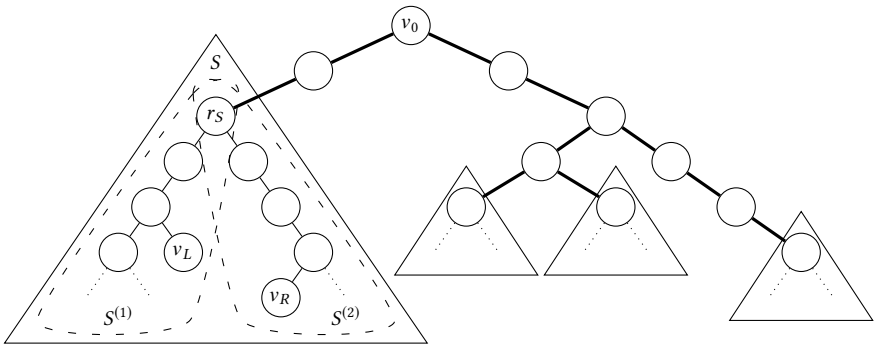


Figure 3.1: Example in which algorithm Divide & Explore in iteration t divides the considered subtree S into two subtrees $S^{(1)}$ and $S^{(2)}$. The tree T_t^R that connects the roots of the subtrees in $\mathcal{T}_t$ is the subtree containing all thick edges.

sequence of S leading to an unexplored vertex.

We further say that an agent A performing an L-DFS **covers** at least s edges $(v_1, v_2), \ldots, (v_s, v_{s+1})$ of the L-DFS sequence of T, if A consecutively visits $v_1, v_2, \ldots, v_s, v_{s+1}$ in this order and the sequence $(v_1, v_2), \ldots, (v_s, v_{s+1})$ is a substring of the L-DFS sequence of T. Similarly, we say that an agent A performing an R-DFS **covers** at least s edges $(v_1, v_2), \ldots, (v_s, v_{s+1})$ of the L-DFS sequence of T, if A consecutively visits $v_{s+1}, v_s, \ldots, v_2, v_1$ in this order and the sequence $(v_1, v_2), \ldots, (v_s, v_{s+1})$ is a substring of the L-DFS sequence of T. Note that two agents A_1 and A_2 may traverse the same edge in the same direction, but still cover two distinct sets of directed edges of the L-DFS sequence, if one agent performs an L-DFS and the other agent an R-DFS.

With these definitions, we are now ready to explain the idea of the algorithm Divide & Explore: During the run of the algorithm, we maintain a set $\mathcal{T}$ of edge-disjoint subtrees of T, initially just containing T. An example is shown in Figure 3.1, where the triangles show the subtrees that are currently contained in the set $\mathcal{T}$. In every iteration, we first move down the root r_S of every subtree S if r_S has no unexplored childen and only one child leading to an unexplored vertex. This first step is later necessary for our analysis. Afterwards, we consider a subtree S which contains an unexplored vertex and has the highest root, i.e., minimizes $d(r_S)$. As long as the leftmost unexplored vertex v_L in S is not too far away from r_S, i.e., $d(v_L) - d(r_S)$ is sufficiently small, we send an agent to v_L and let it continue the L-DFS from there. We do the same if v_R is not too deep and then let the agent continue the R-DFS from v_R. The intuition is that the energy spent to reach r_S is unavoidable, but also the agents in the offline optimum Opt need to spend this energy without exploring new vertices after the tree has been explored up to depth $d(r_S)$. Thus, the agent only potentially wastes energy to reach v_L (or v_R), but from then on explores many new vertices because an agent doing $2m$ edge traversals on a DFS visits at least m distinct vertices. If both v_L and v_R are sufficiently deep, we split S into two edge-disjoint subtrees $S^{(1)}$ and $S^{(2)}$, as shown in Figure 3.1. In this case both $S^{(1)}$ and $S^{(2)}$ contain a sufficiently long part of the L-DFS sequence, which has not been covered by any agent.

This is important because we want to avoid that an agent is sent to a new subtree which only needs little more exploration. A complete description of Divide & Explore is given in Algorithm 3.3.

3.2.2 Proof of 3-Competitiveness

In this subsection, we analyze Algorithm Divide & Explore in order to show that it is 3-competitive. Note that the first agent in Divide & Explore simply performs a depth-first search and explores at least $B/2$ vertices or completely explores the tree. Consequently, if $k = 1$ or if $n < B$, the algorithm is 2-competitive, and thus we assume in the following that $n \geq B$ and $k \geq 2$.

For the analysis of Divide & Explore, we further need the following notation. For every iteration t of the outer while-loop, we let $k_t \in \{1, 2\}$ be the number of agents used by Divide & Explore in this iteration and $k_0 = 2$ be the number of agents used before the first iteration of the outer while-loop. Further, let $\mathcal{T}_t$ be the set of subtrees $\mathcal{T}$ at the end of iteration t and let T_t^R be the unique subtree of T that connects the set of roots $\{r_S \mid S \in \mathcal{T}_t\}$ of all subtrees with the minimum number of edges. Moreover, we denote the subtree S with the highest root considered by Divide & Explore in iteration t by S_t and its root by r_t. Finally $\bar{t}$ denotes the total number of iterations of the while-loop.

The crux of our analysis is to show that the amortized amount of energy spent making progress on the L-DFS or R-DFS is $\frac{2}{3} \cdot k_i \cdot (B - d(r_i))$ for the agents in iteration i, as stated in the following lemma.

Lemma 3.1. *The algorithm Divide & Explore either completely explores T or all agents used by the algorithm together cover at least*

$$\tfrac{2}{3}(|T_{\bar{t}}^R| - 1) + \sum_{0 \leq i \leq \bar{t}} \tfrac{2}{3} \cdot k_i \cdot (B - d(r_i))$$

distinct edges of the total L-DFS sequence of T.

Proof. Let us assume that Divide & Explore does not completely explore T and let $\mathcal{U}_t$ be the subset of $\mathcal{T}_t$ containing all subtrees with an unexplored vertex. We will show by induction over t that all agents used by Divide & Explore up to the end of iteration t together cover at least

$$\tfrac{2}{3}(|T_t^R| - 1) + \sum_{S \in \mathcal{U}_t} \tfrac{2}{3}(B - d(r_S)) + \sum_{0 \leq i \leq t} \tfrac{2}{3} \cdot k_i \cdot (B - d(r_i)) \tag{3.1}$$

distinct edges of the total L-DFS sequence of T. It may happen that in the last iteration $\bar{t}$ of Divide & Explore the third case occurs, but only one agent is left at the root. We will treat this special case separately at the end of the proof. First, we show the lower bound above for all t, for which iteration t is completed, i.e., there are enough agents for Divide & Explore to finish iteration t.

For $t = 0$, we have $\mathcal{U}_0 = \{T\}$ as Divide & Explore does not completely explore T by assumption, $k_0 = 2$, $r_0 = r_T$, and T_t^R only contains r_T. Thus the lower bound (3.1) on the number of edges covered by the first two agents evaluates to $2B$. The first agent used by Divide & Explore performs an L-DFS and covers exactly B edges of the total L-DFS sequence of T. The second agent performs an

R-DFS starting at the root of T and also covers exactly B edges of the total L-DFS sequence of T. The edges covered by the second agent are distinct from the edges covered by the first because T is not completely explored by the algorithm by assumption. Hence, the lower bound (3.1) holds for $t = 0$.

Now, assume that the lower bound (3.1) holds for $t-1$. We will show it for iteration t. Let $\mathcal{U}'_{t-1}$ be the set of subtrees $\mathcal{U}_{t-1}$ after the for-all loop in iteration t terminated and possibly some roots of the trees in $\mathcal{U}_{t-1}$ were moved down. We claim that

$$\tfrac{2}{3}(|T^R_{t-1}| - 1) + \sum_{S \in \mathcal{U}_{t-1}} \tfrac{2}{3}(B - d(r_S)) = \tfrac{2}{3}(|T^R_t| - 1) + \sum_{S \in \mathcal{U}'_{t-1}} \tfrac{2}{3}(B - d(r_S)). \tag{3.2}$$

For any subtree $S \in \mathcal{U}_{t-1}$, let $S' \in \mathcal{U}'_{t-1}$ be the corresponding subtree after the root of S was possibly moved down. The tree T^R_t contains all vertices of the tree T^R_{t-1} plus the path from r_S to $r_{S'}$, i.e., $d(r_S) - d(r_{S'})$ additional vertices, for all $S \in \mathcal{U}_{t-1}$. This already implies (3.2).

Applying (3.2) on the lower bound (3.1) for $t-1$ yields that the number of edges of the total L-DFS sequence of T covered by the agents up to iteration $t-1$ is at least

$$\tfrac{2}{3}(|T^R_t| - 1) + \sum_{S \in \mathcal{U}'_{t-1}} \tfrac{2}{3}(B - d(r_S)) + \sum_{0 \le i \le t-1} \tfrac{2}{3} \cdot k_i \cdot (B - d(r_i)). \tag{3.3}$$

Let now S_t be the subtree with root r_t considered by the algorithm in iteration t as defined above and v_L, v_R be defined as in the algorithm.

First, assume that we have $d(v_L) - d(r_t) \le \max\{1, 1/3 \cdot (B - d(r_t))\}$ and let A_0 be the only agent used by the algorithm in iteration t. Note that if $1/3 \cdot (B - d(r_t)) < 1$, then once it has reached r_t, agent A_0 has either one or two energy left. In the first case, A_0 only explores v_L and makes a progress of 1 on the total L-DFS sequence. In the second case, A_0 makes a progress of 2 on the total L-DFS sequence: it goes to v_L and then either it visits a child of v_L, or it goes back to r_t. Consequently, if $1/3 \cdot (B - d(r_t)) < 1 = d(v_L) - d(r_t)$, A_0 makes a progress of at least $(B - d(r_t)) \ge 2/3 \cdot (B - d(r_t))$ on the total L-DFS sequence.

Suppose now that $1 \le d(v_L) - d(r_t) \le 1/3 \cdot (B - d(r_t))$. Agent A_0 moves to v_L using at most $1/3 \cdot (B - d(r_t))$ energy and then performs an L-DFS. If A_0 does not completely explore S_t, then the set of edges traversed by A_0 starting in v_L and directed in the direction the edge is traversed by A_0 has not been covered by any other agent. Therefore A_0 makes a progress of at least $2/3 \cdot (B - d(r_t))$ edges on the total L-DFS sequence. Adding this progress of agent A_0 to the lower bound in (3.3) on the number of edges covered by the agents in the first $t-1$ iterations and using $\mathcal{U}_t = \mathcal{U}'_{t-1}$ yields the lower bound (3.1) for iteration t.

Next assume that A_0 completely explores the subtree S_t. We then have $\mathcal{U}_t = \mathcal{U}'_{t-1} \setminus \{S_t\}$ and the lower bound (3.1) for iteration t follows directly from the lower bound (3.3) even if A_0 explores only v_L and only covers two new directed edges of the total L-DFS sequence.

The proof when $d(v_R) - d(r_t) \le 1/3 \cdot \max\{1, 1/3 \cdot (B - d(r_t))\}$ is completely analogous.

Finally, assume that the last case occurs in iteration t and S_t is split into two subtrees $S^{(1)}$ and $S^{(2)}$ as defined in the algorithm. Further, let A_1 and A_2 be the agents used in iteration t for performing an R-DFS in $S^{(1)}$ and an L-DFS in $S^{(2)}$, respectively.

We first show that v_L and v_R are below different children of r_t. Note that we have $d(v_L) - d(r_t) > \max\{1, 1/3 \cdot (B - d(r_t))\} \geq 1$ as well as $d(v_R) - d(r_t) > \max\{1, 1/3 \cdot (B - d(r_t))\} \geq 1$. Therefore neither v_L nor v_R are children of r_t. Suppose, for the sake of contradiction, there is a child v of r_t such that both v_L and v_R are contained in $T(v)$. By the definition of v_L and v_R, the subtrees below all other children of r_t must be completely explored. This means r_t only has one child leading to an unexplored vertex. We cannot have $v_L = v_R = v$ as v_L and v_R are not children of r_t. But then the root r_t would be moved down to v and possible further at the beginning of iteration t. This is a contradiction. Therefore, $S^{(1)}$ and $S^{(2)}$ are edge-disjoint, non-empty trees and v_L is contained in $S^{(1)}$ and v_R in $S^{(2)}$.

Agent A_1, which moves according to the call R-DFS($S^{(1)}, r_t$), moves to r_t using $d(r_t)$ energy and starts an R-DFS making a progress of at least $d(v_L) - d(r_t) > 1/3 \cdot (B - d(r_t))$ on the overall L-DFS sequence, as the part of the L-DFS sequence from v_L to r_t has not been covered by any other agent and has length at least $d(v_L) - d(r_t)$. If A_1 does not completely explore $S^{(1)}$, then it makes even a progress of $B - d(r_t)$ on the overall L-DFS sequence.

The second agent used in iteration t, the agent A_2, first moves to r_t using $d(r_t)$ energy and then performs an L-DFS according to the call L-DFS($S^{(2)}, r_t$). We have $d(v_R) - d(r_t) > 1/3 \cdot (B - d(r_t))$ and hence A_2 makes a progress of at least $1/3 \cdot (B - d(r_t))$ edges on the overall L-DFS sequence, as the part of the sequence from r_t to v_R has not been covered by any other agent. If A_2 does not completely explore $S^{(2)}$, then it also makes a progress of $B - d(r_t)$ on the overall L-DFS sequence.

Let $s \in \{0, 1, 2\}$ be the number of subtrees among $\{S^{(1)}, S^{(2)}\}$ that A_1 and A_2 do not explore completely. By the above argument, we showed that overall A_1 and A_2 together make a progress of at least $2/3 \cdot (B - d(r_t)) + s \cdot 2/3 \cdot (B - d(r_t))$ edges on the overall L-DFS sequence of T. Adding this progress to the lower bound (3.3) and using $S_t \in \mathcal{U}'_{t-1} \setminus \mathcal{U}_t$ again yields the lower bound (3.1) for iteration t.

In order to show the claim, let us consider the last iteration $\bar{t}$. If Divide & Explore can complete this iteration, then the claim follows directly from the lower bound (3.1) because $\frac{2}{3}(B - d(r_S)) \geq 0$ for all $S \in \mathcal{U}_t$ as no agent can explore a vertex below depth B in T. Now assume that iteration $\bar{t}$ is not completed. But then we have that the number of edges of the total L-DFS sequence of T covered by the agents up to iteration $\bar{t} - 1$ is at least

$$\tfrac{2}{3}(|T^R_{\bar{t}}| - 1) + \sum_{S \in \mathcal{U}'_{\bar{t}-1}} \tfrac{2}{3}(B - d(r_S)) + \sum_{0 \leq i \leq \bar{t}-1} \tfrac{2}{3} \cdot k_i \cdot (B - d(r_i))$$

by the lower bound (3.3). The above lower bound already implies the claim, as we have $k_{\bar{t}} = 1$ and $\sum_{S \in \mathcal{U}'_{\bar{t}-1}} \frac{2}{3}(B - d(r_S)) \geq \frac{2}{3} \cdot k_{\bar{t}} \cdot (B - dr_{\bar{t}})$. □

With the lower bound above, we can now prove the main result of this section.

Theorem 3.2. *The algorithm Divide & Explore is 3-competitive.*

Proof. Assume that the algorithm Divide & Explore terminates after iteration $\bar{t}$. If it completely explores T, then it is clearly optimal. So let us assume that it runs out of agents in iteration $\bar{t}$.

Let $A_1, A_2, \ldots, A_k$ be the sequence of agents used by Divide & Explore in this order and let agent A_i be used in iteration t_i. We let $d_i := d(r_{t_i})$ be the depth of the root of the subtree visited by A_i in iteration t_i. As the algorithm in every iteration chooses the subtree S with an unexplored vertex which minimizes $d(r_S)$, we have $d_1 \leq d_2 \leq \ldots \leq d_k$.

Note that every undirected edge $\{v, w\}$ of the tree appears exactly twice as a directed edge in the total L-DFS sequence of T, as (v, w) and as (w, v). Thus dividing the bound given by Lemma 3.1 by two yields a lower bound on the number of distinct undirected edges traversed by the agents. As T is a tree, this number plus 1 is a lower bound on the number of vertices visited by the agents. Thus, using the notation T^R instead of $T^R_{\hat{t}}$, we obtain for the given instance I that

$$\textsc{Alg}(I) \geq \tfrac{1}{3}|T^R| + \sum_{1 \leq i \leq k} \tfrac{1}{3} \cdot (B - d_i). \tag{3.4}$$

Let now $A^*_1, \ldots, A^*_k$ be the k agents used by an optimal offline algorithm Opt and let d^*_i be the maximum depth of a vertex in T^R that is visited by the agent A^*_i. This is well-defined as every agent at least visits the root r of T^R. We assume without loss of generality that $d^*_1 \leq d^*_2 \leq \ldots \leq d^*_k$. As the agent A^*_i must use at least d^*_i energy to reach a vertex at depth d^*_i in T^R, we have

$$\textsc{Opt}(I) \leq |T^R| + \sum_{1 \leq i \leq k} (B - d^*_i). \tag{3.5}$$

Consider the maximal index $j \in \{1, \ldots, k\}$ such that $d_j > d^*_j$. If no such j exists, $d_i \leq d^*_i$ holds for all $1 \leq i \leq k$. This implies $\sum_{i=1}^{k}(B - d^*_i) \leq \sum_{i=1}^{k}(B - d_i)$ and thus also $\textsc{Opt}(I)/\textsc{Alg}(I) \leq 3$ by (3.4) and (3.5).

Otherwise, we have $d^*_1 \leq d^*_2 \leq \ldots \leq d^*_j < d_j$. Let $T^j_{\textsc{Alg}}$ be the subtree explored by the first j agents used by Divide & Explore. We claim that all vertices explored by the agents $A^*_1, \ldots, A^*_j$ are contained in $T^j_{\textsc{Alg}}$. Assume, for the sake of contradiction, that there is $1 \leq i \leq j$ such that agent A^*_i explores a vertex u which is not contained in $T^j_{\textsc{Alg}}$. At the moment when the agent A_j is used by Divide & Explore, the root r_S of every subtree $S \in \mathcal{T}_{t_j}$ is contained in $T^R_{t_j}$ and it has depth at least d_j. Let $S' \in \mathcal{T}_{t_j}$ be the subtree containing u. This means that the agent A^*_i must also visit $r_{S'}$ to reach u. But $T^R_{t_j}$ is a subtree of T^R and thus A^*_i visits a vertex in T^R of depth $d(r_{S'}) \geq d_j$. This implies $d^*_i \geq d(r_{S'}) \geq d_j$ contradicting the initial assumption that $d^*_i < d_j$. Consequently, the agents $A^*_1, \ldots, A^*_j$ in Opt only visit vertices in $T^j_{\textsc{Alg}}$. But then the first j agents in Opt visit a strict subset of the vertices visited by the first j agents in Divide & Explore. In this case, we can just replace the agents $A^*_1, \ldots, A^*_j$ and their paths by the agents $A_1, \ldots, A_j$ and their paths in Divide & Explore and $\textsc{Opt}(I)$ does not decrease. By construction and by maximality of j, we then have $d_i \leq d^*_i$ for all $1 \leq i \leq k$, which again implies the claim. □

3.2.3 Lower Bound for Divide & Explore

In this subsection, we construct a sequence of instances to show that the analysis of Divide & Explore is tight. Let $k, d \in \mathbb{N}$, $d \geq 2$ and $B = 3(d-1)$. Our instance $I_{k,d}$ is a tree T consisting of a root v_0 connected to $2k$ paths, of which k have length d and k have length B, as illustrated in Figure 3.2. We

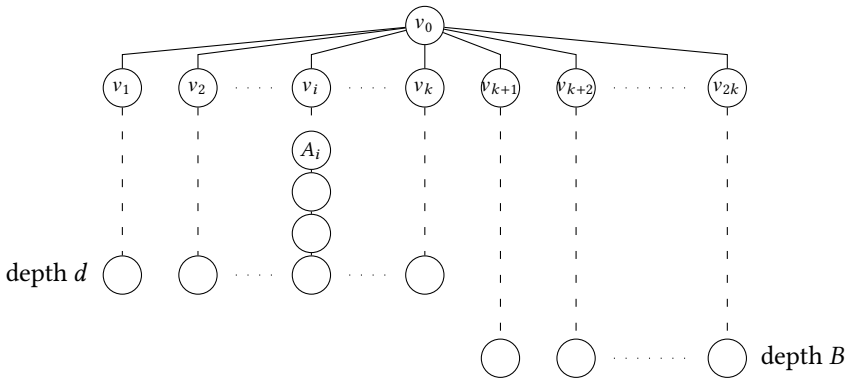


Figure 3.2: Instance showing that the analysis of Divide & Explore is tight.

assume that the edge labels of the edges incident to the root are increasing from left to right, i.e., for all $1 \leq i \leq 2k-1$, the edge label of $\{v_0, v_i\}$ is smaller than the label of $\{v_0, v_{i+1}\}$. We further denote the path $v_0, v_i, \ldots$ up to the leaf of the tree by P_i.

At the beginning of Divide & Explore, one agent A_1 performs an L-DFS and completely explores P_1 and explores P_2 up to depth $d-3$, overall exploring $2d-3$ vertices. The second agent A_2 performs an R-DFS and completely explores the rightmost path P_{2k} of length B, i.e., $B = 3(d-1)$ vertices. From now on, in every iteration of the while loop, we have $\mathcal{T} = \{T\}$, $r_S = v_0$, $d(v_L) = d-2$ and thus

$$d(v_L) - d(r_S) = d - 2 \leq d - 1 = 1/3 \cdot (B - d(r_S)).$$

This means that, for $i \geq 3$, the agent A_i used in the iteration $i-2$ of the outer while-loop, first moves to the unexplored vertex at depth $d-2$ on the path P_{i-1}, then finishes exploring this path, and runs out of energy at depth $d-3$ in P_i. Thus, A_i explores exactly d vertices. Overall, the number of vertices explored by the algorithm is therefore

$$2d - 3 + 3(d-1) + (k-2)d = 5d - 6 + (k-2)d.$$

The optimal offline algorithm sends one agent down each of the paths $P_{k+1}, \ldots, P_{2k}$ exploring $3k(d-1)$ vertices. Hence, we obtain the following lower bound on the competitive ratio:

$$\frac{\textsc{Opt}(I_{k,d})}{\textsc{Alg}(I_{k,d})} = \frac{3k(d-1)}{5d - 6 + (k-2)d} \xrightarrow{d\to\infty, k\to\infty} 3.$$

Algorithm 3.3: Divide & Explore

Input: tree T with root v_0, set of agents $\mathcal{A}$, energy bound B

$\mathcal{T} \leftarrow \{T\}$
L-DFS(T, v_0)
R-DFS(T, v_0)
while T contains unexplored vertex and $\exists$ agent at v_0 **do**
  `// Step 1: move down the roots of the subtrees in` $\mathcal{T}$ `if possible`
  forall $S \in \mathcal{T}$ containing an unexplored vertex **do**
    $r_0 \leftarrow r_S$
    while r_0 only has one child v leading to an unexplored vertex
        and r_0 has no unexplored child **do**
      $r_0 \leftarrow v$
    $\mathcal{T} \leftarrow (\mathcal{T} \setminus \{S\}) \cup \{T(r_0)\}$
  `// Step 2: explore or split the subtree with the highest root`
  $S \leftarrow$ subtree in $\mathcal{T}$ that contains an unexplored vertex and minimizes dr_S
  $v_L \leftarrow$ leftmost unexplored vertex in S
  $v_R \leftarrow$ rightmost unexplored vertex in S
  if $d(v_L) - d(r_S) \leq \max\{1, 1/3 \cdot (B - d(r_S))\}$ **then**
    L-DFS(S, v_L)
  else if $d(v_R) - d(r_S) \leq \max\{1, 1/3 \cdot (B - d(r_S))\}$ **then**
    R-DFS(S, v_R)
  else
    $v \leftarrow$ child of r_S leading to v_R
    $S^{(1)} \leftarrow$ induced subtree of S containing all vertices not in $T(v)$
    $S^{(2)} \leftarrow$ induced subtree of S containing all vertices in $T(v)$ and r_S
    $\mathcal{T} \leftarrow (\mathcal{T} \setminus \{S\}) \cup \{S^{(1)}, S^{(2)}\}$
    R-DFS$(S^{(1)}, r_S)$
    L-DFS$(S^{(2)}, r_S)$

3.3 A General Lower Bound on the Competitive Ratio

In this section, we construct a sequence of instances for a given online algorithm that show a lower bound of $(5 + 3\sqrt{17})/8 \approx 2.17$ on the competitive ratio of any online algorithm. The section is organized as follows: In Section 3.3.3, we first present a simple lower bound of 2 on the competitive ratio and then present our construction for the lower bound of 2.17. As the full proof of the lower bound is quite involved, we first give some intuition and a simplified proof for some special cases in Section 3.3.1. The general proof of the lower bound is then given in Section 3.3.2.

3.3.1 Lower Bound Construction

In order to get some intuition, we first consider a simple example showing a lower bound of 2 on the competitive ratio of any online algorithm.

Proposition 3.3. *There exists no c-competitive online exploration algorithm with $c < 2$.*

Proof. Let k and B be positive integers, B be even and T be a tree with root v_0 connected to k paths of length B and $k \cdot B/2$ paths of length 1. A team of k agents starts at v_0 with energy B each. For every algorithm ALG, the adversary can ensure that no agent that starts at v_0 ever enters one of the long paths by permuting the port numbers of the edges at v_0 accordingly. For every edge that an agent explores, it then needs to go back to v_0 in order to explore other edges. Thus, every agent can explore at most $B/2$ edges and all k agents together at most $k \cdot B/2$ edges since B is even. On the other hand, the offline optimum OPT sends all agents in the long paths exploring $k \cdot B$ edges. □

Note that the simple lower bound of 2 only requires that B is even and otherwise works for any choice of parameter k and B. For the lower bound of $(5 + 3\sqrt{17})/8 \approx 2.17$ on the competitive ratio, we present a sequence of instances where k and B become arbitrarily large. We initially construct an instance with general parameters and at the end choose the parameters to maximize the competitive ratio that the online algorithm can achieve. The lower bound instances that we construct are trees that contain very long paths and high degree vertices at certain depth in the tree. The length of the paths is determined by the online exploration algorithm.

For a given online algorithm ALG, we consider a set of $k := 2l-1$ agents $\mathcal{A}$ for $l \in \mathbb{N}$ with energy B each and we let $\Delta := \left\lceil \sqrt{2 \cdot l \cdot B} \right\rceil + 2l$. We now construct a tree T, which is shown in Figure 3.3, depending on the behavior of the algorithm. The tree T has a root v_0 with l distinct paths, each going from v_0 to a vertex $v_i^{(1)}$ at depth d_1 for $i = 1, \ldots, l$. Each vertex $v_i^{(1)}$ has degree $\Delta + 1$ and is the root of a subtree T_i. There are Δ paths connected to every $v_i^{(1)}$ whose length will be determined by the algorithm. Furthermore, depending on the algorithm, there may exist a vertex $v_i^{(2)}$ at depth d_2 that has degree $\Delta + 1$ and also Δ paths connected to it whose length will be determined by the algorithm. We call the subtrees with root $v_i^{(1)}$ and $v_i^{(2)}$ **adaptive subtrees** as they depend on the behavior of the online exploration algorithm. We further assume that B, d_1, d_2 are even and

$$d_1 + \Delta < d_2 \leq \tfrac{5}{3} \cdot d_1 \quad \text{and} \quad 3 \cdot d_1 < B \leq d_1 + 2 \cdot d_2. \tag{3.6}$$

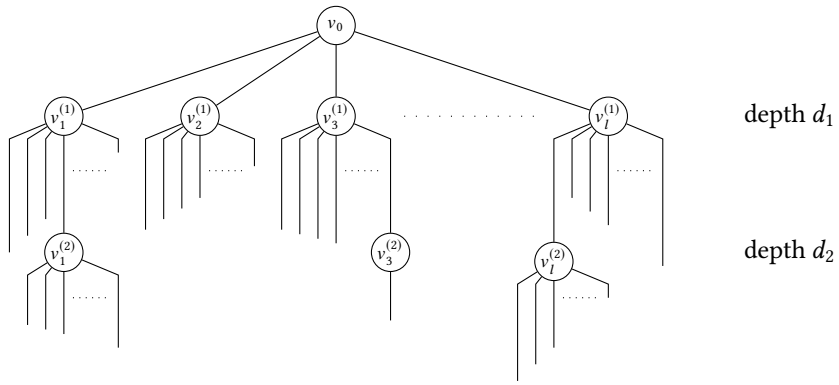


Figure 3.3: Tree for the lower bound of 2.17 on the competitive ratio.

Each of the adaptive trees can be **active**, i.e., as soon as an agent visits an unexplored vertex on a path another unexplored neighbor is presented, or **passive**, i.e., all unexplored vertices in the adaptive tree are leaves. Moreover, every subtree T_i has a **budget** N_i, which limits the total number of non-leaf vertices that are presented to the algorithm, i.e., if N_i vertices that are not leaves have been explored in T_i both adaptive trees in T_i become passive and from now on all unexplored vertices in T_i are leaves. The budget N_i is initially 2 and is increased as described below when agents enter the subtree T_i. Initially every subtree T_i has an active adaptive subtree below $v_i^{(1)}$. We now present new vertices to the algorithm in every subtree T_i for $i \in 1, \ldots, l$ according to the following rules:

I. *When the first agent A_1 that has not visited any other tree $T_j \neq T_i$ before enters T_i for the first time:*
The budget N_i of T_i is increased by $(B + d_2)/2 - d_1 + 2\Delta$, the adaptive tree below $v_i^{(1)}$ is active and $v_i^{(2)}$ has not been discovered. The first vertex at depth d_2 discovered by A_1 is $v_i^{(2)}$, i.e., it has degree $\Delta + 1$ and is the root of another adaptive tree which is active. Additionally, if A_1 explores a new vertex v at depth $d > d_2$ in T_i (below $v_i^{(2)}$ or on any branch below $v_i^{(1)}$) and the remaining energy of A_1 is $\leq d - d_2$, then we stop presenting new vertices on the current path of A_1, i.e., v is a vertex without further unexplored neighbors.

II. *When the second agent A_2 that has not visited any other tree $T_j \neq T_i$ before enters T_i for the first time:*

   (a) *If A_1 has explored at most $(d_1 + d_2)/2$ vertices in T_i:*
   The adaptive trees both at $v_i^{(1)}$ and at $v_i^{(2)}$ become passive. In all following cases below, we assume that A_1 explored more than $(d_1 + d_2)/2$ vertices in T_i.

   (b) *If A_1 has explored the vertex $v_i^{(2)}$ or still has enough energy left to reach a vertex v at depth d_2 via an unexplored vertex:*
   If $v_i^{(2)}$ has been discovered, the adaptive tree at $v_i^{(1)}$ becomes passive, but the adaptive tree at $v_i^{(2)}$ remains active. If A_1 has not visited a vertex at depth d_2, then the adaptive tree at $v_i^{(1)}$ becomes passive except for the path via an unexplored vertex to $v_i^{(2)} := v$ at

depth d_2, which A_1 can reach with its remaining energy. From now on, if any agent A is at depth $d > d_2$, then we stop presenting new vertices on the current path of A as soon as the remaining energy is $\leq d - d_2$.

(c) *If A_1 has not visited a vertex at depth d_2 and has not enough energy to reach a vertex at depth d_2 via an unexplored vertex:*
From now on if any agent A is at depth $d > d_1$, we stop presenting new vertices on the current path of A if the remaining energy of A is $\leq d - d_1$.

III. *Whenever an agent A which before has visited a tree $T_j \neq T_i$ enters T_i for the first time with remaining energy B_A:*
The budget N_i of T_i is increased by $B_A/2+2$. If A discovers a vertex v below $v_i^{(2)}$ at depth $d > d_2$ and the remaining energy of A is $\leq d - d_2$, then we stop presenting new vertices on this path. Similarly, if A discovers a vertex v below $v_i^{(1)}$ at depth $d > d_1$ (but not on a branch containing $v_i^{(2)}$) and the remaining energy of A is $\leq d - d_1$, then we also stop presenting new vertices on that path.

Note that in every tree T_i, if Case II (b) does not occur in T_i, $v_i^{(2)}$ and the adaptive subtree below $v_i^{(2)}$ exist if and only if A_1 discovers a vertex v at depth d_2.

3.3.2 Intuition and Proof of the Lower Bound in Special Cases

In this subsection, we want to give some intuition about our construction by looking at two special cases and making some simplifying assumptions, which do not hold in general. The adaptive trees are constructed in a way that a path ends exactly when the agent currently exploring that path has just enough energy to return to $v_i^{(1)}$ or $v_i^{(2)}$ respectively. So let us make the simplifying assumption that the final position of every agent is either at $v_i^{(1)}$ or $v_i^{(2)}$ for some $i \in \{1, \dots, l\}$. The online algorithm has to balance between sending each agent to only one subtree T_i to completely explore it or to move to a second subtree T_j later to explore more vertices which are close to the root v_0. We will consider instances with increasing values of B and l in such a way that $l = o(B)$. Note that this implies that $\Delta = o(B)$.

Let us consider the special case that the algorithm first sends one agent to each of the subtrees $T_1, \dots, T_l$ and then a second agent to every subtree except T_1 (there are $2l-1$ agents and l subtrees). For the sake of simplification, assume that A_1 visits $v_i^{(2)}$ and Case II (b) occurs in each subtree T_i when the second agent A_2 enters T_i. Note that in this case, A_1 cannot visit another subtree as it visits $v_i^{(2)}$ at depth d_2 and $2d_2 + d_1 \geq B$ by (3.6). We further assume that for each subtree T_i, $2 \leq i \leq l$, either the second agent A_2 entering T_i helps A_1 to explore T_i completely, or it goes to T_1 to explore new vertices.

The first agent A_1 in each subtree T_i can explore at most $(B+d_2)/2$ vertices in T if its final position is at $v_i^{(2)}$ (it traverses at most d_2 edges once and all other edges are traversed an even number of times) and less vertices if its final position is at $v_i^{(1)}$. Note that $d_1 - 2$ of the vertices explored by A_1 are on the path from v_0 to $v_i^{(1)}$ and thus A_1 can only explore at most $(B + d_2)/2 - d_1 + 2$ vertices in T_i. But by

construction the budget N_i is increased by $(B + d_2)/2 - d_1 + 2\Delta$ when A_1 enters T_i so that A_1 alone cannot deplete the whole budget and completely explore T_i.

As the subtree below $v_i^{(1)}$ becomes passive when A_2 enters T_i, A_2 can only explore at most Δ vertices that are not below $v_i^{(2)}$. Therefore if A_1 and A_2 completely explore T_i, A_2 has to go to depth d_2 and then it cannot visit any other subtree as $2d_2 + d_1 \geq B$ by (3.6). In this case, agents A_1 and A_2 together then explore at most N_i vertices in T_i plus at most 2Δ leaves and the path of length d_1 leading to T_i, i.e., they explore at most $(B + d_2)/2 + 4\Delta + 2 = (B + d_2)/2 + o(B)$ vertices.

Suppose now that A_1 and A_2 do not completely explore the subtree T_i and that A_2 goes to T_1 to explore new vertices after having visited T_i. Assume that A_2 has B_{A_2} energy left when it enters T_1, and note that $B_{A_2} \leq (B - 3d_1)/2$ since A_2 went first to T_i before entering T_1. Agent A_2 can explore at most $B_{A_2}/2$ new vertices in T_1 if its final position is in $v_i^{(1)}$ (every edge it traverses in T_1 is traversed an even number of times) and less vertices if its final position is in $v_i^{(2)}$ (since the vertices on the branch from $v_i^{(1)}$ to $v_i^{(2)}$ have already been explored). Note that when A_2 enters T_1, the budget N_1 of T_1 is increased by $B_{A_2}/2 + 2$ and thus the budget of T_1 is never depleted. As A_2 has B_{A_2} energy left when it enters T_1 and spends $3d_1$ energy to first reach T_i and then T_1, it can have explored at most $(B - 3d_1 - B_{A_2})/2$ vertices in T_i because A_2 traverses every edge in T_i an even number of times. Overall, A_2 thus explores at most $(B - 3d_1)/2$ new vertices and A_1 at most $(B + d_2)/2$ vertices in this case.

Recall that for sake of simplification, we consider only two strategies for the online algorithm Alg: either in every tree T_i, $2 \leq i \leq l$, A_1 and A_2 completely explore T_i, or for every tree T_i, $2 \leq i \leq l$, the second agent A_2 entering T_i also visits T_1 (and T_i is not completely explored by the algorithm). In the first case, the algorithm explores at most $l \cdot (B + d_2)/2 + o(lB)$ vertices. In the second case, the algorithm explores at most $l \cdot ((B + d_2)/2 + (B - 3d_1)/2) + o(lB)$ vertices.

Let us now consider an optimal offline algorithm Opt. Whatever the strategy of Alg is, one can show that there is always an unexplored vertex u_1 at depth at most $d_1 + \Delta$ in T_1 (this is proved in Lemma 3.4 (f)). We can assume that u_1 has degree $2l$ and there are $2l - 1$ distinct paths of length B connected to it.

If Alg completely explores every tree T_i, $2 \leq i \leq l$, then Opt can send all agents to u_1 and then each agent explores one of the paths below u_1. In this case, Opt explores at least $B + (2l - 2) \cdot (B - d_1 - \Delta) = 2l \cdot (B - d_1) - o(lB)$ vertices.

If Alg does not completely explore any T_i, $2 \leq i \leq l$, then there exists an unexplored vertex u_i in each tree T_i, $2 \leq i \leq l$, and we can assume that there is a path of length B connected to it. In this case, Opt can send an agent to each u_i, $2 \leq i \leq l$ that can then explore the path below u_i. Then, Opt can send the remaining l agents to u_1 as in the previous case, and each of these agent explores one of the paths below u_1. In this case, Opt explores at least $lB + (l - 1) \cdot (B - d_1 - \Delta) = l \cdot (2B - d_1) - o(lB)$ vertices.

As the algorithm can choose the best strategy among the two, we get for our constructed in-

stance I that

$$\frac{\textsc{Opt}(I)}{\textsc{Alg}(I)} \geq \min\left\{\frac{4l\cdot(B-d_1)-o(lB)}{l\cdot(B+d_2)+o(lB)}, \frac{2l\cdot(2B-d_1)-o(lB)}{l\cdot(2B+d_2-3d_1)+o(lB)}\right\}.$$

In order to maximize the competitive ratio, we want to choose d_2 as small as possible. Because of the initial assumptions on the parameter in (3.6), we must have $2d_2 + d_1 \geq B$ and thus we choose $d_2 = (B - d_1)/2$. Additionally, dividing by l and omitting the terms that vanish as B tends to infinity, we obtain

$$\frac{\textsc{Opt}(I)}{\textsc{Alg}(I)} \geq \lim_{B\to\infty} \min\left\{\frac{8B-8d_1}{3B-d_1}, \frac{8B-4d_1}{5B-7d_1}\right\}.$$

By standard calculus, the competitive ratio is maximized when the two terms on the right-hand side are equal and this is true when $d_1 = (19-3\sqrt{17})B/26$. These choices of d_1 and d_2 satisfy (3.6) and the above lower bound evaluates to $(5+3\sqrt{17})/8 \approx 2.17$.

We made several simplifying assumptions to get to this bound, but one can show that no other strategy can beat the lower bound we established. The challenge in the analysis is that the online algorithm does not necessarily use one agent after the other, but the agents may wait in between. This creates many different cases which need to be grouped and analyzed.

3.3.3 Proof of the Lower Bound for the General Case

In this subsection, we give a complete proof of the lower bound on the competitive ratio of an arbitrary online algorithm Alg using the construction introduced in Section 3.3.1.

For every vertex v in T, we say that v is **explored** by an agent A, if A is the first agent visiting v. If $v_i^{(2)}$ is defined, then we say that every vertex on the path from $v_i^{(1)}$ to $v_i^{(2)}$ is explored by the first agent A_1, which enters T_i and has not visited any other tree $T_j \neq T_i$ before. It may be even the case that A_1 never visits these vertices, but to simplify the analysis, we will still attribute them to A_1.

For $i \in \{1,\dots,l\}$, we let $\mathcal{A}_{1,i}$ be the set of agents for which T_i is the first tree they visit and let $\mathcal{A}_{2,i}$ be the set of agents for which T_i is the second tree they visit, i.e., every agent $A \in \mathcal{A}_{2,i}$ has visited a subtree distinct from T_i before. Note that an agent can visit at most two subtrees as

$$5\cdot d_1 \geq d_1 + 4\cdot\tfrac{3}{5}d_2 > d_1 + 2\cdot d_2 \geq B \tag{3.7}$$

by our assumptions on the parameters in (3.6). Therefore an agent $A \in \mathcal{A}$ can be contained in one set $\mathcal{A}_{1,i}$ and possible in some other set $\mathcal{A}_{2,j}$ for $j \in \{1,\dots,l\}\setminus\{i\}$. For every agent $A \in \mathcal{A}$ we let B_A denote the remaining energy when A enters a second subtree. If A only enters at most one of the subtrees $T_1,\dots,T_l$, we set $B_A = 0$. We now establish the following important properties for the number of vertices that the agents explore.

Lemma 3.4. *Let T_i be a subtree of T as defined above.*

(a) $B_A \leq B - 3d_1$ for all $A \in \mathcal{A}$.

(b) If Case II (b) or Case II (c) occurs, then the first agent A_1 in $\mathcal{A}_{1,i}$ entering T_i does not visit any other subtree, i.e., $B_{A_1} = 0$.

(c) *Every agent $A \in \mathcal{A}_{2,i}$ explores at most $B_A/2 + 2$ vertices in T_i.*

(d) *The first agent A_1 in $\mathcal{A}_{1,i}$ entering T_i explores at most $(B + d_2)/2 - d_1 + 2\Delta$ vertices.*

(e) *If $|\mathcal{A}_{1,i}| \leq 1$, then the agents in $\mathcal{A}_{1,i} \cup \mathcal{A}_{2,i}$ visit strictly less than N_i vertices in T_i.*

(f) *If the adaptive tree below $v_i^{(1)}$ is active and the budget N_i is not depleted, then there is an unexplored vertex in T_i at depth at most $d_1 + \Delta$.*

Proof. (a) Note that we have $B - 3d_1 > 0$ by our initial assumptions on the parameters in (3.6) and thus the claim trivially holds if A visits at most one of the subtrees $T_1, \dots, T_l$, i.e., if $B_A = 0$. Now, consider an agent $A \in \mathcal{A}$ visiting two subtrees and assume without loss of generality, that A first visits T_1 and afterwards enters T_2 with remaining energy B_A. To reach T_1 the agent needs to traverse d_1 edges. In order to afterwards reach T_2, the agent A needs to traverse another $2d_1$ edges. Thus, we must have $B_A \leq B - 3d_1$.

(b) In both cases, agent A_1 has explored more than $(d_1 + d_2)/2$ vertices in T_i. If A_1 visits another subtree it traverses every edge in T_i an even number of times and therefore needs at least $d_1 + d_2$ energy to explore more than $(d_1 + d_2)/2$ vertices. Moreover, $3d_1$ energy is needed to first reach T_i and then another subtree. As $3d_1 + (d_1 + d_2) > 5d_1 \geq B$ by (3.6) and (3.7), A_1 cannot visit another subtree.

(c) By definition, the remaining energy of the agent A when entering T_i is B_A. If the final position of A is not in T_i, then it traverses every edge in T_i an even number of times and in particular A traverses at most $B_A/2$ edges in T_i. These can be incident to at most $B_A/2 + 1$ vertices, which yields the claim.

Now, consider the case that the final position of A is below $v_i^{(1)}$ and not below $v_i^{(2)}$ and not on the path between $v_i^{(1)}$ and $v_i^{(2)}$. This means that at some point A must have visited a vertex v at depth d with remaining energy exactly $d - d_1$. Recall that B and d_1 are even, hence B_A is even and this must happen at some point. Then A has exactly enough energy left to move to $v_i^{(1)}$ and, in particular, A cannot reach any other path below $v_i^{(1)}$. If v is explored by A, then v has no new unexplored neighbor and we can simply assume that A returns to $v_i^{(1)}$ as this does not change the number of neighbors it explores. In this case A has traversed every edge in T_i an even number of times and therefore can have explored at most $B_A/2 + 1$ vertices. If v is not explored by A, then A can only explore at most one more vertex after visiting v with energy $d - d_1$, because the current path ends immediately when A explores a new vertex. Compared to the case that v is explored by A, agent A only explores at most one additional vertex in this case so that we can bound the total number of vertices explored by A by $B_A/2 + 2$.

Next consider the case that the final position of A is on the path between $v_i^{(1)}$ and $v_i^{(2)}$. In particular, this implies that $v_i^{(2)}$ is defined and all vertices on the path between $v_i^{(1)}$ and $v_i^{(2)}$ are attributed to A_1. Note that then all edges that are not on that path, must be traversed an even number of times by A and we therefore again obtain that A can explore at most $B_A/2+1$ vertices, which yields the claim.

Finally, the case the final position of A is below $v_i^{(2)}$ is completely analogous to the case that the final position is below $v_i^{(1)}$ as all vertices on the path from $v_i^{(1)}$ to $v_i^{(2)}$ are attributed to A_1.

(d) Let A_1 be the first agent entering T_i. If A_1 visits another subtree $T_j \neq T_i$ afterwards, then A_1 traverses every edge in T_i an even number of times and needs $3d_1$ energy to first reach T_i and afterwards T_j. Overall, A_1 can therefore explore at most $(B - 3d_1)/2$ vertices in T_i and as $(B + d_2)/2 - d_1 + 2\Delta \geq (B - 3d_1)/2$ this yields the claim.

From now on, we can therefore assume that A_1 only visits the subtree T_i. The energy that A_1 spends in T_i is at most $B - d_1$, as $B - d_1$ is the maximum energy possible when entering T_i. If the final position of the agent A_1 is at depth d_2 or above, then it traverses at most $d_2 - d_1$ edges in T_i once using $d_2 - d_1$ energy and exploring at most $d_2 - d_1 + 1$ vertices. All other edges in T_i traversed by A_1 must be traversed at least twice which means there is at most one explored vertex for every two energy used. Overall, the number of explored vertices is thus bounded by

$$(d_2 - d_1 + 1) + \frac{B - d_1 - (d_2 - d_1)}{2} = \frac{B + d_2}{2} - d_1 + 1,$$

if the final position of A_1 is at depth d_2 or above. If the final position of A_1 is below d_2, there has to be a vertex v at depth d visited by A_1 such that the remaining energy of A_1 when visiting v is exactly $d - d_2$ (recall that d_2 and B are even by assumption). If v is explored by A_1, then v is the last vertex that A_1 explores because v then is a vertex without further neighbors and A_1 cannot reach another path below $v_i^{(1)}$ or $v_i^{(2)}$. If v has been already explored by another agent, then A_1 can only explore one more additional vertex as the path also ends immediately if A_1 explores a vertex. If A_1 after visiting v with remaining energy $d - d_2$, would directly move up towards $v_i^{(1)}$, its final position would be at depth d_2 and by the argument above A_1 could explore at most $(B + d_2)/2 - d_1 + 1$ vertices. As A_1 can explore only at most one more vertex, as we just showed, the total number of vertices explored by A_1 is bounded by $(B + d_2)/2 - d_1 + 2$ in this case.

However, in Case II (b), it can happen that $v_i^{(2)}$ is defined as it can be reached by A_1 with its remaining energy when A_2 enters T_i, but A_1 does not visit $v_i^{(2)}$. Recall that we always attribute the vertices on the path between $v_i^{(1)}$ and $v_i^{(2)}$ to A_1, even if A_1 never visits them. If A_1 visits $v_i^{(2)}$, then it visits all vertices on the path between $v_i^{(1)}$ and $v_i^{(2)}$ and by the argument above the number of vertices visited by A_1 is bounded by $(B + d_2)/2 - d_1 + 2$. As the adaptive tree at $v_i^{(1)}$ becomes passive when A_2 enters T_i, A_1 can from then on only explore Δ vertices which are not on the path between $v_i^{(1)}$ and $v_i^{(2)}$ or below $v_i^{(2)}$. This means compared to the case that A_1 visits $v_i^{(2)}$, A_1 can only visit additional Δ vertices and therefore the overall number of vertices explored by A_1 is bounded by $(B + d_2)/2 - d_1 + 2\Delta$ in this case as $2 + \Delta \leq 2\Delta$. This yields the claim.

(e) By Lemma 3.4 (c), every agent $A \in \mathcal{A}_{2,i}$ entering T_i explores at most $B_A/2 + 2$ vertices and the budget N_i is also increased by this value when A enters T_i. Thus, if $\mathcal{A}_{1,i} = \emptyset$, the number of vertices explored in T_i will always be less than the budget, as N_i is initially 2. Now assume, there is one agent $A_1 \in \mathcal{A}_{1,i}$ entering T_i. By Case I in the construction of the lower bound, the

budget N_i is increased by $(B + d_2)/2 - d_1 + 2\Delta$ and by Lemma 3.4 (d), A_1 also explores at most $(B + d_2)/2 - d_1 + 2\Delta$ vertices in T_i. Thus the budget N_i, which is initially 2, is also larger than the number of explored vertices in T_i in this case.

(f) Suppose, for the sake of contradiction, that the budget N_i is not depleted and the adaptive tree below $v_i^{(1)}$ is active, but there is no unexplored vertex at depth at most $d_1 + \Delta$ in T_i. Recall that there are Δ path below $v_i^{(1)}$ and $\Delta = \left\lceil \sqrt{2 \cdot l \cdot B} \right\rceil + 2l$. We have $2l-1$ agents and each agent can be responsible for at most one path to be fully explored and end because the agent has remaining energy $\leq d - d_1$ at depth d. If all other $\left\lceil \sqrt{2 \cdot l \cdot B} \right\rceil + 1$ paths are fully explored up to depth Δ, then these path contain at least $\Delta \cdot \left\lceil \sqrt{2 \cdot l \cdot B} \right\rceil \geq 2 \cdot l \cdot B$ vertices. But all agents together only have $(2 \cdot l - 1) \cdot B$ energy and hence cannot visit all these vertices. This is a contradiction.

□

We will say that Case II (a) occurs in T_i if $|\mathcal{A}_{1,i}| \geq 2$ and Case II (a) occurs when the second agent $A_2 \in \mathcal{A}_{1,i}$ enters T_i. Analogously for Case II (b) and Case II (c). We partition the subtrees into the following three sets:

$M_0 := \{i \mid B_A > 0$ for all $A \in \mathcal{A}_{1,i}$ or Case II (a) occurs in $T_i\}$,

$M_1 := \{i \mid T_i$ is not completely explored, $\exists A \in A_{1,i}$ with $B_A = 0$ and Case II (a) does not occur$\}$,

$M_2 := \{i \mid T_i$ is completely explored and Case II (b) or Case II (c) occurs in $T_i\}$.

Lemma 3.5. *Let T_i be a subtree of T, $|T_i|$ be the number of vertices explored in T_i by ALG and M_0, M_1 and M_2 as defined above.*

(a) We have $M_0 \cup M_1 \cup M_2 = \{1, \ldots, l\}$ and $M_i \cap M_j = \emptyset$ for all $i, j \in \{0, 1, 2\}$ with $i \neq j$.

(b) For every $i = 1, \ldots, l$, we have

$$|T_i| \leq \frac{B + d_2}{2} - d_1 + 6\Delta + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2}. \tag{3.8}$$

(c) If $i \in M_0$, then

$$|T_i| \leq \frac{B + d_2}{2} - d_1 + 4\Delta + (|\mathcal{A}_{1,i}| - 2) \cdot \frac{B - 3d_1}{2} + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2} - \sum_{A \in \mathcal{A}_{1,i}} \frac{B_A}{2} \tag{3.9}$$

(d) If $i \in M_1$, then

$$\sum_{A \in \mathcal{A}_{1,i}} B_A \leq (|\mathcal{A}_{1,i}| - 1) \cdot (B - 3d_1). \tag{3.10}$$

(e) If $i \in M_2$, then

$$\sum_{A \in \mathcal{A}_{1,i}} B_A \leq (|\mathcal{A}_{1,i}| - 2) \cdot (B - 3d_1). \tag{3.11}$$

Proof. (a) For the first part of the statement, let $i \in \{1, \dots, l\} \setminus (M_0 \cup M_1\}$, and note that there exists $A \in \mathcal{A}_{1,i}$ with $B_A = 0$, Case II (a) does not occur in T_i, and T_i is completely explored. By Lemma 3.4 (e) and Lemma 3.4 (f), we have $|\mathcal{A}_{1,i}| \geq 2$. Consequently, since Case II (a) does not occur in T_i, necessarily Case II (b) or Case II (c) occurs in T_i and $i \in M_2$.

We obviously have $M_0 \cap M_1 = \emptyset$ and $M_1 \cap M_2 = \emptyset$. By Lemma 3.4 (b), $B_{A_1} = 0$ if Case II (b) or Case II (c) occurs and thus also $M_0 \cap M_2 = \emptyset$.

(b) The budget N_i of the tree T_i, which is initially 2, satisfies

$$N_i \leq 2 + \frac{B + d_2}{2} - d_1 + 2\Delta + \sum_{A \in \mathcal{A}_{2,i}} \left(\frac{B_A}{2} + 2 \right) \leq \frac{B + d_2}{2} - d_1 + 4\Delta + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2},$$

where we used $2 + 2|\mathcal{A}_{2,i}| \leq 4l + 2 \leq 2\Delta$. Since T_i has at most $2\Delta - 1$ leaves, and since the number of vertices explored in T_i, which are not leaves, is at most N_i, we have $|T_i| \leq N_i + 2\Delta$. This yields the claim.

(c) First we show the claim for the case that $B_A > 0$ for all $A \in \mathcal{A}_{1,i}$. This means that every agent $A \in \mathcal{A}_{1,i}$ also visits a second subtree. As $3d_1$ energy is spent to reach T_i and afterwards the second subtree and A has still B_A energy left when entering the second subtree, at most $B - 3d_1 - B_A$ energy is spent in T_i. As every edge in T_i is traversed an even number of times, at most $(B - 3d_1 - B_A)/2$ vertices are explored by A in T_i for all $A \in \mathcal{A}_{1,i}$. Moreover, every agent $A \in \mathcal{A}_{2,i}$ explores at most $B_A/2 + 2$ vertices in T_i by Lemma 3.4. Additionally using $2|\mathcal{A}_{2,i}| \leq 2\Delta$, we thus have

$$|T_i| \leq \sum_{A \in \mathcal{A}_{1,i}} \frac{B - 3d_1 - B_A}{2} + \sum_{A \in \mathcal{A}_{2,i}} \left(\frac{B_A}{2} + 2 \right) = |\mathcal{A}_{1,i}| \cdot \frac{B - 3d_1}{2} + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2} - \sum_{A \in \mathcal{A}_{1,i}} \frac{B_A}{2} + 2\Delta.$$

We obtain the claim using $(B + d_2)/2 - d_1 \geq 2 \cdot (B - 3d_1)/2$ as $d_2 > d_1$ and $5d_1 > B$ by (3.6) and (3.7).

Now assume Case II (a) occurs and let $A_1 \in \mathcal{A}_{1,i}$ be the first agent entering T_i and $A_2 \in \mathcal{A}_{1,i}$ the second agent entering T_i. As Case II (a) occurs, A_1 explores at most $(d_1 + d_2)/2$ vertices in T_i. If $B_{A_1} > 0$, i.e., A_1 also enters a second tree, we can even bound the number of vertices explored by A_1 in T_i by $(B - 3d_1 - B_{A_1})/2$. We have $(d_1 + d_2)/2 > (B - 3d_1)/2$ as $d_2 > d_1$ and $5d_1 > B$ by (3.6) and (3.7). Therefore, we can both for $B_{A_1} = 0$ and for $B_{A_1} > 0$ bound the number of vertices explored by A_1 until A_2 enters T_i by $(d_1 + d_2 - B_{A_1})/2$. As soon as A_2 enters T_i all agents together can only explore the unexplored leaves, i.e., at most 2Δ vertices. Moreover, every agent $A \in \mathcal{A}_{2,i}$ explores at most $B_A/2+2$ vertices in T_i by Lemma 3.4. Overall, we hence have

$$|T_i| \leq \frac{d_1 + d_2 - B_{A_1}}{2} + 2\Delta + \sum_{A \in \mathcal{A}_{2,i}} \left(\frac{B_A}{2} + 2 \right) \leq \frac{d_1 + d_2 - B_{A_1}}{2} + 4\Delta + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2},$$

where we again used $2|\mathcal{A}_{2,i}| \leq 2\Delta$. We also have $0 \leq B - 3d_1 - B_A$ for all $A \in \mathcal{A}_{1,i}$ by Lemma 3.4

and obtain

$$\begin{aligned}|T_i| &\leq \frac{d_1 + d_2 - B_{A_1}}{2} + 4\Delta + \sum_{A \in \mathcal{A}_{1,i} \setminus \{A_1\}} \frac{B - 3d_1 - B_A}{2} + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2} \\ &= \frac{d_1 + d_2}{2} + 4\Delta + (|\mathcal{A}_{1,i}| - 1) \cdot \frac{B - 3d_1}{2} - \sum_{A \in \mathcal{A}_{1,i}} \frac{B_A}{2} + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2} \\ &= \frac{B + d_2}{2} - d_1 + 4\Delta + (|\mathcal{A}_{1,i}| - 2) \cdot \frac{B - 3d_1}{2} - \sum_{A \in \mathcal{A}_{1,i}} \frac{B_A}{2} + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2}.\end{aligned}$$

(d) The bound follows directly from the fact that $B_A = 0$ for some $A \in \mathcal{A}_{1,i}$ and $B_A \leq B - 3d_1$ for all $A \in \mathcal{A}_{1,i}$ by Lemma 3.4.

(e) In order to show the bound (3.11), we proceed along the following key claims:

(i) The bound (3.11) follows, if the set of agents $\mathcal{A}_{1,i} \setminus \{A_1\}$ together visit at least $(B - 3d_1)/2$ distinct vertices in T_i or if there is an agent in $\mathcal{A}_{1,i} \setminus \{A_1\}$ that does not visit another subtree.

(ii) The bound (3.11) holds if Case II (b) occurs.

(iii) For Case II (c), the agents in $\left(\mathcal{A}_{i,1} \setminus \{A_1\}\right) \cup \mathcal{A}_{i,2}$ need to visit at least $(B - 3d_1) + \sum_{A \in \mathcal{A}_{i,2}} (B_A/2 + 2)$ vertices in T_i for T_i to be completely explored. Some of these vertices may have already been explored by agent A_1.

(iv) Let V_1 be the set of vertices visited by A_1. Further let e_2 be the number of vertices explored by the agents in $\mathcal{A}_{i,2}$ that are not contained in V_1 and n_2 be the total number of vertices visited by the agents in $\mathcal{A}_{i,2}$ that are contained in V_1. Then it holds that $e_2 + n_2/2 \leq \sum_{A \in \mathcal{A}_{i,2}} (B_A/2 + 2)$.

(v) The claims (iii) and (iv) yield the bound (3.11) if Case II (c) occurs.

We now show each of the above claims.

(i) By Lemma 3.4 (b), we know that A_1 cannot visit another subtree, i.e., $B_{A_1} = 0$, as Case II (b) or Case II (c) occurs when A_2 enters T_i. If there exists another agent $A' \in \mathcal{A}_{1,i}$ such that $B_{A'} = 0$, then the claim follows directly from the fact that $B_A \leq B - 3d_1$ for all $A \in \mathcal{A}_{1,i} \setminus \{A_1, A'\}$ by Lemma 3.4. So assume that for every $A \in \mathcal{A}_{1,i} \setminus \{A_1\}$, $B_A > 0$ holds, i.e., every agent in $\mathcal{A}_{1,i} \setminus \{A_1\}$ visits two subtrees and the agents in $\mathcal{A}_{1,i} \setminus \{A_1\}$ together visit at least $(B - 3d_1)/2$ distinct vertices in T_i. As every agent A in $\mathcal{A}_{1,i} \setminus \{A_1\}$ visits a distinct subtree after T_i, A traverses every edge in T_i an even number of times. Thus at least $B - 3d_1$ energy is needed to visit $(B - 3d_1)/2$ distinct vertices. But then we already have

$$\sum_{A \in \mathcal{A}_{1,i} \setminus \{A_1\}} B_A \leq (|\mathcal{A}_{1,i}| - 1) \cdot (B - 3d_1),$$

as every agents spends an additional $3d_1$ energy to first reach T_i and then the second subtree. This implies (3.11).

(ii) The budget of T_i is increased by $(B + d_2)/2 - d_1 + 2\Delta$ when A_1 enters T_i, but this is also the maximum number of vertices that A_1 can explore by Lemma 3.4. Similarly, for every agent $A \in \mathcal{A}_{2,i}$ the budget is increased by $B_A/2+2$ and the agent can also explore at most $B_A/2 + 2$ vertices by Lemma 3.4. Note that when A_2 enters T_i, the adaptive tree rooted at $v_i^{(1)}$ becomes passive, and thus agents not entering $v_i^{(2)}$ can collectively explore at most Δ vertices after A_2 entered T_i. We claim that if no agent from $\mathcal{A}_{1,i} \setminus \{A_1\}$ enters $v_i^{(2)}$, then T_i cannot be explored. Indeed, there are Δ paths starting from $v_i^{(1)}$ and Δ paths starting from $v_i^{(2)}$. When the budget N_i is depleted, the agents must have explored N_i vertices that are not leaves, and consequently, $|T_i| \geq N_i + 2\Delta$. Since the agents from $\mathcal{A}_{2,i} \cup \{A_1\}$ can explore at most $N_i - 2$ vertices, the agents from $\mathcal{A}_{1,i} \setminus \{A_1\}$ have to explore at least $2\Delta + 2$ vertices in T_i. Consequently, at least one agent A' from $\mathcal{A}_{1,i} \setminus \{A_1\}$ has to visit $v_i^{(2)}$ and thus $B_{A'} = 0$ as $d_1 + 2d_2 \geq B$ by (3.6). By (i), this yields (3.11).

(iii) As Case II (c) occurs when A_2 enters T_i, agent A_1 has not enough energy to reach a vertex at depth d_2 via an unexplored vertex. We first show that then A_1 never visits a vertex at depth d_2+1 (it is clear by assumption that A_1 never explores a vertex at depth d_2 or below, but A_1 could still visit a vertex at depth $d_2 + 1$ on a path that was explored by another agent). If any agent A from $\mathcal{A}_{i,2}$ explores a vertex v at depth d_2 in T_i, then it must have spend at least $2d_1$ energy to reach the tree it visited before T_i and then come back to the root and another d_2 energy to reach v. We have $B - 2d_1 - d_2 \leq d_2 - d_1$ as $d_1 + 2d_2 \geq B$ by (3.6). Thus A has at most $d_2 - d_1$ energy left when it visits v at depth d_2 and the path of A ends by Case III in the construction of the lower bound. Therefore, A_1 cannot reach any vertex at depth $d_2 + 1$ on a path that was explored by an agent from $\mathcal{A}_{i,2}$ as this path ends at depth d_2 at the latest. Agent A_1 also cannot visit a vertex at depth $d_2 + 1$ that was explored by an agent in $\left(\mathcal{A}_{i,1} \setminus \{A_1\}\right)$ as this vertex would be unexplored at the time A_2 enters T_i and we assume that at this point A_1 cannot reach an unexplored vertex at depth d_2.

This means that A_1 never visits any vertex at depth d_2+1 and can therefore only completely explore one path below $v_i^{(1)}$ containing at most d_2-d_1+1 vertices. All other vertices visited by A_1 that are not on that path have to be visited by other agents since otherwise there is an unexplored vertex at the end of that path. For T_i to be completely explored, the budget N_i must be completely depleted as otherwise the adaptive tree below $v_i^{(1)}$ remains active and there is an unexplored vertex in T_i by Lemma 3.4 (f). Thus all N_i vertices, except for at most $d_2 - d_1 + 1$, need to be visited by the agents in $\left(\mathcal{A}_{i,1} \setminus \{A_1\}\right) \cup \mathcal{A}_{i,2}$ for T_i to be completely explored. We have

$$N_i - (d_2 - d_1 + 1) \geq \frac{B - d_2}{2} + \sum_{A \in \mathcal{A}_{i,2}} \left(\frac{B_A}{2} + 2\right). \tag{3.12}$$

Using, $d_1 + 2d_2 \geq B$ and $d_2 \leq 5/3 \cdot d_1$ by (3.6), we obtain

$$2B - 6d_1 \leq (d_1 + 2d_2) + B - 6d_1 = 3d_2 - 5d_1 + (B - d_2) \leq B - d_2.$$

This implies $B - 3d_1 \leq (B - d_2)/2$ and together with (3.12) this yields the claim.

(iv) For an agent $A \in \mathcal{A}_{i,2}$, let e_A be the number of vertices in T_i that are explored by A and not visited by A_1. Moreover, let n_A be the number of moves performed by agent A in T_i increasing the distance from A to $v_i^{(1)}$ while visiting a new distinct vertex in V_1. We show that $e_A + n_A/2 \leq B_A/2 + 2$. The claim then follows by using $n_2 = \sum_{A \in \mathcal{A}_{i,2}} n_A$ and $e_2 = \sum_{A \in \mathcal{A}_{i,2}} e_A$.

Consider the last time an agent $A \in \mathcal{A}_{i,2}$ visits a vertex v at depth d and exactly has enough energy to move to $v_i^{(1)}$ (as B and d_1 are even, this will happen at some point). Note that A cannot reach any other path below $v_i^{(1)}$ and that it can explore at most one vertex as any unexplored vertex that A visits will have no further neighbor.

First, assume v is explored by A. By Case III in the construction of the lower bound, the current path ends and v is a vertex without further neighbors. We can now assume that A returns to $v_i^{(1)}$, as this does not change e_A or n_A. Then A has traversed every edge in T_i an even number of times and we have $e_A + n_A \leq B_A/2 + 1$ and thus in particular, $e_A + n_A/2 \leq B_A/2 + 1$ as $n_A \geq 0$.

Next, assume that v is not explored by A and also not visited by A_1. If A would return to $v_i^{(1)}$, then we can again argue that A traverses every edge an even number of times and obtain $e_A + n_A \leq B_A/2$ because now we even know that the edge traversal to v was neither an exploration move nor is v contained in V_1. On the other hand, if A does not return to $v_i^{(1)}$ from v then it cannot visit any new vertex in V_1 as A_1 never visits v and therefore also no vertex below v. Moreover, A can explore at most one additional vertex because then the current path will end immediately. Overall, we therefore again obtain $e_A + n_A \leq B_A/2 + 1$, which yields $e_A + n_A/2 \leq B_A/2 + 1$.

Finally, assume that v is not explored by A but visited by A_1. Let e'_A be the number of vertices not visited by A_1 and explored by A until the visit of v with remaining energy $d - d_1$ and analogously let n'_A be the number of moves performed by agent A up to that time increasing the distance from A to $v_i^{(1)}$ while visiting a new distinct vertex in V_1. If A would return to $v_i^{(1)}$ with its remaining energy, it would have traversed every edge an even number of times and we obtain $e'_A + n'_A \leq B_A/2 + 1$. After visiting v agent A can explore only at most one more vertex as then the path ends immediately. Thus, we have $e_A \leq e'_A + 1$. As v is visited by A_1, all vertices between v and $v_i^{(1)}$ must also be visited by A_1. Hence, it holds that $n'_A \geq d - d_1$. Moreover, after visiting v agent A only has $d - d_1$ energy left for visiting vertices in V_1 implying $n_A - n'_A \leq d - d_1$. Overall, this yields

$$e_A + \frac{n_A}{2} \leq e'_A + 1 + \frac{(d - d_1) + n'_A}{2} \leq e'_A + 1 + \frac{2n'_A}{2} \leq \frac{B_A}{2} + 2.$$

(v) Let n_1 be the total number of vertices in T_i visited by the agents in $\mathcal{A}_{i,1} \setminus \{A_1\}$. We assume $n_1 < (B - 3d_1)/2$ as otherwise the claim follows by (i). First of all, we must have $n_1 + e_2 \geq \sum_{A \in \mathcal{A}_{i,2}} (B_A/2 + 2)$ as T_i contains at least $N_i + \Delta$ vertices if it is completely explored of which $\sum_{A \in \mathcal{A}_{i,2}} (B_A/2 + 2)$ are not visited by A_1 by Lemma 3.4 (d). Using (iv),

this implies

$$(B - 3d_1)/2 > n_1 \geq \sum_{A \in \mathcal{A}_{i,2}} (B_A/2 + 2) - e_2 \geq n_2/2. \tag{3.13}$$

By (iii), we must further have

$$n_1 + n_2 + e_2 \geq B - 3d_1 + \sum_{A \in \mathcal{A}_{i,2}} (B_A/2 + 2) \tag{3.14}$$

for the budget N_i to be depleted and T_i completely explored. As we have $\sum_{A \in \mathcal{A}_{i,2}} (B_A/2 + 2) \geq n_2/2 + e_2$ by (iv), we obtain $n_1 + n_2/2 \geq B - 3d_1$ from (3.14). But this implies $n_1 \geq (B - 3d_1)/2$ as $n_2/2 < (B - 3d_1)/2$ by (3.13), which is a contradiction. □

Theorem 3.6. *There exists no c-competitive online exploration algorithm with $c < (5+3\sqrt{17})/8 \approx 2.17$.*

Proof. Let ALG be an online exploration algorithm and let I be the instance defined above, i.e., the tree T depending on ALG and the parameters l, d_1, d_2 and B. Assume t of the l subtrees $T_1, T_2 \ldots, T_l$ are completely explored and for $j \in \{1, 2, 3\}$ let $k_j := |\bigcup_{i \in M_j} \mathcal{A}_{1,i}|$.

We have $\text{ALG}(I) \leq l \cdot d_1 + \sum_{i=1}^{l} |T_i|$, as there are l paths with d_1 edges each connecting the root v_0 to every subtree. We now apply (3.8) from Lemma 3.5 for all subtrees T_i with $i \in M_1 \cup M_2$ and Inequality (3.9) for all subtrees T_i with $i \in M_0$ and additionally use that $\bigcup_{i=1}^{l} \mathcal{A}_{1,i} \supseteq \bigcup_{i=1}^{l} \mathcal{A}_{2,i}$. This yields

$$\begin{aligned}
\text{ALG}(I) &\leq l \cdot d_1 + \sum_{i=1}^{l} |T_i| \\
&\leq l \cdot d_1 + \sum_{i \in M_1 \cup M_2} \left(\frac{B + d_2}{2} - d_1 + 6\Delta + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2} \right) \\
&\quad + \sum_{i \in M_0} \left(\frac{B + d_2}{2} - d_1 + 4\Delta + (|\mathcal{A}_{1,i}| - 2) \cdot \frac{B - 3d_1}{2} + \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2} - \sum_{A \in \mathcal{A}_{1,i}} \frac{B_A}{2} \right) \\
&\leq l \cdot \left(\frac{B + d_2}{2} + 6\Delta \right) + \sum_{i=1}^{l} \sum_{A \in \mathcal{A}_{2,i}} \frac{B_A}{2} - \sum_{i \in M_0} \sum_{A \in \mathcal{A}_{1,i}} \frac{B_A}{2} + \sum_{i \in M_0} (|\mathcal{A}_{1,i}| - 2) \cdot \frac{B - 3d_1}{2} \\
&\leq l \cdot \left(\frac{B + d_2}{2} + 6\Delta \right) + (k_0 - 2|M_0|) \cdot \frac{B - 3d_1}{2} + \frac{1}{2} \sum_{i \in M_1 \cup M_2} \sum_{A \in \mathcal{A}_{1,i}} B_A.
\end{aligned}$$

Now we can apply the Inequalities (3.10) and (3.11). We further use $k_0 + k_1 + k_2 \leq k = 2l - 1$,

$|M_0| + |M_1| + |M_2| = l$, $t \leq |M_0| + |M_2|$ and obtain

$$\begin{aligned}
\textsc{Alg}(I) &\leq l\left(\frac{B+d_2}{2} + 6\Delta\right) + (k_0 - 2|M_0|) \cdot \frac{B-3d_1}{2} + \frac{1}{2}\sum_{i\in M_1}\sum_{A\in\mathcal{A}_{1,i}} (|\mathcal{A}_{1,i}| - 1)\cdot(B-3d_1) \\
&\quad + \frac{1}{2}\sum_{i\in M_2}\sum_{A\in\mathcal{A}_{1,i}} (|\mathcal{A}_{1,i}| - 2)\cdot(B-3d_1) \\
&\leq l\left(\frac{B+d_2}{2} + 6\Delta\right) + (k_0 + k_1 + k_2 - 2|M_0| - |M_1| - 2|M_2|)\frac{B-3d_1}{2} \\
&\leq l\left(\frac{B+d_2}{2} + 6\Delta\right) + (l-1-t)\frac{B-3d_1}{2}.
\end{aligned}$$

Next, we will give a lower bound on the number of vertices explored by an optimal offline algorithm Opt. As there are $2l-1$ agents and l subtrees, there has to be a subtree T_i with $|\mathcal{A}_{1,i}| \leq 1$. Without loss of generality let this subtree be T_1. By Lemma 3.4 the subtree T_1 then has an unexplored vertex u_1 at depth at most $d_1 + \Delta$ and, in particular, is not completely explored, implying $t < l$.

For every subtree T_i that is not completely explored, let u_i be an unexplored vertex in this tree. We can just assume that every u_i has degree $2l$ and $2l-1$ distinct paths of length B connected to it. The optimal offline algorithm Opt can then send $l-t$ agents each to one of the unexplored leaves u_i and then down one of the $2l-1$ distinct paths. These agents in total explore $(l-t)\cdot B$ vertices. All other $l-1+t$ agents are send to the unexplored vertex u_1 in T_1 and then each down one path which is not taken by any other agent. These agents in total explore at least $(l-1+t)\cdot(B-d_1-\Delta)$ vertices. Overall, this yields

$$\textsc{Opt}((I) \geq (l-t)\cdot B + (l-1+t)\cdot(B-d_1-\Delta) = (2l-1)\cdot B + (l-1+t)\cdot(-d_1-\Delta).$$

For the competitive ratio, we hence obtain

$$\frac{\textsc{Opt}(I)}{\textsc{Alg}(I)} \geq \min_{t\in\{0,\dots,l-1\}} \frac{(4l-2)\cdot B + (2l-2+2t)\cdot(-d_1-\Delta)}{l\cdot(B+d_2+12\Delta) + (l-1-t)(B-3d_1)}.$$

In order to maximize the term on the right-hand side, we want to choose d_2 as small as possible. Because of the initial assumptions on the parameters in (3.6), we must satisfy $2d_2 + d_1 \geq B$. We can therefore choose $d_2 = (B-d_1)/2$ and get

$$\frac{\textsc{Opt}(I)}{\textsc{Alg}(I)} \geq \min_{t\in\{0,\dots,l-1\}} \frac{(8l-4)\cdot B + (4l-4+4t)\cdot(-d_1-\Delta)}{l\cdot(3B-d_1+24\Delta) + (2l-2-2t)(B-3d_1)}.$$

Note that since we assumed $d_2 \leq 5d_1/3$, we need to have that $B \leq 13d_1/3$, i.e, $d_1 \geq 3B/13$. We also need to satisfy $3d_1 < B$ by (3.6) or equivalently $d_1 < B/3$.

We now consider an infinite sequence of instances with the following parameters: For every $i \in \mathbb{N}$, let the energy B of the agents be $B^{(i)} := 2^{2i}$, the parameter l be $l^{(i)} := 2^i$ and the depth d_1 be $d_1^{(i)} := b_1 \cdot B^{(i)}$ for some $b_1 \in (3/13, 1/3)$. Note that $d_1^{(i)}$ then satisfies $3d_1^{(i)} < B^{(i)} < 13d_1^{(i)}/3$ as required by our initial assumptions on the parameters. Furthermore, we have

$$\frac{\Delta^{(i)}}{B^{(i)}} = \frac{\left\lceil\sqrt{2l^{(i)}\cdot B^{(i)}}\right\rceil + 2l^{(i)}}{B^{(i)}} \xrightarrow{i\to\infty} 0.$$

By dividing all terms in the numerator and denominator by $l^{(i)} \cdot B^{(i)}$ and using the property above, we can compute

$$\begin{aligned}\frac{\textsc{Opt}(I)}{\textsc{Alg}(I)} &\geq \min_{t \in \{0,\dots,l^{(i)}-1\}} \frac{(8l^{(i)}-4)\cdot B^{(i)} + (4l^{(i)}-4+4t)\cdot(-d_1^{(i)} - \Delta^{(i)})}{l^{(i)} \cdot \left(3B^{(i)} - d_1^{(i)} + 24\Delta^{(i)}\right) + (2l^{(i)}-2-2t)(B^{(i)} - 3d_1^{(i)})} \\ &\xrightarrow{i\to\infty} \inf_{t\in[0,1)} \frac{8-4b_1-4b_1\cdot t}{3-b_1+2-6b_1-2t+6t\cdot b_1}.\end{aligned}$$

We still have the freedom to choose $b_1 \in (3/13, 1/3)$ to maximize the term on the right-hand side, so we even have

$$\frac{\textsc{Opt}(I)}{\textsc{Alg}(I)} \geq \sup_{b_1\in(3/13,1/3)} \inf_{t\in[0,1)} \frac{8-4b_1-4b_1\cdot t}{5-7b_1-2t+6t\cdot b_1}.$$

By standard calculus, we obtain that $b_1 = \frac{-3\sqrt{17}+19}{26} \approx 0.26$ maximizes the infimum and satisfies $3/13 \leq b_1 \leq 1/3$. Finally, we get

$$\frac{\textsc{Opt}(I)}{\textsc{Alg}(I)} \geq \frac{5+3\sqrt{17}}{8} \approx 2.17. \qquad \square$$

Chapter 4

Energy Efficient Delivery

In this chapter, we study the problem of moving a set of distinct messages from their current locations to specific destinations by a team of mobile agents. In an application, a messages could be some person or object to be transported and a mobile agent some autonomous robot or vehicle. The messages can be located at different initial locations and every message has a specific destination. Each mobile agent consumes energy proportional to the distance it travels and the proportionality factor, i.e., the efficiency of the agent, may be different for different agents. The different efficiencies of the agents can be due to different power sources or technologies of the autonomous robot or vehicle, for instance. The agents may carry several messages at the same time, however, there is a capacity κ bounding the number of messages any agent can carry simultanously. We model the environment as a weighted undirected graph, where the initial position and destination of every message is specified as a source-target pair. Previous work on energy-efficient delivery of messages studied agents with different energy budgets, i.e., bounds on the overall energy an agent can spend traversing the environment, but with the same energy efficiency [Cha+13; Cha+14; Bär+16]. In our setting, which we refer to as **WeightedDelivery**, the energy of an agent is unlimited, and we study the problem of delivering all messages to their destinations while minimizing the total energy consumption.

In this chapter, we focus on one aspect of the WeightedDelivery problem, namely, we investigate how much the agents can benefit by collaborating on delivering messages compared to the case that every message is only delivered by one agent. We call the best approximation factor achieved by an algorithm using only one agent for delivering every message the **benefit of collaboration** (BoC). We start by giving a formal introduction of the model in Section 4.1. Afterwards, in Section 4.2, we construct an instance showing that no algorithm that delivers every message by only one agent can achieve an approximation factor better than $\ln\left(\left(1+\frac{1}{2r}\right)^r\left(1+\frac{1}{2r+1}\right)\right)^{-1}$, where r is the minimum of the agent capacity κ and number of messages μ. For a single message this implies a lower bound of $1/\ln 2$ on the benefit of collaboration, whereas for arbitrary large agent capacity and number of messages this lower bound converges to 2. In Section 4.3, we show how to transform an arbitrary solution for the message delivery problem to a solution where every message is only transported by

one agents while the cost is at most twice the cost of the original solution. This implies a general tight upper bound of 2 on the benefit of collaboration for arbitrary capacities and number of messages. Additionally, for the special case of one message, we give a different transformation showing a tight upper bound of $1/\ln 2 \approx 1.44$.

Other aspects of the delivery problem, that we do not cover in this chapter, were presented in [Bär+17]. The authors showed that for only one message an optimal solution can be found in $O(|V|^3)$ independent of the number of agents k. However, for more messages it is shown that already the subproblem of planning in which order an agent delivers a set of messages is NP-hard on planar graphs, but it can be 2-approximated in polynomial time if agents have capacity $\kappa = 1$ and do not collaborate. It is further shown that the coordination aspect of WeightedDelivery, i.e., deciding which agent delivers which subset of messages, is NP-hard, but can be efficiently solved if the agents have the same efficiency. Combining the approximation results and the bound on the benefit of collaboration yield a polynomial-time $(4 \max \frac{\alpha_i}{\alpha_j})$-approximation for message delivery with unit capacities, where $\max \frac{\alpha_i}{\alpha_j}$ is the maximum ratio between the different energy consumption rates of the agents.

Bibliographic Information The results presented in this chapter are joint work with Andreas Bärtschi, Jérémie Chalopin, Shantanu Das, Yann Disser, Daniel Graf, and Paolo Penna, and have been published in [Bär+17].

4.1 Terminology and Model

We model the environment as an undirected labeled and edge weighted graph $G = (V, E)$. Every edge $e = \{u, v\} \in E$ has a **length** denoted by $w(e) \in \mathbb{R}_{\geq 0}$. The length of a walk is the sum of the edge lengths along the walk. The **distance** between a vertex u and a vertex v is the length of a shortest path from u to v in G and denoted by $\mathrm{dist}(u, v)$. There is a set $\mathcal{A}$ of k mobile agents denoted by $A_1, \ldots A_k$ initially located at arbitrary vertices $v_0^{(1)}, \ldots, v_0^{(k)}$ of G. The agents have a complete map of the graph and can communicate globally. Each agent A_i further has a **weight** $\alpha_i > 0$, which is the rate of energy consumption per unit distance traveled by the agent, i.e., every time agent A_i traverses an edge $e \in E$ it incurs an energy cost of $\alpha_i \cdot w(e)$. Note that a higher weight α_i of an agent, implies a higher rate of energy consumption and therefore a lower efficiency so that $1/\alpha_i$ can be interpreted as the efficiency of the agent. Moreover, there is a set of μ messages M to be delivered. For every message $j \in M$ there is a pair (s_j, t_j) giving the **source vertex** $s_j \in V$ and **target vertex** $t_j \in V$ of message j. A message at a vertex v can be picked up by any agent located at v. It can be carried by an agent to any other vertex of G and dropped there. A message $j \in M$ is **delivered** if it is dropped by an agent at its target vertex t_j. Furthermore, the agents have a **capacity** $\kappa \in \mathbb{N} \cup \{\infty\}$, which is a limit on the number of messages an agent can carry simultaneously. We do not impose any restriction on how far an agent may travel and let d_i denote the total distance traveled by agent A_i, i.e., the length of the walk performed by A_i in G. We call a feasable solution S to an instance I of the WeightedDelivery problem a **schedule**. A schedule is a complete description of the agents

trajectories including all message pick-up and message drop-off actions and times. The cost of a schedule S for an instance I is the total energy consumption of the agents, i.e., $c(S,I) := \sum_{i=1}^{k} \alpha_i d_i$. The goal is to find a schedule S minimizing the total energy $c(S,I)$ needed to deliver all messages of instance I.

4.2 Lower Bound on the Benefit of Collaboration

In this section, we construct an instance showing a lower bound on the approximation ratio by an algorithm using only one agent for delivering every message. For our construction, we make use of the fact that the agents have different starting locations and they can finish at any vertex of the graph. Due to different agent efficiencies it may therefore be cheaper that an agent close to the message source first transports a message before handing it over to another agent with a better efficiency compared to the case that the message is transported the whole time by only the agent with the better efficiency. In general, it can even be the case that an agent hands over the message to a less efficient agent if there are multiple messages and capacity constraints for the agents.

Theorem 4.1. *On instances of* WeightedDelivery *with agent capacity κ and μ messages, an algorithm using one agent for delivering every message cannot achieve an approximation ratio better than*

$$\frac{1}{\ln\left(\left(1+\frac{1}{2r}\right)^{r}\left(1+\frac{1}{2r+1}\right)\right)},$$

where $r := \min\{\kappa, \mu\}$.

Proof. The instance I showing the lower bound is constructed as follows: Consider the graph $G = (V,E)$ given in Figure 4.1, where the length $w(e)$ of every edge $e \in E$ is $1/t$. This means that G is a star graph with center $v_{t,0}$ and $r+1$ paths of total length 1 each. We have r messages and message j needs to be transported from $v_{0,j}$ to $v_{2t,0}$ for $j = 1,\dots,r$. There further is an agent $A_{i,j}$ with weight $\alpha_{i,j} = \frac{2r}{2r+i/t}$ starting at every vertex $v_{i,j}$ for $(i,j) \in \{0,\dots,t-1\}\times\{1,\dots,r\}\cup\{t,\dots,2t\}\times\{0\}$.

We first show the following: If any agent transports s messages from their sources to their destinations, then this incurs a cost of at least $2s$. Note that this implies that any schedule S for delivering all messages by the agents such that every message is only carried by one agent satisfies $c(I,S) \geq 2r$.

So let an agent $A_{i,j}$ transport s messages from the sources to the destination $v_{2t,0}$. Without loss of generality let these messages be $1,\dots,s$, which are picked up in this order. By construction, agent $A_{i,j}$ needs to travel a distance of at least i/t to reach message 1, next a distance of 1 to move back to $v_{t,0}$, then a distance of 2 for picking up message j and going back to $v_{t,0}$ for $j = 2,\dots,s$. Finally it needs to move a distance of 1 from $v_{t,0}$ to $v_{2t,0}$. Overall, agent $A_{i,j}$ therefore travels a distance of at least $2s + \frac{i}{t}$. The overall cost for agent $A_{i,j}$ to deliver the s messages therefore is at least

$$\left(2s+\frac{i}{t}\right)\cdot\alpha_{i,j} = \left(2s+\frac{i}{t}\right)\cdot\frac{2r}{2r+i/t} \geq \left(2s+\frac{i}{t}\right)\cdot\frac{2s}{2s+i/t} = 2s.$$

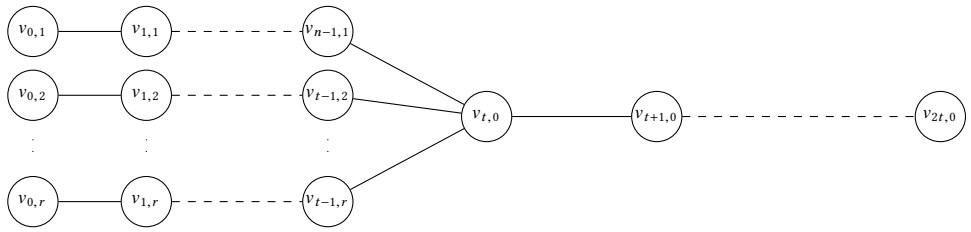


Figure 4.1: Lower bound construction for the benefit of collaboration.

Now, consider a schedule S_{col}, where the agents collaborate, i.e., agent $A_{i,j}$ transports message j from $v_{i,j}$ to $v_{i+1,j}$ for $j = 1, \dots, r$, $i = 0, \dots, t-1$, where we identify $v_{t,j}$ with $v_{t,0}$. Then agent $A_{i,0}$ transports all r messages from $v_{i,0}$ to $v_{i+1,0}$ for $i = t, \dots, 2t-1$. This is possible because $r \leq \kappa$ by the definition of r. The total cost of this schedule is given by

$$c(I, S_{\text{col}}) = r \cdot \int_0^1 f_{\text{step}}(x)\,\mathrm{d}x + \int_1^2 f_{\text{step}}(x)\,\mathrm{d}x,$$

where $f_{\text{step}}(x)$ is a step-function defined on $[0, 2]$ giving the current cost of transporting the message, i.e., $f_{\text{step}}(x) = \frac{2r}{2r+i/t}$ on the interval $[i/t, (i+1)/t)$ for $i = 0, \dots, 2n-1$. The first integral corresponds the first part of the schedule, where the r messages are transported separately and therefore the cost of transporting message j from $v_{i,j}$ to $v_{i+1,j}$ is exactly $\int_{i/t}^{(i+1)/t} f_{\text{step}}(x)\,\mathrm{d}x = \frac{1}{t} \cdot \frac{2r}{2r+i/t}$. The second part of the schedule corresponds to the part, where all r messages are transported together by one agent at a time.

Observe that the function $f(x) = 2r \cdot \frac{1}{2r-1/t+x}$ satisfies $f(x) \geq f_{\text{step}}(x)$ on $[0, 2]$. Hence, we obtain

$$\begin{aligned} c(I, S_{\text{col}}) &\leq r \int_0^1 f(x)\,\mathrm{d}x + \int_1^2 f(x)\,\mathrm{d}x = 2r \cdot \left(r \ln(2r - 1/t + x)\Big|_0^1 + \ln(2r - 1/t + x)\Big|_1^2 \right) \\ &= 2r \cdot \ln\left(\left(\frac{2r - 1/t + 1}{2r - 1/t} \right)^r \left(\frac{2r - 1/t + 2}{2r - 1/t + 1} \right) \right) \overset{t\to\infty}{\to} 2r \cdot \ln\left(\left(1 + \frac{1}{2r} \right)^r \left(1 + \frac{1}{2r+1} \right) \right). \end{aligned}$$

The best approximation ratio of an algorithm that transports every message by only one agent compared to an algorithm that uses an arbitrary number of agents for every message is therefore bounded from below by

$$\text{BoC} \geq \frac{c(I, S)}{c(I, S_{\text{col}})} \geq \frac{2r}{2r \cdot \ln\left(\left(1 + \frac{1}{2r}\right)^r \left(1 + \frac{1}{2r+1}\right) \right)} = \frac{1}{\ln\left(\left(1 + \frac{1}{2r}\right)^r \left(1 + \frac{1}{2r+1}\right) \right)}. \qquad \square$$

By observing that $\lim_{r\to\infty} \ln\left(\left(1 + \frac{1}{2r}\right)^r \left(1 + \frac{1}{2r+1}\right) \right)^{-1} = \ln\left(e^{1/2}\right)^{-1} = 2$, we obtain the following corollary.

Corollary 4.2. *An algorithm for* WeightedDelivery *delivering every message by a single agent cannot achieve an approximation ratio better than 2 in general, and better than* $1/\ln 2 \approx 1.44$ *for a single message.*

4.3 Upper Bounds on the Benefit of Collaboration

In this section, we show a general upper bound on the benefit of collaboration of 2 and an upper bound of $1/\ln 2$ for the case of one message. Our proof of the upper bound of 2 transforms an optimal schedule S_{OPT} for an instance I to a schedule S where every message is transported by only one agent and $c(I,S) \leq 2 \cdot c(I, S_{\text{OPT}})$, see Theorem 4.3. Note that this result does not yield an efficient algorithm. In fact, finding an optimal schedule in which every message is transported by only one agent, is still NP-hard, as shown in [Bär+17]. But the result is an important part in designing the polynomial-time $(4 \max \frac{\alpha_i}{\alpha_j})$-approximation for WeightedDelivery with unit capacities. However, for only one message the simple greedy strategy of choosing the cheapest agent to deliver the message yields an efficient algorithm with an approximation factor of $1/\ln 2$, see Theorem 4.4. In this special case of one message, it is also possible to find an optimal solution in polynomial time, see [Bär+17].

Theorem 4.3. *Let S_{OPT} be an optimal schedule for a given instance I of WeightedDelivery. Then there exists a schedule S such that every message is only transported by one agent and $c(I,S) \leq 2 \cdot c(I, S_{OPT})$.*

Proof. We can assume without loss of generality that in the optimal schedule S_{OPT} for instance I every message $j \in M$ is transported on a path from its starting point s_j to its destination t_j. This can be easily achieved by letting agents drop a message at an intermediate vertex if it would otherwise be transported in a cycle. We now define the directed multigraph $G_{\text{OPT}} = (V, E_{\text{OPT}} \dot\cup \bar{E}_{\text{OPT}})$ as follows:

- V is the set of vertices of the original graph G.
- For every time in the optimal schedule that an agent traverses an edge $\{u,v\}$ from u to v while carrying a set of messages $M' \subseteq M$, we add the arc $e = (u,v)$ to E_{OPT} and $\bar{e} = (v,u)$ to $\bar{E}_{\text{OPT}}$. Note that we can have $M' = \emptyset$ if the agent carries no messsages. We further label both edges with the set of messages M' and write $M_e := M_{\bar{e}} := M'$ to denote these labels. We call the edges in E_{OPT} **original** edges and the edges in $\bar{E}_{\text{OPT}}$ **reverse** edges.

We say that a schedule S in G_{OPT} for an agent A **satisfies the edge labels**, if every original edge $e \in E_{\text{OPT}}$ is traversed at most once by A and only while carrying exactly the set of messages M_e, and every reverse edge $\bar{e} \in \bar{E}_{\text{OPT}}$ is traversed by A at most once and without carrying any message. We further identify a schedule S in G_{OPT} with the schedule S' in G by replacing the traversal of a directed edge $e = (u,v)$ in G_{OPT} by the traversal of the corresponding edge $\{u,v\}$ in G.

The idea of the proof is as follows: The graph G_{OPT} is Eulerian by construction and we show that in each strongly connected component an agent can follow some Eulerian tour that allows to deliver all messages, i.e., a Eulerian tour that satisfies the edge labels. In particular, the agent needs exactly twice as many moves as the total number of moves of all agents in the component in S_{OPT}. By choosing the cheapest agent (in terms of weight) in each component, we obtain a schedule S with $c(I,S) \leq 2 \cdot c(I, S_{\text{OPT}})$.

By only considering a subset of the messages and a subschedule of S_{OPT}, we may from now on assume that G_{OPT} is strongly connected (by construction, every connected component of G_{OPT} is strongly connected). We further let $M(v)$ denote the set of messages currently at a vertex v and use the

notation $S \oplus S'$ to denote the concatenation of a schedule S and a schedule S', i.e., first the schedule S is executed and then S'. The desired schedule is computed using the procedure computeTour() given in Algorithm 4.1, which utilizes the subroutine fetchMessage() given in Algorithm 4.2. We proceed along the following key claims:

1. The schedules returned by computeTour() and fetchMessage() satisfy the edge labels in G_{Opt} and correspond to a closed walk.
2. The following invariants hold after every iteration of any of the two while-loops in computeTour():
   (i) G_{Opt} is Eulerian.
   (ii) For every message $j \in M$ currently at a vertex v_j holds that there is a simple path from v_j to t_j in G_{Opt} with edges in E_{Opt} containing j in their labels, and a path in the reverse direction with edges in $\bar{E}_{\text{Opt}}$ containing j in their labels.
3. For every vertex v_0 in G_{Opt}, a call computeTour(G_{Opt}, v_0) terminates. The returned schedule starts and ends at v_0 and satisfies the edge labels in every step.
4. Combining the schedules of multiple calls of computeTour() yield a schedule S of G_{Opt} for an agent $A_{\min}$ that satisfies the edge labels in every step and corresponds to a Eulerian tour of G_{Opt}. The schedule S satisfies $c(I, S) \leq 2 \cdot c(I, S_{\text{Opt}})$.

Note that the last claim shows our desired result. We now show each of the above claims.

1. It is an easy observation that the schedules computed by computeTour() and fetchMessage() traverse every edge $e \in E_{\text{Opt}}$ while carrying the set of messages M_e and every edge $\bar{e} \in \bar{E}_{\text{Opt}}$ while carrying no messages. Note that in the second else-case in computeTour(), we have $M_e = \emptyset$ so this also holds in this case.

   Next we show that both computeTour() and fetchMessage() return a schedule corresponding to a closed walk. We assume that both procedures terminate, which is shown in the proof of Claim 3. The second while-loop in fetchMessage() only terminates if the current vertex is again the initial vertex v so fetchMessage() clearly returns a closed walk.

   For the procedure computeTour() we show that after every iteration of any of the while-loops the current schedule S corresponds to a closed walk. Initially, S is the empty schedule and clearly corresponds to a closed walk. If in the iteration of the while-loop we add the schedule returned by a call of fetchMessage() to S, then S still corresponds to a closed walk as the added schedule corresponds to a closed walk. Otherwise, first the traversal of an edge e, then the schedule returned by a recursive call of computeTour() and finally the traversal of the reverse edge $\bar{e}$ is added to the current schedule S. By a simple induction over the recursion depth, we can assume that the schedule returned by the recursive call of computeTour() corresponds to a closed walk so that again S corresponds to a closed walk as we travese the reverse edge $\bar{e}$ after traversing e.

Algorithm 4.1: Compute schedule to deliver messages for agent starting at vertex v.

Input: graph G_{OPT}, starting vertex v

Output: schedule S satisfying the edge labels, starting and ending at vertex v

```
1  function computeTour(G_OPT, v)
2      S ← ⊥
3      while ∃ outgoing edge e = (v, w) ∈ E_OPT do
4          if M(v) ⊇ M_e then
5              S ← S ⊕ traversal of e carrying messages M_e
6              delete e, ē from G_OPT and update positions of messages M_e in G_OPT
7              S ← S ⊕ computeTour(G_OPT, w)
8              S ← S ⊕ traversal of ē carrying no messages
9          else
10             let j ∈ M_e \ M(v)
11             S ← S ⊕ fetchMessage(G_OPT, j, v)
12     while ∃ outgoing edge ē = (v, w) ∈ Ē_OPT do
13         if ∃ j ∈ M_ē then
14             S ← S ⊕ fetchMessage(G_OPT, j, v)
15         else
16             S ← S ⊕ traversal of ē carrying no messages
17             delete e, ē from G_OPT
18             S ← S ⊕ computeTour(G_OPT, w)
19             S ← S ⊕ traversal of e carrying no messages
20     return S
```

2. By construction, the graph G_{OPT} is Eulerian at the beginning. As all messages are delivered in the optimal schedule S_{OPT} and they are transported on a path, also the second property holds at the beginning.

   If we assume that a call to fetchMessage() maintains these two properties, then it is easy to see that the two properties are preserved in computeTour(): First of all, an original edge $e \in E_{\text{OPT}}$ is always deleted together with the corresponding reverse edge $\bar{e} \in \bar{E}_{\text{OPT}}$ and thus G_{OPT} is still Eulerian. Moreover, an edge $e = (u, v)$ and a reserve edge $\bar{e}$ are deleted if and only if the set of messages M_e is transported from u to v preserving the second property. Note that if a message is delivered, then the empty path satisfies the second property. What is left to show is that the properties are also preserved by a call fetchMessage(G_{OPT}, j, v) in computeTour(). Again, initially both properties hold by assumption. In the procedure fetchMessage(), we first move on the path of reverse edge with $j \in M_{\bar{e}}$ from the current vertex v to the current location v_j of message j while deleting the reverse edges. Afterwards, we move from v_j on the path of original

Algorithm 4.2: Compute schedule for transporting message j to current vertex v.

Input: graph G_{OPT}, message j, curent vertex v

Output: schedule S transporting message j to vertex v

```
1  function fetchMessage(G_OPT, j, v)
2      S ← ⊥
3      v_cur ← v
4      while j ∉ M(u) do
5          if ∃ outgoing edge ē = (v_cur, w) ∈ Ē_OPT with j ∈ M_ē leaving the current vertex then
6              S ← S ⊕ traverse ē carrying no messages
7              delete ē from G_OPT
8              v_cur ← w
9          else
10             give up
11     while v_cur ≠ v do
12         let e = (v_cur, w) ∈ E_OPT with j ∈ M_e
13         if M(v_cur) ⊇ M_e then
14             S ← S ⊕ traversal of e carrying messages M_e
15             delete e from G_OPT and update positions of messages M_e in G_OPT
16             v_cur ← w
17         else
18             let j' ∈ M_e \ M(v_cur)
19             S ← S ⊕ fetchMessage(G_OPT, j', v_cur)
20     return S
```

edges with $j \in M_e$ back to v while deleting the original edges. Ignoring further recursive calls of fetchMessage(), this means that for every original edge also the reverse edge is deleted. Furthermore, message j is moved to vertex v and thus there again is a path from the current position of message j to t_j in G_{OPT} with edges in E_{OPT} containing j in their labels, and a path in the reverse direction with edges in $\bar{E}_{\text{OPT}}$ containing j in their labels. As this holds for every level of recursive calls of fetchMessage(), G_{OPT} is Eulerian and also the second porperty holds after all recursive calls of fetchMessage() are finished.

3. By Claim 1, computeTour() and fetchMessage() return a schedule corresponding to a closed walk that satisfies the edge labels. What is left to show that a call of computeTour() terminates. First observe that a call to fetchMessage() always leads to some progress as the procedure is only called, if a message j is not at the current vertex so at least one edge is deleted from G_{OPT} in the first while-loop (unless the procedure gives up). Similarly, for every call of computeTour() either an edge e and the corresponding reserve edge $\bar{e}$ are deleled from G_{OPT} or fetchMessage() is called

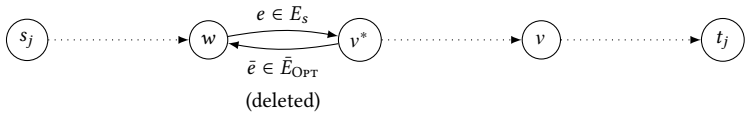


Figure 4.2: Path that message $j \in M$ is transported in graph G_{OPT} according to S_{OPT}.

and also at least one edge is deleted. Thus, there cannot be an infinite sequence of recursive calls as always edges from G_{OPT} are deleted.

We therefore only have to show that the case "give up" in FETCHMESSAGE() cannot occur. Assume, for the sake of contradiction, that this case occurs in a call FETCHMESSAGE(G_{OPT}, j, v). This means that at a vertex v^* in the first while-loop, there is no edge $\bar{e} \in \bar{E}_{\text{OPT}}$ with a label containing message j and v^* also does not contain the message j. By construction of G_{OPT}, the vertex v^* must be on the path that message j takes from its start s_j to its destination t_j in the optimal schedule S_{OPT} and thus initially there must have been an outgoing edge $\bar{e} = (v^*, w) \in \bar{E}_{\text{OPT}}$ at v^* with $j \in M_{\bar{e}}$ that was traversed and deleted. If the corresponding original edge $e = (w, v^*) \in E_{\text{OPT}}$ were already traversed and deleted, then message j would have reached v^* as edge labels are obeyed. This contradicts that in the current call FETCHMESSAGE(G_{OPT}, j, v) the first while-loop has not terminated because we have not encountered message j. Thus, the current setting is as shown in Figure 4.2. The edge $\bar{e}$ cannot have been deleted in a call COMPUTETOUR(G_{OPT}, v^*), as then e would also be deleted. Thus, $\bar{e}$ must have been traversed and deleted during a call FETCHMESSAGE($G^{(1)}_{\text{OPT}}, j_1, v_1$) with $j_1 \neq j$ before as message j is transported on a path. We claim that this call is not completed. Indeed, if the call were already completed, the original edge e would have been traversed and deleted.

As we established that the call FETCHMESSAGE($G^{(1)}_{\text{OPT}}, j_1, v_1$) is not complete, there must be a vertex v_2 and a message j_2 missing at this vertex to further carry j_1 on its paths to the destination, and a call FETCHMESSAGE($G^{(2)}_{\text{OPT}}, j_2, v_2$), which is also incomplete. By iterating this argument, we obtain that the current stack of functions is FETCHMESSAGE($G^{(s)}_{\text{OPT}}, j_s, v_s$), ..., FETCHMESSAGE($G^{(1)}_{\text{OPT}}, j_1, v_1$) for some $s \in \mathbb{N}$, where $j_s = j$ and $v_s = v$. In the optimal schedule S_{OPT} the message j_2 needs to be transported to v_2 before j_1 can be further transported from v_2 together with j_2. Similarly, message j_r needs to be transported to v_r before message j_{r-1} can be transported further together with message j_r from v_r for $r = 2, \ldots, s$. In particular, this implies that message $j_s = j$ needs to be transported to v (via v^*) before j_1 can be transported further. Hence, also in S_{OPT} j message j is transported to v before j_1 is transported further. But this contradicts that $j, j_1 \in M_e$, i.e., in S_{OPT} the messages j and j_1 are transported together along the edge e. Therefore COMPUTETOUR() terminates.

4. Let $A_{\min}$ be an agent with minimum weight among the agents that move in S_{OPT}, let v_0 be the starting vertex of $A_{\min}$ and let T be the schedule resulting from a call COMPUTETOUR(G_{OPT}, v_0). Assume that T does not traverse all edges of G_{OPT}. Let v be the last vertex visited on the tour of $A_{\min}$ according to the schedule T that is is incident to an edge of G_{OPT}, which is not traversed. Further, let v_j be

the position of message j after the schedule T is finished and G'_{OPT} be the graph G_{OPT} after the call of computeTour(G_{OPT}, v_0), i.e., without the edges deleted in the call computeTour(G_{OPT}, v_0) and message j at position v_j instead of s_j. We want to show that we can add the schedule T' returned by a call computeTour(G'_{OPT}, v) to the schedule T as follows: First $A_{\min}$ follows T until the last time it visits v, then it follows T', and finally the remaining part of T.

The graph G'_{OPT} is a feasible input to computeTour() as both properties of Claim 2 are satisfied. By Claim 3, computeTour(G'_{OPT}, v) will produce a schedule T' corresponding to a closed walk that satisfies the edge labels. The only problem that can occur when combining the schedules T and T' therefore is that there is a message j such that $A_{\min}$ visits v_j to transport message j further, but message j has not arrived at v_j as the schedule T is not complete. But this would mean that vertex v_j is visited in the schedule T (in order to transport message j to v_j) after the last time v is visited by the schedule T. However, by the choice of v, all edges incident to v_j must be visited and deleted by the schedule T when $A_{\min}$ starts the schedule T'. This contradicts that v_j is visited in the schedule T'.

By iterative applying the above argument, we obtain a schedule S, which traverses all edges in G_{OPT} while satisfying the edge labels as well as starts and ends at v_0. As $A_{\min}$ is the agent with minimum weight $\alpha_{\min}$, we have

$$2 \cdot c(I, S_{\text{OPT}}) \geq \sum_{e=(v,w) \in E_{\text{OPT}} \dot{\cup} \bar{E}_{\text{OPT}}} w(\{v, w\}) \cdot \alpha_{\min} = c(I, S). \qquad \square$$

For the case of a single message, we can improve the upper bound of 2 on the benefit of collaboration from Theorem 4.3, to a tight bound of $1/\ln 2 \approx 1.44$.

Theorem 4.4. *There is a* $(1/\ln 2)$*-approximation algorithm using a single agent for* WeightedDelivery *with one message.*

Proof. By running an algorithm for the all-pairs shortest path problem, such as the Floyd-Warshall algorithm [CLR89, Chapter 25], we can efficiently determine the agent that can transport the message from s to t with lowest cost in an instance I. We need to show that this is at most $1/\ln(2)$ the cost of an optimum using all agents.

Fix an optimum schedule S_{OPT} for instance I and let the agents $A_1, A_2, \ldots, A_k$ be labeled in the order in which they transport the message in this optimum solution (ignoring unused agents). We first show that we can without loss of generality assume that $\alpha_i > \alpha_{i+1}$ holds for all $i \in \{1, \ldots, k-1\}$. Assume that we have $\alpha_i \leq \alpha_{i+1}$ for some $i \in \{1, \ldots, k-1\}$. Then the part of the message transport carried out by agent A_{i+1} in S_{OPT} can be taken over by agent A_i. Since we have $\alpha_i \leq \alpha_{i+1}$, the cost of the schedule does not increase and thus is still optimal.

By scaling the edge length and agent weights, we can further assume without loss of generality that $\alpha_k = 1$ and that the total distance traveled by the message is 1. Now, for each point $x \in [0, 1]$ along the message path there is an agent A_i with cost α_i carrying the message at this point in the optimum schedule and we can define a function f with $f(x) = \alpha_i$. The function f is a step function

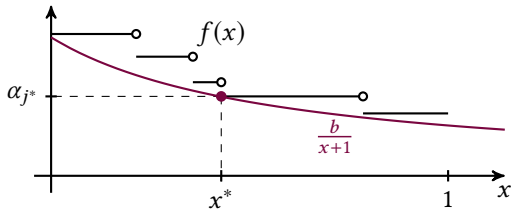


Figure 4.3: Choosing the largest b such that $\frac{b}{x+1}$ is a lower bound on the step-function f representing the weight of the agent currently transporting the message.

that is monotonically decreasing as $\alpha_1 > \alpha_2 > \ldots > \alpha_k$. We further have $f(0) = \alpha_1$ and $f(1) = \alpha_k = 1$. We now choose the largest $b \in [0, 1]$ such that $f(x) \geq \frac{b}{x+1}$, see Figure 4.3.

Note that $b \geq 1$ as $f(x) \geq 1 \geq \frac{b}{x+1}$ for $b = 1$ and all $x \in [0, 1]$. Further, let g_i be the distance traveled by agent A_i without the message and $g := \sum_{i=1}^{k} g_i \alpha_i$ the total cost for the distances traveled by all agents without the message. We obtain the following lower bound for an optimum solution

$$c(I, S_{\text{OPT}}) = \int_0^1 f(x)\,\mathrm{d}x + g \geq \int_0^1 \frac{b}{x+1}\,\mathrm{d}x + g = b\ln(2) + g.$$

By the choice of b, the functions $f(x)$ and $\frac{b}{x+1}$ coincide in at least one point in the interval $[0, 1]$. Let this point be x^* and A_{i^*} be the agent carrying the message at this point. This means that $f(x^*) = \frac{b}{x^*+1} = \alpha_{i^*}$. We will show that it costs at most $c(I, S_{\text{OPT}})/\ln(2)$ for agent A_{i^*} to transport the message alone from s to t. The cost for agent A_{i^*} to reach s is bounded by $g_{i^*}\alpha_{i^*} + x^* \cdot \alpha_{i^*}$ and the cost for transporting the message from s to t is bounded by α_{i^*}. Thus, the cost of a schedule S using only one agent can be bounded by

$$c(I, S) \leq g_{i^*}\alpha_{i^*} + x^* \cdot \alpha_{i^*} + \alpha_{i^*} = g_{i^*}\alpha_{i^*} + (x^* + 1) \cdot \frac{b}{x^* + 1} = b + g_{i^*}\alpha_{i^*}.$$

By using that $g_{i^*}\alpha_{i^*} \leq g$, we finally obtain

$$\frac{c(I, S)}{c(I, S_{\text{OPT}})} \leq \frac{b + g_{i^*}\alpha_{i^*}}{b\ln(2) + g} \leq \frac{b}{b\ln(2)} = \frac{1}{\ln(2)}.$$

□

Bibliography

[AG03] Steve Alpern and Shmuel Gal. *The Theory of Search Games and Rendezvous*. Vol. 55. Springer Science & Business Media, 2003. DOI: 10.1007/b100809 (cit. on pp. 17, 21).

[AH00] Susanne Albers and Monika R. Henzinger. "Exploring Unknown Environments". In: *SIAM J. Comput.* 29.4 (2000), pp. 1164–1188. DOI: 10.1137/S009753979732428X (cit. on p. 15).

[Ale+79] 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: *Proc. 20th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*. 1979, pp. 218–223. DOI: 10.1109/SFCS.1979.34 (cit. on pp. 12, 14, 16).

[Alp+13] Steve Alpern, Robbert Fokkink, Leszek Gasieniec, Roy Lindelauf, and V.S. Subrahmanian. *Search Theory*. Springer, 2013 (cit. on p. 21).

[Amb+11] Christoph Ambühl, Leszek Gąsieniec, Andrzej Pelc, Tomasz Radzik, and Xiaohui Zhang. "Tree Exploration with Logarithmic Memory". In: *ACM Trans. Algorithms* 7.2 (2011), pp. 1–21. DOI: 10.1145/1921659.1921663 (cit. on pp. 13, 16).

[Ana+16] Julian Anaya, Jérémie Chalopin, Jurek Czyzowicz, Arnaud Labourel, Andrzej Pelc, and Yann Vaxès. "Convergecast and Broadcast by Power-Aware Mobile Agents". In: *Algorithmica* 74.1 (2016), pp. 117–155. DOI: 10.1007/s00453-014-9939-8 (cit. on p. 22).

[Awe+99] Baruch Awerbuch, Margrit Betke, Ronald L Rivest, and Mona Singh. "Piecemeal Graph Exploration by a Mobile Robot". In: *Inform. and Comput.* 152.2 (1999), pp. 155–172. DOI: 10.1006/inco.1999.2795 (cit. on pp. 14, 61).

[Bam+17] Evangelos Bampas, Shantanu Das, Dariusz Dereniowski, and Christina Karousatou. "Collaborative Delivery by Energy-Sharing Low-Power Mobile Robots". In: *Proc. 13th Int. Symp. Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS)*. 2017, pp. 1–12. DOI: 10.1007/978-3-319-72751-6_1 (cit. on p. 22).

[Bam+18] Evangelos Bampas, Jérémie Chalopin, Shantanu Das, Jan Hackfeld, and Christina Karousatou. "Maximal Exploration of Trees with Energy-Constrained Agents". In: *ArXiv e-prints* (2018). URL: https://arxiv.org/abs/1802.06636 (cit. on p. 62).

[Bär+16] Andreas Bärtschi, Jérémie Chalopin, Shantanu Das, Yann Disser, Barbara Geissmann, Daniel Graf, Arnaud Labourel, and Matús Mihalák. "Collaborative Delivery with Energy-Constrained Mobile Robots". In: *Proc. 23rd Int. Colloquium Structural Information and Communication Complexity (SIROCCO)*. 2016, pp. 258–274. DOI: 10.1007/978-3-319-48314-6_17 (cit. on pp. 22, 87).

[Bär+17] Andreas Bärtschi, Jérémie Chalopin, Shantanu Das, Yann Disser, Daniel Graf, Jan Hackfeld, and Paolo Penna. "Energy-Efficient Delivery by Heterogeneous Mobile Agents". In: *Proc. 34th Annu. Sympos. Theoretical Aspects Comput. Sci. (STACS)*. 2017, 10:1–10:14. DOI: 10.4230/LIPIcs.STACS.2017.10 (cit. on pp. 88, 91).

[BBV08] Roberto Baldacci, Maria Battarra, and Daniele Vigo. "Routing a Heterogeneous Fleet of Vehicles". In: *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer, 2008, pp. 3–27. DOI: 10.1007/978-0-387-77778-8_1 (cit. on p. 21).

[BE98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998. ISBN: 978-0-521-56392-5 (cit. on pp. 10, 11).

[Ben+02] Michael A. Bender, Antonio Fernández, Dana Ron, Amit Sahai, and Salil Vadhan. "The Power of a Pebble: Exploring and Mapping Directed Graphs". In: *Inform. and Comput.* 176.1 (2002), pp. 1–21. DOI: 10.1006/inco.2001.3081 (cit. on pp. 16, 17, 19, 23).

[Ber98] Piotr Berman. "On-line Searching and Navigation". In: *Online Algorithms, The State of the Art*. 1998, pp. 232–241. DOI: 10.1007/BFb0029571 (cit. on p. 17).

[Bil+13] Davide Bilò, Yann Disser, Luciano Gualà, Matús Mihalák, Guido Proietti, and Peter Widmayer. "Polygon-Constrained Motion Planning Problems". In: *Proc. 9th Int. Symp. Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS)*. 2013, pp. 67–82. DOI: 10.1007/978-3-642-45346-5_6 (cit. on p. 22).

[BK78] Manuel Blum and Dexter Kozen. "On the Power of the Compass (or, Why Mazes Are Easier to Search than Graphs)". In: *Proc. 19th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*. 1978, pp. 132–142. DOI: 10.1109/SFCS.1978.30 (cit. on pp. 1, 12, 16, 17, 20, 23).

[BMS02] Wolfram Burgard, Mark Moors, and Frank E. Schneider. "Collaborative Exploration of Unknown Environments with Teams of Mobile Robots". In: *Proc. Dagstuhl Seminar on Advances in Plan-Based Control of Robotic Agents*. LNCS. 2002, pp. 52–70. DOI: 10.1007/3-540-37724-7_4 (cit. on p. 1).

[Bra+11] Peter Brass, Flavio Cabrera-Mora, Andrea Gasparri, and Jizhong Xiao. "Multirobot Tree and Graph Exploration". In: *IEEE Trans. Robotics* 27.4 (2011), pp. 707–717. DOI: 10.1109/TRO.2011.2121170 (cit. on p. 18).

[BRS95] Margrit Betke, Ronald L. Rivest, and Mona Singh. "Piecemeal Learning of an Unknown Environment". In: *Machine Learning* 18.2-3 (1995), pp. 231–254. DOI: 10.1007/BF00993411 (cit. on pp. 14, 61).

[BRT92] Allan Borodin, Walter L. Ruzzo, and Martin Tompa. "Lower Bounds on the Length of Universal Traversal Sequences". In: *J. Comput. System Sci.* 45.2 (1992), pp. 180–203. DOI: 10.1016/0022-0000(92)90046-L (cit. on p. 14).

[BS77] Manuel Blum and William J. Sakoda. "On the Capability of Finite Automata in 2 and 3 Dimensional Space". In: *Proc. 18th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*. 1977, pp. 147–161. DOI: 10.1109/SFCS.1977.20 (cit. on pp. 17, 23).

[BS94] Michael A. Bender and Donna K. Slonim. "The Power of Team Exploration: Two Robots Can Learn Unlabeled Directed Graphs". In: *Proc. 35th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*. 1994, pp. 75–85. DOI: 10.1109/SFCS.1994.365703 (cit. on pp. 15, 19, 20, 23).

[BT17] Andreas Bärtschi and Thomas Tschager. "Energy-Efficient Fast Delivery by Mobile Agents". In: *Proc. 21th Int. Symp. Fundamentals of Computation Theory (FCT)*. 2017, pp. 82–95. DOI: 10.1007/978-3-662-55751-8_8 (cit. on p. 22).

[Bud75] Lothar Budach. "On the Solution of the Labyrinth Problem for Finite Automata". In: *Elektronische Informationsverarbeitung und Kybernetik* 11.10-12 (1975), pp. 661–672 (cit. on pp. 1, 12, 13).

[Bud78] Lothar Budach. "Automata and Labyrinths". In: *Mathematische Nachrichten* 86.1 (1978), pp. 195–282. DOI: 10.1002/mana.19780860120 (cit. on pp. 1, 12, 13, 23).

[CDK10] Jérémie Chalopin, Shantanu Das, and Adrian Kosowski. "Constructing a Map of an Anonymous Graph: Applications of Universal Sequences". In: *Proc. 14th Int. Conf. Principles Distributed Systems (OPODIS)*. 2010, pp. 119–134. DOI: 10.1007/978-3-642-17653-1_10 (cit. on p. 14).

[Cha+13] Jérémie Chalopin, Shantanu Das, Matús Mihalák, Paolo Penna, and Peter Widmayer. "Data Delivery by Energy-Constrained Mobile Agents". In: *Proc. 9th Int. Symp. Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS)*. 2013, pp. 111–122. DOI: 10.1007/978-3-642-45346-5_9 (cit. on pp. 22, 87).

[Cha+14] Jérémie Chalopin, Riko Jacob, Matús Mihalák, and Peter Widmayer. "Data Delivery by Energy-Constrained Mobile Agents on a Line". In: *Proc. 41st Int. Colloquium Automata, Languages and Programming (ICALP)*. 2014, pp. 423–434. DOI: 10.1007/978-3-662-43951-7_36 (cit. on pp. 22, 87).

Bibliography

[Cha+97] Ashok K. Chandra, Prabhakar Raghavan, Walter L. Ruzzo, Roman Smolensky, and Prasoon Tiwari. "The Electrical Resistance of a Graph Captures its Commute and Cover Times". In: *Comput. Complexity* 6.4 (1997), pp. 312–340. DOI: 10.1007/BF01270385 (cit. on p. 14).

[Chu36] Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363 (cit. on p. 7).

[CL07] Jean-François Cordeau and Gilbert Laporte. "The dial-a-ride problem: models and algorithms". In: *Annals OR* 153.1 (2007), pp. 29–46. DOI: 10.1007/s10479-007-0170-8 (cit. on p. 21).

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989. ISBN: 0-262-03141-8 (cit. on pp. 4, 96).

[Com58] President's Science Advisory Committee. *Introduction to Outer Space*. NASA Historical Reference Colleciton. 1958. URL: https://history.nasa.gov/sputnik/16.html.

[CR80] Stephen A. Cook and Charles Rackoff. "Space Lower Bounds for Maze Threadability on Restricted Machines". In: *SIAM J. Comput.* 9.3 (1980), pp. 636–652. DOI: 10.1137/0209048 (cit. on pp. 1, 25).

[Czy+17] Jerzy Czyzowicz, Krzysztof Diks, Jean Moussi, and Wojciech Rytter. "Energy-Optimal Broadcast in a Tree with Mobile Agents". In: *Proc. 13th Int. Symp. Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS)*. 2017, pp. 98–113. DOI: 10.1007/978-3-319-72751-6_8 (cit. on p. 22).

[Das+06] Shantanu Das, Paola Flocchini, Amiya Nayak, and Nicola Santoro. "Effective Elections for Anonymous Mobile Agents". In: *Proc. 17thth Int. Symp. Algorithms and Computation (ISAAC)*. 2006, pp. 732–743. DOI: 10.1007/11940128_73 (cit. on pp. 19–21).

[Das+07] Shantanu Das, Paola Flocchini, Shay Kutten, Amiya Nayak, and Nicola Santoro. "Map construction of unknown graphs by multiple agents". In: *Theoret. Comput. Sci.* 385.1-3 (2007), pp. 34–48. DOI: 10.1016/j.tcs.2007.05.011 (cit. on pp. 19–21).

[Das13] Shantanu Das. "Mobile agents in distributed computing: Network exploration". In: *Bull. Eur. Assoc. Theor. Comput. Sci. (EATCS)* 109 (2013), pp. 54–69. URL: bulletin.eatcs.org/index.php/beatcs/article/download/27/23 (cit. on p. 19).

[DDK15] Shantanu Das, Dariusz Dereniowski, and Christina Karousatou. "Collaborative Exploration by Energy-Constrained Mobile Robots". In: *Proc. 22nd Int. Colloquium Structural Information and Communication Complexity (SIROCCO)*. 2015, pp. 357–369. DOI: 10.1007/978-3-319-25258-2_25 (cit. on pp. 19, 20, 61).

[Dem+09] Erik D. Demaine, Mohammad Taghi Hajiaghayi, Hamid Mahini, Amin S. Sayedi-Roshkhar, Shayan Oveis Gharan, and Morteza Zadimoghaddam. "Minimizing movement". In: *ACM Trans. Algorithms* 5.3 (2009), 30:1–30:30. DOI: 10.1145/1541885.1541891 (cit. on p. 22).

[Der+15] Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Pająk, and Przemysław Uznański. "Fast collaborative graph exploration". In: *Inform. and Comput.* 243 (2015), pp. 37–49. DOI: 10.1016/j.ic.2014.12.005 (cit. on pp. 18, 20).

[DHK16] Yann Disser, Jan Hackfeld, and Max Klimm. "Undirected Graph Exploration with $\Theta(\log \log n)$ Pebbles". In: *Proc. 27th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*. 2016, pp. 25–39. DOI: 10.1137/1.9781611974331.ch3 (cit. on p. 24).

[DHK18] Yann Disser, Jan Hackfeld, and Max Klimm. "Tight bounds for undirected graph exploration with pebbles and multiple agents". In: *ArXiv e-prints* (2018). URL: https://arxiv.org/abs/1805.03476 (cit. on p. 24).

[Dik+04] Krzysztof Diks, Pierre Fraigniaud, Evangelos Kranakis, and Andrzej Pelc. "Tree exploration with little memory". In: *J. Algorithms* 51.1 (2004), pp. 38–63. DOI: 10.1016/j.jalgor.2003.10.002 (cit. on pp. 13, 16).

[Dis+17] Yann Disser, Frank Mousset, Andreas Noever, Nemanja Skoric, and Angelika Steger. "A General Lower Bound for Collaborative Tree Exploration". In: *Proc. 24th Int. Colloquium Structural Information and Communication Complexity (SIROCCO)*. 2017, pp. 125–139. DOI: 10.1007/978-3-319-72050-0_8 (cit. on pp. 18, 20).

[DKK06] Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. "Optimal Constrained Graph Exploration". In: *ACM Trans. Algorithms* 2.3 (2006), pp. 380–402. DOI: 10.1145/1159892.1159897 (cit. on pp. 14, 16, 61).

[DKS06] Miroslaw Dynia, Miroslaw Korzeniowski, and Christian Schindelhauer. "Power-Aware Collective Tree Exploration". In: *Proc. 19th Int. Conf. Architecture of Computing Systems (ARCS)*. 2006, pp. 341–351. DOI: 10.1007/11682127_24 (cit. on pp. 18, 20, 61).

[DŁS07] Miroslaw Dynia, Jakub Łopuszański, and Christian Schindelhauer. "Why Robots Need Maps". In: *Proc. 14th Int. Colloquium Structural Information and Communication Complexity (SIROCCO)*. 2007, pp. 41–50. DOI: 10.1007/978-3-540-72951-8_5 (cit. on pp. 18–20, 61).

[DP99] Xiaotie Deng and Christos H. Papadimitriou. "Exploring an Unknown Graph". In: *J. Graph Theory* 32.3 (1999), pp. 265–297. DOI: 10.1002/(SICI)1097-0118(199911)32:3<265::AID-JGT6>3.0.CO;2-8 (cit. on p. 15).

[DR59] George B Dantzig and John H Ramser. "The truck dispatching problem". In: *Management Sci.* 6.1 (1959), pp. 80–91. DOI: 10.1287/mnsc.6.1.80 (cit. on p. 21).

Bibliography

[DS17] Yann Disser and Steven S. Skiena. "Geometric Reconstruction Problems". In: *Handbook of Discrete and Computational Geometry, Third Edition*. 3rd ed. CRC Press LLC, 2017. Chap. 35 (cit. on p. 17).

[Dud+91] Gregory Dudek, Michael Jenkin, Evangelos E. Milios, and David Wilkes. "Robotic Exploration as Graph Construction". In: *IEEE Trans. Robot. Autom* 7.6 (1991), pp. 859–865. DOI: 10.1109/70.105395 (cit. on p. 14).

[EHK15] Thomas Erlebach, Michael Hoffmann, and Frank Kammer. "On Temporal Graph Exploration". In: *Proc. 42nd Int. Colloquium Automata, Languages and Programming (ICALP)*. 2015, pp. 444–455. DOI: 10.1007/978-3-662-47672-7_36 (cit. on p. 17).

[EIS76] Shimon Even, Alon Itai, and Adi Shamir. "On the Complexity of Timetable and Multicommodity Flow Problems". In: *SIAM J. Comput.* 5.4 (1976), pp. 691–703. DOI: 10.1137/0205048 (cit. on p. 21).

[EJ73] Jack Edmonds and Ellis L. Johnson. "Matching, Euler tours and the Chinese postman". In: *Math. Program.* 5.1 (1973), pp. 88–124. DOI: 10.1007/BF01580113 (cit. on pp. 15, 22).

[EK72] Jack Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *J. ACM* 19.2 (1972), pp. 248–264. DOI: 10.1145/321694.321699 (cit. on p. 21).

[FHK78] Greg N. Frederickson, Matthew S. Hecht, and Chul E. Kim. "Approximation Algorithms for Some Routing Problems". In: *SIAM J. Comput.* 7.2 (1978), pp. 178–193. DOI: 10.1137/0207017 (cit. on p. 22).

[FI04] Pierre Fraigniaud and David Ilcinkas. "Digraphs Exploration with Little Memory". In: *Proc. 21st Annu. Sympos. Theoretical Aspects Comput. Sci. (STACS)*. 2004, pp. 246–257. DOI: 10.1007/978-3-540-24749-4_22 (cit. on pp. 16, 17, 23).

[FMS09] Paola Flocchini, Bernard Mans, and Nicola Santoro. "Exploration of Periodically Varying Graphs". In: *Proc. 20th Int. Symp. Algorithms and Computation (ISAAC)*. 2009, pp. 534–543. DOI: 10.1007/978-3-642-10631-6_55 (cit. on p. 17).

[Fra+05] Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. "Graph exploration by a finite automaton". In: *Theoret. Comput. Sci.* 345.2-3 (2005), pp. 331–344. DOI: 10.1016/j.tcs.2005.07.014 (cit. on pp. 3, 13, 23).

[Fra+06a] Pierre Fraigniaud, Leszek Gasieniec, Dariusz R. Kowalski, and Andrzej Pelc. "Collective Tree Exploration". In: *Networks* 48.3 (2006), pp. 166–177. DOI: 10.1002/net.20127 (cit. on pp. 18, 20).

[Fra+06b] Pierre Fraigniaud, David Ilcinkas, Sergio Rajsbaum, and Sébastien Tixeuil. "The Reduced Automata Technique for Graph Exploration Space Lower Bounds". In: *Theoret. Comput. Sci., Essays in Memory of Shimon Even* (2006), pp. 1–26. DOI: 10.1007/11685654_1 (cit. on pp. 3, 17, 23, 24, 47, 49).

[FS06] Paola Flocchini and Nicola Santoro. "Distributed Security Algorithms by Mobile Agents". In: *8th Int. Conf. Distributed Computing and Networking (ICDCN)*. 2006, pp. 1–14. DOI: 10.1007/11947950_1 (cit. on p. 19).

[FT05] Rudolf Fleischer and Gerhard Trippen. "Exploring an Unknown Graph Efficiently". In: *Proc. 13th Annu. European Symp. Algorithms*. 2005, pp. 11–22. DOI: 10.1007/11561071_4 (cit. on pp. 15, 16).

[FW16] Klaus-Tycho Foerster and Roger Wattenhofer. "Lower and upper competitive bounds for online directed graph exploration". In: *Theoret. Comput. Sci.* 655 (2016), pp. 15–29. DOI: 10.1016/j.tcs.2015.11.017 (cit. on pp. 15, 16).

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7 (cit. on pp. 7, 8, 11, 21).

[GK10] Subir Kumar Ghosh and Rolf Klein. "Online algorithms for searching and exploration in the plane". In: *Computer Science Review* 4.4 (2010), pp. 189–201. DOI: 10.1016/j.cosrev.2010.05.001 (cit. on p. 17).

[GR08] Leszek Gasieniec and Tomasz Radzik. "Memory Efficient Anonymous Graph Exploration". In: *34th Int. Workshop Graph-Theoretic Concepts in Comput. Sci.* 2008, pp. 14–29. DOI: 10.1007/978-3-540-92248-3_2 (cit. on p. 17).

[Hig+14] Yuya Higashikawa, Naoki Katoh, Stefan Langerman, and Shin-ichi Tanigawa. "Online graph exploration algorithms for cycles and trees by multiple searchers". In: *J. Comb. Optim.* 28.2 (2014), pp. 480–495. DOI: 10.1007/s10878-012-9571-y (cit. on p. 18).

[Hof81] Frank Hoffmann. "One pebble does not suffice to search plane labyrinths". In: *Proc. 3rd Int. Symp. Fundamentals of Computation Theory (FCT)*. 1981, pp. 433–444. DOI: 10.1007/3-540-10854-8_47 (cit. on pp. 2, 12, 16, 23).

[Ist88] Sorin Istrail. "Polynomial Universal Traversing Sequences for Cycles Are Constructible". In: *Proc. 20th Annu. ACM Symp. Theory Computing (STOC)*. 1988, pp. 491–503. DOI: 10.1145/62212.62260 (cit. on p. 13).

[Kah+89] Jeff D. Kahn, Nathan Linial, Noam Nisan, and Michael E. Saks. "On the Cover Time of Random Walks on Graphs". In: *Theor. Probability* 2.1 (1989), pp. 121–128. DOI: 10.1007/BF01048274 (cit. on p. 14).

[Kou02] Michal Koucký. "Universal traversal sequences with backtracking". In: *J. Comput. System Sci.* 65.4 (2002), pp. 717–726. DOI: 10.1016/S0022-0000(02)00023-5 (cit. on pp. 6, 29).

[Kou03] Michal Koucký. "On Traversal Sequences, Exploration Sequences and Completeness of Kolmogorov Random Strings". PhD thesis. Rutgers University, 2003 (cit. on pp. 14, 16, 30, 33).

[KP94] Bala Kalyanasundaram and Kirk Pruhs. "Constructing competitive tours from local information". In: *Theoret. Comput. Sci.* 130.1 (1994), pp. 125–138. DOI: 10.1016/0304-3975(94)90155-4 (cit. on pp. 14, 15).

[KV18] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Sixth Edition. Springer, 2018. DOI: 10.1007/978-3-642-24488-9 (cit. on pp. 4, 7, 11, 21).

[LMS92] Chung-Lun Li, S. Thomas McCormick, and David Simchi-Levi. "The point-to-point delivery and connection problems: complexity and algorithms". In: *Discrete Appl. Math.* 36.3 (1992), pp. 267–292. DOI: 10.1016/0166-218X(92)90258-C (cit. on p. 21).

[LP82] Harry R. Lewis and Christos H. Papadimitriou. "Symmetric Space-Bounded Computation". In: *Theoret. Comput. Sci.* 19 (1982), pp. 161–187. DOI: 10.1016/0304-3975(82)90058-5 (cit. on pp. 10, 12).

[Mar12] Euripides Markou. "Identifying Hostile Nodes in Networks Using Mobile Agents". In: *Bull. Eur. Assoc. Theor. Comput. Sci. (EATCS)* 108 (2012), pp. 93–129. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/52 (cit. on p. 19).

[Mau03] M. Maurette. "Mars Rover Autonomous Navigation". In: *Auton. Robots* 14.2-3 (2003), pp. 199–208. DOI: 10.1023/A:1022283719900 (cit. on p. 1).

[Mir+13] Seyed M. Mirtaheri, Mustafa Emre Dinçktürk, Salman Hooshmand, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel Onut. "A Brief History of Web Crawlers". In: *Center for Advanced Studies on Collaborative Research (CASCON)*. 2013, pp. 40–54. URL: http://dl.acm.org/citation.cfm?id=2555529 (cit. on p. 1).

[MMS12] Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. "Online graph exploration: New results on old and new algorithms". In: *Theoret. Comput. Sci.* 463 (2012), pp. 62–72. DOI: 10.1016/j.tcs.2012.06.034 (cit. on pp. 15, 16).

[Nis92] Noam Nisan. "Pseudorandom generators for space-bounded computation". In: *Combinatorica* 12.4 (1992), pp. 449–461. DOI: 10.1007/BF01305237 (cit. on p. 13).

[OS12] Christian Ortolf and Christian Schindelhauer. "Online Multi-Robot Exploration of Grid Graphs with Rectangular Obstacles". In: *Proc. 24th ACM Symp. Parallelism Algorithmics and Architecture*. 2012, pp. 27–36. DOI: 10.1145/2312005.2312010 (cit. on p. 18).

[OS14] Christian Ortolf and Christian Schindelhauer. "A Recursive Approach to Multi-robot Exploration of Trees". In: *Proc. 21st Int. Colloquium Structural Information and Communication Complexity (SIROCCO)*. 2014, pp. 343–354. DOI: 10.1007/978-3-319-09620-9_26 (cit. on p. 18).

[Pel12] Andrzej Pelc. "Deterministic rendezvous in networks: A comprehensive survey". In: *Networks* 59.3 (2012), pp. 331–347. DOI: 10.1002/net.21453 (cit. on p. 21).

[Pen+12] Zhaomeng Peng, Nengqiang He, Chunxiao Jiang, Zhihua Li, Lei Xu, Yipeng Li, and Yong Ren. "Graph-Based AJAX Crawl: Mining Data from Rich Internet Applications". In: *Proc. Int. Conf. Computer Science and Electronics Engineering*. 2012, pp. 590–594. DOI: 10.1109/ICCSEE.2012.38 (cit. on p. 1).

[Plo+17] Patrick A. Plonski, Joshua Vander Hook, Cheng Peng, Narges Noori, and Volkan Isler. "Environment Exploration in Sensing Automation for Habitat Monitoring". In: *IEEE Trans. Autom. Sci. Eng.* 14.1 (2017), pp. 25–38. DOI: 10.1109/TASE.2016.2613403 (cit. on p. 1).

[PP99] Petrisor Panaite and Andrzej Pelc. "Exploring Unknown Undirected Graphs". In: *J. Algorithms* 33.2 (1999), pp. 281–295. DOI: 10.1006/jagm.1999.1043 (cit. on pp. 13, 16).

[Rao+93] Nageswara Rao, Srikumar Kareti, Weimin Shi, and Sundararaj Iyengar. *Robot Navigation in Unknown Terrains: Introductory Survey of Non-Heuristic Algorithms*. Tech. rep. Oak Ridge National Lab. (United States), 1993 (cit. on p. 17).

[Rei08] Omer Reingold. "Undirected Connectivity in Log-Space". In: *J. ACM* 55.4 (2008), 17:1–17:24. DOI: 10.1145/1391289.1391291 (cit. on pp. 1, 3, 13, 16, 23, 29, 30).

[Rol80] Hans-Anton Rollik. "Automaten in planaren Graphen". In: *Acta Inf.* 13 (1980), pp. 287–298. DOI: 10.1007/BF00288647 (cit. on pp. 3, 13, 16, 17, 23, 24).

[RSI77] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. "An Analysis of Several Heuristics for the Traveling Salesman Problem". In: *SIAM J. Comput.* 6.3 (1977), pp. 563–581. DOI: 10.1137/0206041 (cit. on pp. 15, 16).

[Sav73] Walter J. Savitch. "Maze Recognizing Automata and Nondeterministic Tape Complexity". In: *J. Comput. System Sci.* 7.4 (1973), pp. 389–403. DOI: 10.1016/S0022-0000(73)80031-5 (cit. on p. 1).

[SE84] Cees F. Slot and Peter van Emde Boas. "On Tape Versus Core an Application of Space Efficient Perfect Hash Functions to the Invariance of Space". In: *Proc. 16th Annu. ACM Symp. Theory Computing (STOC)*. 1984, pp. 391–400. DOI: 10.1145/800057.808705 (cit. on p. 8).

[Sha51] Claude A. Shannon. "Presentation of a Maze-Solving Machine". In: *Trans. 8th Conf. on Cybernetics*. 1951, pp. 173–180 (cit. on p. 12).

[Sha74] Anupam N. Shah. "Pebble Automata on Arrays". In: *Computer Graphics and Image Processing* 3.3 (1974), pp. 236–246. DOI: 10.1016/0146-664X(74)90017-3 (cit. on pp. 1, 12, 23).

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. "Amortized Efficiency of List Update and Paging Rules". In: *Commun. ACM* 28.2 (1985), pp. 202–208. DOI: 10.1145/2786.2793 (cit. on p. 11).

Bibliography

[TV02] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. SIAM, 2002. ISBN: 978-0-898-71851-5 (cit. on p. 21).

[WS11] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011. ISBN: 978-0-521-19527-0 (cit. on p. 11).