POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

The author of the PhD dissertation: Dorota Osula
Scientific discipline: Informatics

DOCTORAL DISSERTATION

Title of PhD dissertation: Multi-agent graph searching and exploration algorithms

Title of PhD dissertation (in Polish): Algorytmy przeszukiwania i eksploracji grafów przez grupę mobilnych agentów

Supervisor *signature*	Second supervisor *signature*
dr hab. inż. Dariusz Dereniowski, prof. nadzw. PG	
Auxiliary supervisor *signature*	Cosupervisor *signature*

Gdańsk, year 2019

GDAŃSK UNIVERSITY OF TECHNOLOGY

DOCTORAL THESIS

Multi-agent graph searching and exploration algorithms

Author:
mgr inż. Dorota Osula

Supervisor:
dr hab. inż. Dariusz Dereniowski,
prof. nadzw. PG

FACULTY OF ELECTRONICS,
TELECOMMUNICATIONS AND INFORMATICS

Department of Algorithms and Systems Modelling

2020

Acknowledgements

First and foremost, I would like to express my profound gratitude to my supervisor, Professor Dariusz Dereniowski for the continuous support during my PhD studies and research. I am sincerely grateful for his patience, motivation, immense knowledge and invaluable guidance throughout my research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my PhD.

Secondly, I wish to express my warmest gratitude to my co-authors for sharing your experiences and knowledge, and for providing invaluable support and feedback throughout the research. I would also like to thank Professor Rita Zuazua for her friendship, guidance and giving me the amazing opportunity to visit Mexico.

Finally, my deep and sincere gratitude to my family for their continuous and unparalleled love, help and support. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am today. I am grateful for my husband for always being there for me. They selflessly encouraged me to explore new horizons and to seek my own destiny. This journey would not have been possible if not for them. I dedicate this milestone achievement to them.

Abstrakt

Grupa mobilnych *agentów* musi wykonać powierzone im zadanie na grafie, zaczynając od specjalnie wyznaczonych wierzchołków zwanych *bazami*. Celem jest skonstruowanie *strategii* (tj., sekwencji kroków, z których każdy jest zbiorem ruchów agentów), która pozwoli agentom na wykonanie zleconego zadania. Strategie analizujemy pod kątem ich efektywności, np. liczby wykorzystanych agentów, sumy wszystkich wykonanych ruchów czy liczby kroków strategii. Obecnie dziedzina *przeszukiwania* oraz *eksploracji a priori* nieznanego agentom grafu (tj. *on-line*) dynamicznie się rozwija. Ciągle proponowane są nowe podejścia, w celu ulepszenia sposobu modelowania rzeczywistych problemów, takich jak przeszukiwanie niebezpiecznych obszarów czy konstrukcja mapy nieznanego terenu przez grupę robotów.

Problemy przeszukiwania i eksploracji grafów w sposób scentralizowany i gdy graf jest dany agentom na wejściu (tj. *off-line*) są znanymi problemami w teorii grafów i wiele wyczerpujących wyników zostało już uzyskanych. W poniższej pracy skupiamy się na monotonicznym spójnym przeszukiwaniu grafu, rozproszonej eksploracji on-line grafów oraz częściowej eksploracji digrafów. Zwracamy w tym miejscu uwagę, że zadania przeszukiwania i eksploracjii są ze sobą mocno powiązane. Pierwsze może być postrzegane jako konstrukcja strategii, która ma za zadanie złapać ruchomego uciekiniera, zaś w drugim uciekinier jest statyczny. Dlatego, różnica między nimi tak naprawdę leży w zachowaniu się jednostki będącej celem pościgu. W tej pracy używamy obydwóch pojęć, aby być zgodnymi z powrzechną literaturą.

Rozprawę otwiera rozdział przeglądowy, który w wyczerpujący sposób traktuje o przeszukiwaniu oraz eksploracji grafów. Po nim, w kolejnych czterech rozdziałach zaprezentowane są poniższe wyniki:

– Dowolny, nieznany graf z klasy częściowych krat (ang. *partial grids*) o liczbie wierzchołków n może być przeszukany w sposób spójny przez grupę mobilnych agentów o rozmiarze $O(\sqrt{n})$, gdy $\Omega(\sqrt{n}/\log n)$ jest niezbędne.

- Sprawdzenie czy spójna szerokość ścieżkowa grafu (ang. *connected pathwidth*) jest mniejsza od ustalonej liczby całkowitej k może być wykonane w czasie wielomianowym.

- Dla dowolnego cyklu oraz drzewa w czasie wielomianowym może zostać znaleziony algorytm, który pozwoli grupie mobilnych agentów na eksplorację znanego grafu w sposób optymalny, gdzie parametrem minimalizacyjnym jest suma przebytych dróg oraz cena agentów. Dla *a priori* nieznanych cykli konstruujemy 2-kompetytywny algorytm. Pokazaliśmy również, że każdy on-line algorytm jest co najmniej $3/2$ razy gorszy (2 razy gorszy) niż optymalny algorytm off-line dla cykli (dla drzew).

- Problem stwierdzenia możliwości uspójnienia podgrafu spinającego wybrane wierzchołki w digrafie poprzez wyczyszczenie przez zadaną liczbę mobilnych agentów jest NP-zupełny oraz FPT (Fixed Parameter Tractable).

Abstract

A team of mobile *entities*, which we refer to as *agents* or *searchers* interchangeably[1], starting from *homebases* needs to complete a given task in a graph. The goal is to build a *strategy*, which allows agents to accomplish their task. We analyze strategies for their effectiveness (e.g., the number of used agents, the total number of performed moves by the agents or the completion time). Currently, the fields of *on-line* (i.e., agents have no *a priori* knowledge about the graph topology) multi-agent graph *searching* and *exploration* are rapidly expanding. Recent studies have presented new approaches and models to better describe real-life problems like clearing danger areas by a group of robots or constructing a map of an unknown terrain.

A centralized searching and exploration in the *off-line* setting (i.e., when the topology of a graph is known in advance) are well studied, due to their wide applications in robotic and network fields, and many profound results have been established. In this thesis we are focusing on the issues of the *monotone connected decontamination* problem, the on-line collaborative exploration and the partial exploration of digraphs. We point out that the two tasks, namely graph searching and exploration are closely related. The former can be seen as designing a strategy that aims at finding a moving entity while the latter can be seen as finding a static entity. Thus, the difference lies in the behavior of the target. We use the names graph searching and exploration also to be consistent with existing literature.

Firstly, we provide two comprehensive surveys on the topics of graph searching and exploration. Then in the four subsequent chapters, we present the following results:

– We give a distributed algorithm for the searchers that allows them to compute a connected and monotone strategy that guarantees searching any unknown *partial grid* of order n with the use of $O(\sqrt{n})$ searchers. Moreover, we give a lower bound of $\Omega(\sqrt{n}/\log n)$ in terms of achievable competitive ratio of any distributed algorithm.

[1]Historically, the terms 'agent' and 'searcher' have been used in the contexts of graph exploration and searching, respectively. Thus, in this thesis we use them both, dependently on the described topic.

– Checking if the *connected pathwidth* of any graph is at most some fixed integer k can be done in polynomial time.

– Let the *cost* of a strategy be the total distance traversed by agents coupled with the price of invoking them. We construct two cost-optimal off-line algorithms for rings and trees, respectively. For unknown rings, we give a 2-competitive algorithm. We prove a lower bound of competitive ratio of $3/2$ (for rings) and 2 (for trees) for any on-line algorithm.

– The problem of establishing if there exists a subgraph, which connects a chosen vertices and can be explored by a given number of agents is NP-hard and FPT.

Summary

This thesis is organized as follows. Firstly, in Chapter 1, we give the necessary definitions and notations that are used in the rest of the work. In particular we define the concepts of a graph and digraph (weighted and non-weighted) together with some basic properties (Section 1.1). In Section 1.2 we define a strategy as a sequence of sets of agents' moves and give the formal definition of the basic model. Then in Section 1.3, we describe models of our interests and state the main results obtained in this thesis.

Chapter 2 provides comprehensive surveys on the two deterministic agents' problems: graph decontamination (Section 2.1) and graph exploration (Section 2.2). Different models are described based on: the optimization factor of a strategy, communication between agents, time clock and the initial knowledge about the graph (i.e., the on-line and off-line settings). In Section 2.1 the numerous results are presented for various search numbers, also in the context of other graph's parameters, e.g., pathwidth or vertex separation. Then, we look closer at monotone, connected and internal strategies, on which we would focus on later. We finish this section with tables (Table 2.1-Table 2.7) presenting all (to the best of this author's knowledge) results in the field, including several model modifications (e.g., equipping agents in special proprieties of *visibility* or *cloning*). Section 2.2 surveys the distributed graph exploration by a team of mobile agents. We divided it into edge- and vertex-exploration and present current results for different optimization factors (e.g., completion time or the number of agents) on several graph classes. Up to date results are shown in Tables 2.8-2.10.

The thesis is divided into two parts: Part I Decontamination and Part II Exploration, where both consist of two chapters with newly obtained results in the fields.

As a way of modeling two-dimensional shapes, we restrict our attention in Chapter 3 to networks that can be embedded into partial grids: nodes are placed on the plane at integer coordinates and only nodes at distance one can be adjacent. Distributed, connected and monotone decontamination in the on-line setting is investigated, where the number of

searchers is being minimized. Although searchers do not have any knowledge about the graph *a priori*, we equip them in the sense of direction, i.e., they recognize the direction of an incident edge (up, down, left or right). We give a distributed algorithm for the searchers that allows them to compute a connected and monotone strategy that guarantees searching any unknown partial grid with the use of $O(\sqrt{n})$ searchers, where n is the number of nodes in the grid. As for a lower bound, there exist partial grids that require $\Omega(\sqrt{n})$ searchers. Moreover, we prove that for each searching algorithm in the on-line setting there is a partial grid that forces the algorithm to use $\Omega(\sqrt{n})$ searchers but $O(\log n)$ searchers are sufficient in the off-line scenario. This gives a lower bound of $\Omega(\sqrt{n}/\log n)$ in terms of achievable competitive ratio of any distributed algorithm.

The graph searching number is strictly linked to the pathwidth parameter of the graph. During the GRASTA 2017 workshop, Fedor V. Fomin [75] raised an open question, whether we can verify in polynomial time, if the connected pathwidth of a given graph is at most k, for a fixed k. In Chapter 4 we answer this question in the affirmative by providing an algorithm inspired by the algorithm for computing minimum-length path decompositions by Dereniowski, Kubiak, and Zwols [50].

Chapter 5 presents results on the exploration of rings and trees in the off-line and on-line settings. We are interested in the cost-optimal strategies, where the cost of a strategy is understood as the total distance traversed by agents coupled with the price of invoking them. The algorithms that compute optimal strategies for a given ring or tree of order n are constructed. For unknown rings, we give a 2-competitive algorithm and prove a lower bound of competitive ratio of $3/2$ for any on-line algorithm. For every on-line algorithm for trees, we prove the competitive ratio to be no less than 2, which can be achieved by the DFS algorithm.

In the second chapter of the exploration part (Chapter 6) we study several problems of clearing subgraphs by mobile agents in digraphs. The agents can move only along directed walks of a digraph and, depending on the variant, their initial positions may be pre-specified. In general, for a given subset $\mathcal{S}$ of vertices of a digraph D and a positive integer k, the objective is to determine whether there is a subgraph $H = (V, A)$ of D such that (a) $\mathcal{S} \subseteq V$, (b) H is the union of k directed walks in D, and (c) the underlying graph of H includes a Steiner tree for $\mathcal{S}$ in D. Since a directed walk is not necessarily a simple directed path, the problem is actually on covering with paths. We provide several results on the polynomial time tractability, hardness, and parameterized complexity of the problem. Our main fixed-parameter algorithm is randomized.

Finally, we close this thesis with conclusion in Chapter 7.

Most of the results in this thesis have been published. Section 2.1 is based on the survey accepted for publication in *Utilitas Mathematica* [128]. Extended abstract of Chapter 3 has been presented at the *15th Workshop on Approximation and Online Algorithms (WAOA 2017)* and published in the *LNCS* series [51]. Authors have been invited to publish the full version of the paper in journal of *Theory of Computing Systems*. Chapter 4 answers the open question raised in [75] and has been accepted for publication in the special issue of *Theoretical Computer Science*. Extended abstract of Chapter 5 has been presented at the *45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2019)* and published in the *LNCS* series [129]. Finally, Chapter 6 is based on the article from *Journal of Computer and System Sciences* [52].

Contents

List of Symbols

$A(G,H)$	a number of searchers used by algorithm A for graph G starting from the set of homebases H
$A^{opt}(G,H)$	a minimum number of searchers to search (or explore) graph G starting from the set of homebases H
$\alpha(H,\mathcal{P})$	start point of interval of connected graph H in decomposition $\mathcal{P}$
$b(C)$	bottleneck of checkpoint C
B	agent-quantity function
$\mathcal{B}$	set of all bases of some digraph
$\mathcal{B}(S)$	set of all S-branches
$\beta(H,\mathcal{P})$	end point of interval of connected graph H in decomposition $\mathcal{P}$
$c(v)$	children of vertex v
$c(\mathcal{S})$	cost of strategy $\mathcal{S}$
$\mathtt{cpw}(G)$	connected pathwidth of graph G
$\mathtt{cs}(G)$	connected edge search number of graph G
$C\langle i\rangle$	i-th expansion of checkpoint C
$d_H(v,u)$	distance between vertices v and u in graph or digraph H
$\deg_H(v)$	degree of vertex v in graph or digraph H
$\deg_{\text{in}}(v)$	in-degree of vertex v
$\deg_{\text{out}}(v)$	out-degree of vertex v
$D=(V,A)$	digraph D with vertex-set V and arc-set A
$\mathtt{ens}(G)$	mixed search number of graph G
$F(p_1,p_2)$	a frontier of ends in points p_1 and p_2
$G=(V,E)$	graph G with vertex-set V and edge-set E
$G[Y]$	subgraph of graph G induced by vertex set Y
$\mathtt{is}(G)$	internal edge search number of graph G
$I(S,\mathcal{P})$	interval of S in path decomposition $\mathcal{P}$
$\mathcal{L}(T)$	leaves of tree T
m	number of edges (arcs) of a graph (digraph)
$\mathtt{ms}(G)$	monotone edge search number of graph G
n	number of vertices of a graph or digraph

$\mathtt{ns}(G)$	node search number of graph G
$\mathbb{N}$	natural numbers and zero
$\mathbb{N}^+$	natural numbers
$N_G(Y)$	neighborhood of vertex set Y in G
o	asymptotic little O notation
O	asymptotic big O notation
$p(v)$	parent of vertex v
$\mathtt{ppw}(G)$	proper-pathwidth of graph G
$\mathtt{pw}(G)$	pathwidth of graph G
$\mathcal{P}$	a path decomposition
$P_T(v,u)$	path between two vertices v and u in tree T
$\pi_1 \circ \pi_2$	concatenation of two walks π_1 and π_2
$\mathbb{R}$	rational positive numbers and zero
$\mathbb{R}^+$	rational positive numbers
$\mathcal{R}(F,i)$	i-th rectangle of frontier F
$s(D)$	set of all source vertices in digraph D
$\mathtt{s}(G)$	edge search number of graph G
T_v	tree rooted in vertex v
$\mathrm{TC}(D)$	transitive closure of digraph D
θ	asymptotic little Theta notation
Θ	asymptotic big Theta notation
$\mathtt{vs}(G)$	vertex separation of graph G
$w(e)$	weight of edge e
$w(G)$	sum of weight of all edges of graph G
$W(v,u)$	walk from vertex v to vertex u
q	invoking cost
ω	asymptotic little Omega notation
Ω	asymptotic big Omega notation

Chapter 1

Introduction

This thesis starts with giving the necessary definitions and notations, that are used in the rest of the work. In particular, Section 1.1 gives the basic graph-theoretic definitions and Section 1.2 describes basic multi-agent searching and exploration models. Then, in Section 1.3 we describe models formally and state the main results obtained in this thesis.

1.1 Graph Theory Definitions

An *undirected graph* (which we refer shortly as to *graph*) $G = (V(G), E(G))$ is an ordered pair of two sets, where $V(G)$ denotes the set of *vertices* (or *nodes*) of G and $E(G)$ the set of *edges* of G. We write $V = V(G)$ and $E = E(G)$, when a graph is clear from the context. Each edge is a pair of two vertices, i.e., $e = \{u, v\}$, where $e \in E$ and $u, v \in V$. A *vertex-weighted* graph is a triple $G = (V, E, w_V)$, where $w_V : V \to \mathbb{R}$ is a weight function defined for all vertices. An *edge-weighted* graph is a triple $G = (V, E, w_E)$, where $w_E : E \to \mathbb{R}$ is a weight function defined for all edges. The *degree* of a vertex of a graph is the number of edges incident to the vertex.

A digraph $D = (V(D), A(D))$ is an ordered pair of two sets, where $V(D)$ denotes the set of vertices of D and $A(D)$ the set of *arcs* of D. Similarly, we write $V = V(D)$ and $A = A(D)$, when a digraph is clear from the context. Arcs are directed edges, i.e., $(u, v) \neq (v, u)$ for any $u, v \in V$. We can look at digraphs as a generalization of undirected graphs. Indeed, every graph G can be represented by a digraph by replacing each edge $\{u, v\} \in E(G)$ by two arcs: (u, v) and (v, u). Thus, problems solved on digraphs (as the one in Chapter 6), are (in the considered cases) more general than on undirected graphs. We define *vertex-weighted* and *arc-weighted* digraphs analogously as for undirected graphs. The *underlying graph* G of D is a graph with the same vertex set and $\{u, v\} \in E(G)$ if and only if there is an arc between u and v in D. The *in-degree* $\deg_{\text{in}}(v)$ of $v \in V$ is the number of arcs $(u, v) \in A$ and the *out-degree* $\deg_{\text{out}}(v)$ of v is the number of arcs

$(v,u) \in A$. A *source* of a directed graph is a vertex of in-degree zero. The set of all source vertices in a directed graph D is denoted by $s(D)$. By the *transitive closure* $\mathrm{TC}(D)$ of a directed graph D, we mean a directed graph on the same vertices such that for each pair (u,v) of distinct vertices, there is an arc from u to v if and only if there is a directed path from u to v in the original digraph.

For any graph G or digraph D we denote as $W(v,u)$ a walk that starts in $v \in V$ and finishes in $u \in V$ (if $v = u$ then either walk is a single vertex or a closed walk). A path is understood as an open walk with no repeated vertices. If a graph (digraph) is non-weighted, then the *distance* between two vertices in it is the number of edges (arcs) in a shortest (directed) path connecting them. For edge-weighted graph (digraph) the distance is the sum of weights of all edges (arcs) in a shortest (directed) path. For any graph or digraph H and $v,u \in V(H)$ we denote the distance between v and u by $d_H(v,u)$ and omit the bottom index, when H is clear from the context. The number of vertices of any walk π is denoted shortly as $|\pi| = |V(\pi)|$. For two walks π_1 and π_2, where π_2 starts at the ending point of π_1, the concatenation of π_1 and π_2 is denoted by $\pi_1 \circ \pi_2$.

For any graph G and a set $Y \subseteq V(G)$, the subgraph with vertex set Y and edge set $\{\{u,v\} \in E(G) \mid u,v \in Y\}$ is denoted by $G[Y]$ and is called the subgraph *induced* by Y. For $Y \subseteq V(G)$, we write $N_G(Y)$ to denote the *neighborhood* of Y in G, defined as $N_G(Y) = \{v \in V(G) \setminus Y \mid \exists u \in Y \text{ s.t. } \{u,v\} \in E(G)\}$.

A graph is *connected* if for every pair of vertices u and v, there is a walk from u to v. A digraph is *weakly connected* if its underlying graph is connected. A *simple* graph (digraph) has neither self-loops nor multi-edges (multi-arcs). In this work we are interested only in connected (weakly connected) and simple graphs (digraphs).

1.2 Search and Exploration Strategies

In this subsection we formally define the basic *search* and *exploration* models. In general, there exist many different models (see surveys in Chapter 2), but the definitions below are common for them all. Additional requirements and restrictions that have to be fulfilled in order for *strategies* to become solutions to the problems solved in this thesis are described firstly in the next subsection, and then in more details in subsequent chapters.

Let $\mathcal{G}$ be a class of connected (weakly connected) simple graphs (digraphs), $G = (V,E) \in \mathcal{G}$ and let $n = |V|$. In this work (unless stated

differently) it is assumed that agents have identifiers and unbounded computational power. Each agent, while traversing a graph, executes its own algorithm and this algorithm uses some local memory. We say for short that this is the *memory of the agent* and we assume that it is polynomial in the size of the graph.

For the search and exploration problems a strategy must follow a certain pattern, that is, all searchers have to be placed at specific nodes, called *homebases*.

Definition 1.2.1. *When an agent first occurs in a graph G it can only be placed on a node called* homebase*. A set of homebases is predefined for a given graph (i.e., it is not chosen by the agents). In particular, it can be only one node or the whole set V.*

For a graph G, set of homebases and a group of available agents our goal is to find a strategy, which is a sequence of *steps*, which are sets of agents' *moves*.

Definition 1.2.2. *A* move *consists of selecting one agent and performing one of the following actions: (1) placing a chosen agent on one of the homebases, (2) sliding a chosen agent along an edge from the node this agent occupies to a neighbouring one or (3) removing a chosen agent from the graph.*

Definition 1.2.3. Step *is a set of moves. In other words, step includes moves of agents that are performed simultaneously. We require that a step includes at most one move of each agent.*

Definition 1.2.4. *For a graph G, set of homebases $H \subseteq V$ and a set of k agents a* strategy *$\mathcal{S}$ is a sequence of $l \in \mathbb{N}^+$ steps, i.e., $\mathcal{S} = (S_1, \ldots, S_l)$, where S_i is the i-th step. Moves from every step are performed simultaneously and moves from S_i, $i \in \{2, \ldots, l\}$ are made after all moves from S_{i-1} have finished.*

In order to find a strategy, we construct an *algorithm*, which based on the provided input computes the next step of a strategy. We consider in this thesis two types of algorithms.

Definition 1.2.5. *By a* centralized algorithm *in this work we understand a deterministic procedure that runs on a single processor, which for a given input computes a strategy that is then executed by the agents.*

Definition 1.2.6. *A* distributed algorithm *is an algorithm designed to run the private processors of mobile agents. Thus, such an algorithm runs concurrently and independently on multiple processors with only a limited amount of information.* [1]

[1] Definition taken from the book *Distributed Algorithms* by Nancy A. Lynch [115].

Both types of algorithms can run in the two different settings: *on-line* and *off-line* one.

Definition 1.2.7. *In the* off-line *setting an algorithm receives the entire graph G as an input.*

Definition 1.2.8. *In the* on-line *setting agents initially have no knowledge about the graph.*

Although an algorithm computes a strategy, we often view it as a procedure that computes the next step of the strategy (especially in the on-line setting, where the input is provided to the algorithm gradually after each step of the strategy is performed). In other words, a centralized algorithm, based on its current knowledge about the graph, computes the next step of a strategy, i.e., it dictates agents their next moves. In particular an algorithm may know the whole graph from the beginning (off-line setting) or learn its structure with time (on-line setting). On the other hand, in the distributed model, each agent calculates its own next move based on the content of its local memory (i.e., part of a graph, which it has explored so far or has learned from the other agents). Distributed algorithm can also run in two settings: off-line (where although the structure of a graph is known to all agents from the beginning, they might be unaware of, e.g., other agents' positions) and on-line one.

For brevity, whenever we refer to an algorithm in the on-line (off-line) setting we call it *on-line* (respectively *off-line*) algorithm. Moreover, when we write that an algorithm A *uses* $f(n)$ agents, what we mean is that a strategy computed by A for any n-node graph uses at most $f(n)$ agents.

In a distributed algorithm, a channel of communication has to be provided for the agents, i.e., the agents have to communicate with each other. The content of the exchanged messages is used by agents' algorithms to determine their further moves. We distinguish two models.

Definition 1.2.9. Global communication *gives agents the ability to exchange an unbounded number of messages in spite of their locations. Moreover, it is assumed that communication does not take any time, i.e., sending or receiving a message is not considered as a separate move.*

Definition 1.2.10. *In the* local communication *model agents can only exchange messages when present and the same node. Similarly, as in the global communication model, sending and receiving a message does not take any time.*

In this thesis, we are interested in finding strategies that solve two graph-theoretic problems: *decontamination* and *exploration*. The word

searching is used interchangeably with decontamination throughout this work.[2] [3]

Problem 1.2.1. *In the* decontamination problem *initially, all edges are* contaminated*. After each move of sliding a searcher along an edge, it is declared to be* clear*. It becomes contaminated again (*recontaminated*) if at any time during execution of the strategy $\mathcal{S}$ at least one of its endpoints is not occupied by a searcher and is incident to a contaminated edge. We consider only strategies in which recontamination does not occur and we call such strategies* monotone*. We say that a strategy $\mathcal{S}$* decontaminates *(or* clears*) graph G if after performing the last step of $\mathcal{S}$ all edges are clear.*

Problem 1.2.2. *In the* exploration problem *each vertex has to be visited at least once. In other words, we say that a strategy $\mathcal{S}$* explores *a graph G if after performing the last step of $\mathcal{S}$ each vertex has been visited at least once by some agent.*

Finally, we would like to measure the *efficiency* of a strategy in order to compare strategies. A common approach is to look for a strategy that minimizes the number of used agents. In this work, apart from minimizing the number of agents we are interested also in other factors, which (for more clarity) are described in details in specific chapters. Following the common approach, we also want to compare on-line algorithms with the results of the optimal strategies computed in the off-line setting, by calculating theirs *competitiveness*.

Definition 1.2.11. *Let A be any on-line algorithm that computes a decontaminating (or exploration) strategies for all graphs from some class $\mathcal{G}' \subseteq \mathcal{G}$. By $A(G,H)$ we denote the number of agents used by a strategy computed by A for a graph $G \in \mathcal{G}'$ starting from the set of homebases H. Let now $A^{opt}(G,H)$ be a minimum number such that there exists a strategy that decontaminates (or explores) graph G and starts from H. We say that an algorithm A is $f(n)$-*competitive*, for some function f, if*

$$\max_H \frac{A(G,H)}{A^{opt}(G,H)} \leq f(n)$$

[2] Notice, that in computer science the concept of graph search is often equivalent to *graph traversal*, i.e., the process of visiting all nodes in a graph in the centralized setting. To the group of the graph traversal algorithms we include strategies such as depth-first search, breadth-first search, best-first search and many others.

[3] The reason for using in this thesis searching as a broader concept than in computer science, is that the word decontamination is used mostly in the context of distributed or on-line computations, while searching is used for off-line problems related to pathwidth, treewidth and other width-like measures of the graph. This is done to stay consistent with the terminology used in the literature.

for each n-node graph $G \in \mathcal{G}'$. If H consists of only one homebase h, then for brevity we write $A(G,h)$ and $A^{opt}(G,h)$.

Similarly, a measure of the efficiency of an on-line algorithm the *competitive ratio* is understood as the worst-case ratio between the number of agents produced by the on-line algorithm and the optimal off-line algorithm.

1.3 Problems' Statements and Main Results

In this section we formally describe each of our four considered subproblems and present the main results. We would like to notice here that all of these subproblems are described in more details together with all necessary notation in particular chapters.

1.3.1 Decontamination of Partial Grids

Let G be a simple, undirected, connected partial-grid (i.e., a subgraph of a grid) with a single homebase h. A *monotone connected k-search strategy* $\mathcal{S}$ for a network G is defined as follows. Initially, k searchers are placed on h of G. Then, $\mathcal{S}$ is a sequence of moves, where each move consists of selecting one searcher present at some node u and sliding the searcher along an edge $\{u,v\}$. (Thus, the searcher moves from its current location to one of the neighbors.) We require $\mathcal{S}$ to decontaminate G, be monotone and *connected*, i.e., the *clear* subgraph, that is, the subgraph consisting of all clear edges, is connected after each move of the search strategy.

We now state the distributed model in the on-line setting that we use. All searchers start at the homebase and the network itself is not known in advance to the searchers (except for the fact that the searchers may expect that the network is a partial grid). We assume that nodes are anonymous and searchers have identifiers. The edges incident to each node are marked with unique labels (port numbers) and because only partial grids are considered in this work we assume that labels naturally reflect all possible directions for each edge (i.e., left, right, up and down). For the searchers, we assume that they communicate locally by exchanging information when present at the same node.

In Chapter 3, our main results consists of constructing a distributed algorithm `ModGridSearching` and proving:

– For each distributed algorithm A computing a connected monotone search strategy in the on-line setting there exists an n-node network

G with homebase h such that

$$\max_h \frac{A(G,h)}{A^{opt}(G,h)} = \Omega(\sqrt{n}/\log n)$$

(Lemma 3.3.1 and Theorem 3.3.1).

– The distributed algorithm `ModGridSearching` clears (starting at an arbitrary homebase) in a connected and monotone way any unknown underlying partial grid network of order n using $O(\sqrt{n})$ searchers. The algorithm receives no prior information on the network (Theorem 3.5.1 and Theorem 3.6.1).

1.3.2 Computation of Connected Pathwidth

Let $G = (V(G), E(G))$ be a simple, undirected, connected graph of order n.

Definition 1.3.1. *A* path decomposition *of a graph G is a sequence $\mathcal{P} = (X_1, \ldots, X_l)$, where $X_i \subseteq V(G)$ for each $i \in \{1, \ldots, l\}$, and*

A. $\bigcup_{i=1}^{l} X_i = V(G)$,

B. for each $\{u, v\} \in E(G)$ there exists $i \in \{1, \ldots, l\}$ such that $u, v \in X_i$,

C. for each i, j, k with $1 \leq i \leq j \leq k \leq l$ it holds that $X_i \cap X_k \subseteq X_j$.

The width *of a path decomposition $\mathcal{P}$ is* $\mathrm{width}(\mathcal{P}) = \max_{i \in \{1,\ldots,l\}} |X_i| - 1$. *The* pathwidth *of G, denoted by* $\mathrm{pw}(G)$, *is the minimum width over all path decompositions of G.*

We say that a path decomposition $\mathcal{P} = (X_1, \ldots, X_l)$ is *connected* if the subgraph $G[X_1 \cup \cdots \cup X_i]$ is connected for each $i \in \{1, \ldots, l\}$. The *connected pathwidth* of a graph G, denoted by $\mathrm{cpw}(G)$, is the minimum width taken over all connected path decompositions of G.

This version of the classical pathwidth problem is motivated by several pursuit-evasion games. More precisely, computing the minimum number of searchers $\mathtt{cs}(G)$ needed to decontaminate in a connected way a given graph G is equivalent to computing the connected pathwidth of G. Moreover, a connected path decomposition can be easily translated into the corresponding search strategy that cleans G and vice versa. Indeed, suppose that $\mathcal{P} = (X_1, \ldots, X_l)$ is a connected path decomposition of G. We can use it to obtain a search inductively as follows. Assuming that the searchers are located at the vertices in X_i, we perform the following moves. First, remove from G all searchers on the vertices in $X_i \setminus X_{i-1}$. Then place the

searchers on the vertices in X_{i+1} that are not occupied by any searchers. Finally, use an additional searcher to clear all edges with two endpoints in X_{i+1}. This completes the inductive construction. Using also an induction, one can prove that once the vertices in X_i are occupied by the searchers, all edges in the subgraph $G[X_1 \cup \cdots \cup X_i]$ are clear. Such a search uses $1 + \max_i |X_i|$ searchers, which is equal to $\mathtt{cs}(G)$ or $\mathtt{cs}(G) + 1$ [44, 99].[4]

In Chapter 4 (Theorem 4.4.1) we prove that for every fixed $k \geq 1$, there is an algorithm deciding in time $f(k) \cdot n^{O(k^2)}$ whether $\mathrm{cpw}(G) \leq k - 1$, for some function f depending on k only, i.e., in time polynomial in n.

1.3.3 Minimum-cost Graph Exploration

Let G be a simple, undirected, edge-weighted, connected graph of order n with a single homebase h and let $w : E \to \mathbb{R}^+$ be an edge-weight function.

The goal is to find an algorithm, which for any graph G computes a strategy, which explores G. In this problem, we consider strategies, that consists of two types of moves: (1) traversing an edge by an agent and (2) invoking a new agent in the homebase. Let $\mathcal{S}$ be a strategy constructed for some graph G, $k \in \mathbb{N}^+$ be the number of agents used by $\mathcal{S}$ (notice that k is not fixed) and $d_i \in \mathbb{R}^+ \cup \{0\}$ the distance traversed by the i-th agent during the execution of $\mathcal{S}$, $i \in \{1, \ldots, k\}$. Let q be the *invoking cost*. We define the *cost* of $\mathcal{S}$ as $c(\mathcal{S}) = kq + \sum_{i=1}^{k} d_i$. In other words, cost is understood as the sum of invoking costs and the total distance traversed by entities. Intuitively, before exploring any vertex the algorithm needs to decide what is more profitable: invoke a new agent (and pay for it q) or use an agent already present in the graph. The number of agents, that can be invoked, is unbounded.

For the off-line setting in Chapter 5 we present two algorithms that compute cost-optimal strategies for exploring rings and trees (Theorem 5.4.1), respectively.

In the on-line setting it is assumed that an agent, which occupies the vertex v, knows the length of edges incident to v and the *status* of vertices adjacent to v, i.e., if they have been already explored. We assume global communication for this problem. We construct a 2-competitive distributed algorithm for rings (Lemma 5.3.1) and prove a lower bound of $3/2$ of the competitive ratio for any on-line algorithm for rings (Theorem 5.3.1). For

[4] To be more precise, the (connected) pathwidth is equal to the (connected) *node search* number, which can differ from $\mathtt{s}(G)$ ($\mathtt{cs}$(G)) by at most one. For more details see the survey in the following section, in particular Subsection 2.1.1 for the description of different search models and Subsection 2.1.3 for relationships between search numbers and different graph parameters.

every algorithm for trees in the on-line setting, we prove the competitive ratio to be no less than 2 (Theorem 5.5.1), which can be achieved by the DFS algorithm.

1.3.4 The Link Up Problem

Let D be a vertex-weighted digraph with an additional subset of vertices, such that its underlying graph is connected. In other words, $D = (V(D), A(D), F, B)$ is a quadruple, where $V = V(D)$ is a set of all vertices, $A = A(D)$ set of arcs, $F \subseteq V$ and $B : V \to \mathbb{N}$ a vertex-weight function. Let $n = |V|$.

The link up problem is modeled by D as follows. The vertices of D correspond to terminals while its arcs correspond to (one-way) transmission links, the set F corresponds to locations of facilities, and the set $\mathcal{B} = B^{-1}(\mathbb{N}^+)$ corresponds to homebases, where a (positive) number of agents is placed (so we shall refer to the function B as an *agent-quantity* function). Let $k = \sum_{v \in V} B(v)$ be the total number of agents placed in the digraph.

The Link Up Problem (LU)

> Do there exist k directed walks in D, with exactly $B(v)$ starting points at each vertex $v \in V$, whose edges induce a subgraph H of D such that all vertices in F belong to one connected component of the underlying graph of H? Note that the k directed walks may overlap in vertices and even edges.

The LU problem may be understood as a question, whether for a team of size k, initially located at homebases in $\mathcal{B} = B^{-1}(\mathbb{N}^+)$, where the number of agents located at $v \in \mathcal{B}$ is equal to $B(v)$, it is possible to follow k walks in D clearing their arcs so that the underlying graph obtained by the union of cleared walks includes a Steiner tree for all facilities in F.

Our two main results in Chapter 6 are stated below:

- The LU problem admits a fixed-parameter randomized algorithm with respect to the total number l of facilities and homebases, running in $2^{O(l)} \cdot \mathrm{poly}(n)$ time, where n is the order of the input graph (Theorem 6.2.2 and Corollary 6.2.1).
- The LU problem is strongly NP-complete even for directed acyclic graphs $D = (V, A, F, B)$ with $F = V$ and $B(v) = 1$ if v is a source vertex in D and $B(v) = 0$ otherwise (Theorem 6.3.1).

Chapter 2

Survey

In this chapter we provide comprehensive surveys on the two distributed deterministic agents' problems on undirected graphs: decontamination (or searching, Section 2.1) and exploration (Section 2.2). The terminology from this chapter has been broadly used in the literature and is inspired by several publications such as those about on-line algorithms [66], distributed algorithms [115], complexity theory [87, 111], parameterized complexity [56] and approximation algorithms [147].

Let G be any simple, connected, undirected graph and let $n = |V(G)|$. In Section 1.2 we have formally defined a strategy and basic searching and exploration models. Let us now, as an introduction to surveys, look more at the different optimization factors and communication and time models.

Efficiency measures. We distinguish the following optimization factors:

- *number of agents* - the total number of agents used on a graph;
- *total moves* - the sum of all movements performed by agents;
- *completion time* - the number of *time units* required to complete the search, with the assumption that for edge-weighted graphs a walk along an edge e takes $w(e)$ time units (where $w(e)$ is the weight of the edge e) and for non-weighted graphs takes one time unit[1];
- *total distance* - the sum of distances traversed by all agents;
- *total clearing moves* - the sum of clearing movements performed by agents;
- *energy* - maximum value taken over all agents traversed distances;

[1] More precisely, it is the number of time units that passed from the first to the last move performed by agents.

– *cost* - the total distance traversed by agents coupled with the cost of invoking them, introduced in SOFSEM 2019 (see Chapter 5).

The minimal number of needed agents can stay in the contradiction to other optimization factors, as it was proved for the total clearing moves factor in [48] and can be easily observed e.g., for the completion time factor. We refer to the total moves and completion time factors also as the *moves* and *time complexity* of the strategy.

As the efficiency measure of on-line algorithms we use competitiveness (defined formally in Section 1.2), just in a general case we can compare an on-line algorithm with the optimal off-line one on any optimization factor (not only on the number of agents as it has been stated before).

Communication Between Agents. In this thesis, we assume that agents are not able to see each others' positions on a graph, but they can exchange with each other messages, i.e., *communicate*. Fraigniaud *et al.* show in [83], that without communication every on-line exploration algorithm performs $\Omega(k)$ times longer, where k is the number of agents, than the optimal off-line one (i.e., has the competitive ratio $\Omega(k)$). In other words, without communication each of the agents would simply explore the graph on its own. In the case of decontamination it is even impossible to achieve the goal without communication, while apart from trivial cases (e.g., paths), one agent is not able to clean the whole graph alone. Therefore some way of communication must be provided for agents. In literature, we can find two approaches: global and local communication. Global communication allows all agents to send and receive unlimited number of messages at any time in spite their locations, which is a very strong assumption, but reflects the real life ability of robots to communicate by, e.g., a WiFi network. In the local model, on the other hand, communication is limited and following approaches can be distinguished: *pebbles (tokens)*, *whiteboards*, *face-to-face* or *bounded* communication. Historically, the first studied model of pebbles allowed agents to drop and pick tokens in nodes in order to tag them, while the model of whiteboards provides agents a memory space on each node, to which they can write and read in the mutual exclusion. In the face-to-face communication, agents can exchange information only when they meet and in bounded model global communication is allowed but only on a specified distance, which reflects the real life situation of a bounded range of communicating equipment.

Time Clock. In the *synchronous* model all moves of agents are synchronized and are being made in regards to the common clock. In each time

unit an agent can decide to stay on its position or make one of allowed moves, which has the same unit length duration. Note here that agents know exactly how many units of time it will take other agents to perform their actions and based on that they can compute their next steps. On the other hand, in the *asynchronous* settings, a move of an agent takes an unknown but finite amount of time, which does not allow agents to make any assumptions based on time about other agents' current positions. Lastly, in the *quasi-synchronous* model [73] only an upper bound of the length of agents' moves is given. Intuitively, an asynchronous algorithm (i.e., an algorithm that perform in the asynchronous setting) can never perform better than a synchronous one (respectively, in the synchronous setting) for the same network, thus a good practice while investigating time or move complexity is to compute the lower bounds for synchronous and upper ones for asynchronous models.

Let us notice now that because usually we are interested in asymptotic results, the choice of the communication model and time setting are not very significant when we minimize the number of agents. Indeed, the choice of communication model affects, e.g., move and time complexity of search strategy [53], but (for the searching and exploration problems) it does not affect the optimal number of needed agents. The global environment can be simulated in the local setting by designating one searcher called the *leader* who at the beginning of each move visits all nodes of the cleaned subgraph in order to gather all needed information to compute agents' next moves. Leader coordinates moves of the whole team, by informing the searcher, who is supposed to perform the next move.

This simple explanation also works for transposing a strategy in a synchronous environment into the asynchronous setting. In other words, the choice of the time model is not significantly relevant when minimizing the number of agents, although it affects most of the other optimization factors[2]. Moreover, one additional searcher in the asynchronous setting is often not only sufficient, but required in order to make the search or exploration feasible.

[2]In an asynchronous models for mobile agent computing, while calculating the completion time it is often assumed that one move takes one time unit.

2.1 Decontamination

Let $\mathcal{G}$ be a class of connected, undirected, simple graphs. For every $G \in \mathcal{G}$ we denote the number of vertices and edges as $n = |V|$ and $m = |E|$ respectively. Recall that the decontamination problem is to construct a deterministic search strategy, which clears initially infected network. It has been extensively studied due to its various application in the computer science area, where protecting a network from hostile viruses is crucial, as well as in the robotic field, where robots need to cooperate in a real-life terrain (which can be modeled as a graph) in order to achieve a common goal, e.g., decontaminate a set of polluted tunnels, build a map or catch a hostile intruder.

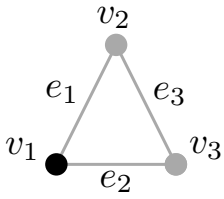
In this whole subsection we assume that graphs are non-weighted, unless it is said differently. This chapter is based on the survey accepted for publication in *Utilitas Mathematica* [128].

2.1.1 Search Models

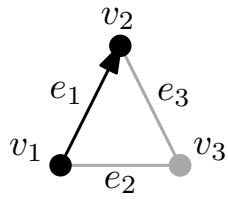
Edge Search. The basic and historically first model of graph searching, studied in [132, 133], is the *edge search* in which initially all vertices and edges of the network are *contaminated* and the goal is to find a strategy, which will allow searchers to *clean* the whole network. The search strategy is understood as a sequence of moves, where each move is one of the following: (1) placing a searcher on a node, (2) sliding a searcher along an edge and (3) removing a searcher from the node it occupies. A contaminated node becomes clean if a searcher is placed on it and an edge becomes clean if a searcher slides along it. A node v (or edge e) becomes *recontaminated* (contaminated again) if after a move there exists a free of searchers path containing v (or respectively e) and any infected node or edge. We will refer to the vertex which is occupied by a searcher and at least one of its incident edges is contaminated as to a *guarded* one. See Figure 2.1 for an example of an edge search decontamination.

Node and Mixed Search. Another approach given in [16] is the *node search* variant, where only moves of type (1) and (3) are allowed (sliding along an edge is forbidden) and an edge becomes clean, when two its incident nodes are simultaneously occupied by searchers. Combining both ways of clearing edges one obtains the *mixed search* variant. See Figure 2.2 for an example of the node search.

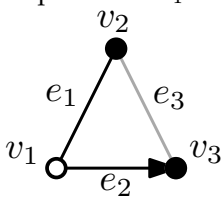
Node-decontamination. In the *node-decontamination* (oppositely to *edge-decontamination* or simply *decontamination*) problem the fugitive can hide only in nodes, so there is no need to clear edges [114]. In this model an unoccupied node stays clean only if all of its neighbors are clean and get recontaminated otherwise. Searchers here are allowed to make all of three moves, as in the edge search. Interestingly, in the literature (e.g., [69, 67]) node search and node-decontamination concepts are used interchangeably, which can be sometimes misleading. Moreover, the node-decontamination model is (in practice, not in the assumptions) equivalent to the mixed search one. It is because in articles written in the same time different authors used different terminology to describe their models. One of the goals of this survey was to introduce a common nomenclature and classify models by it. The minimum number of searchers needed to clean in this model can be different than the one required in the node search variant - for the example from Figure 2.2 for the node-decontamination problem, only two searchers are needed, which perform the total of three moves, i.e., place one searcher on v_1, place one searcher on v_2, slide a searcher from v_1 to v_3.



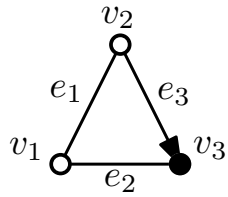
(A) Initially graph is empty and in first two moves two searchers are placed on v_1.



(B) In the third move the searcher from v_1 is slid along the edge e_1.

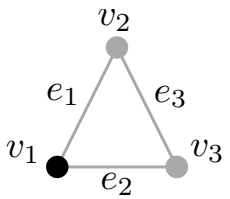


(C) In the forth move the searcher from v_1 is slid along the edge e_2.

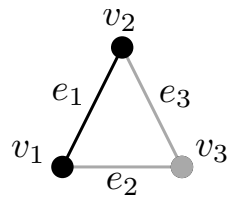


(D) Finally one of the searchers cleans the remaining edge e_3.

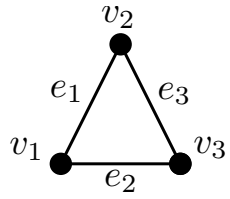
FIGURE 2.1: Decontaminating a cycle C_3 in the *edge search* model; *gray dots* and *edges* denote contaminated elements, *black dots* are the nodes occupied by searcher(s), *black edges* stand for the cleaned edges, *black arrows* shows the cleaning movements of the searchers and *black circles* denote the cleaned nodes without any searcher. Optimal (in the sense of the number of searchers) cleaning of C_3 requires two searchers and five moves.



(A) Initially graph is empty and one searcher is placed on v_1.



(B) In the second move a searcher is placed on v_2.



(C) Lastly the third searcher is placed on v_3, which immediately cleans the remaining two edges.

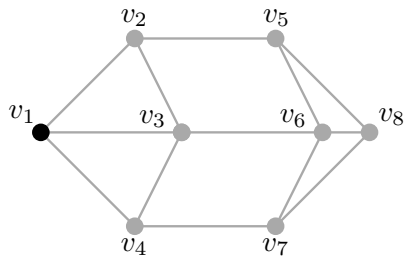
FIGURE 2.2: Decontaminating a cycle C_3 in the *node search* model. Optimal (in the sense of the number of searchers) cleaning of C_3 requires three searchers and three moves.

Lets us notice here that when the number of searchers is being optimized, all these models will not differ by more than a small constant (see Section 2.1.2). As for the relation between search variants and moves complexity, apart from the work of Dereniowski and Dyer [48], where the asymptotic difference for cliques is shown, no significant results were established. We notice also that by combining moves into steps one can redefine given strategies for the previous example in order to decrease the number of steps (i.e., completion time) by 2. At the end of this subsection let us present an example of recontamination in the mixed search model in the Figure 2.3, where we have combined placing and sliding moves of searchers into steps.

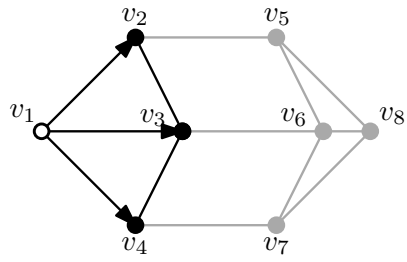
2.1.2 The Search Number

Search Number. The *search number* $\mathtt{s}(G)$ of a network G is the smallest number of searchers in the edge search variant for which a search strategy exists.

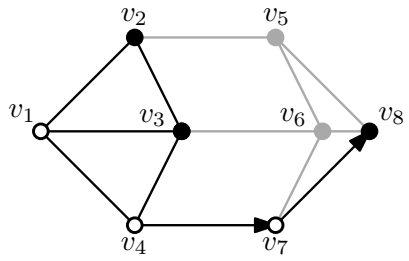
Monotonicity. In a *monotone* search strategy a recontamination of the nodes is forbidden - once a node or an edge is cleared by a searcher, it must



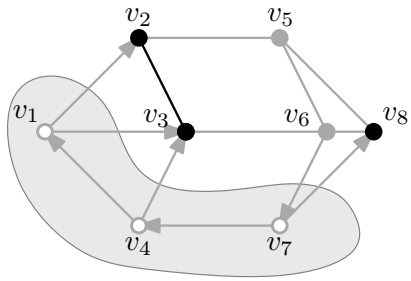
(A) Initially graph is empty and in the first step three searchers are placed on v_1.



(B) In the second step each of them slides along one of the incidental edges.



(C) In the next two steps searcher from the node v_4 traverses to v_7 and then to v_8, leaving v_7 unprotected.

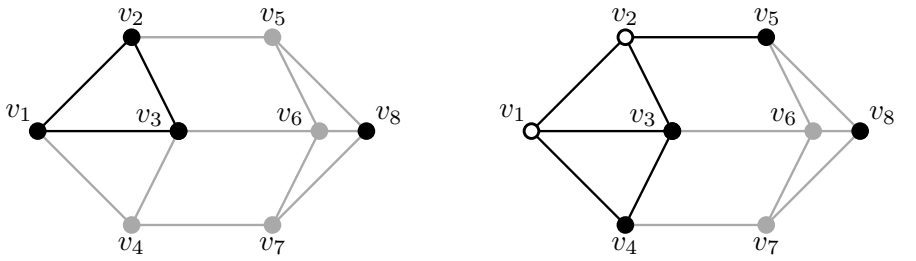


(D) Recontamination immediately spreads from v_6 to the unprotected cleaned part of the graph.

FIGURE 2.3: An example of recontamination of the nodes in the mixed search model; *gray arrows* denote directions of spreading of recontamination, *gray circles* denote the recontaminated nodes and *the gray area* shows the recontaminated part of the network.

stay clean till the end of the procedure. It forces searchers to guard vertices that have contaminated neighbors and only a vertex with the entire neighborhood cleaned can be left empty. Monotone strategies are of particular interest for many reasons due to their broad applications in e.g., computer and telecommunication networks. Because the number of cleaned nodes do not decrease, monotone strategies perform in a polynomial number of steps and for most of the models allowing recontamination does not improve the results (see inequalities 2.1 - 2.3). It is also *a priori* difficult to design a non-monotone search strategy. Additionally, if we do not assume monotonicity in the on-line case we will reduce it to the off-line one, by just sending one searcher to explore the whole graph, return to the homebase and share its knowledge with other searchers allowing them to compute an

optimal strategy and then execute it[3]. The minimum number of searchers needed to search a graph G in the edge search model in the monotone way is denoted by $\mathtt{ms}(G)$. See Figure 2.4 for an example of the allowed movements in the monotone search strategies.



(A) In order to ensure monotonicity, only two moves are allowed: from v_2 to v_5 and from v_1 to v_4.

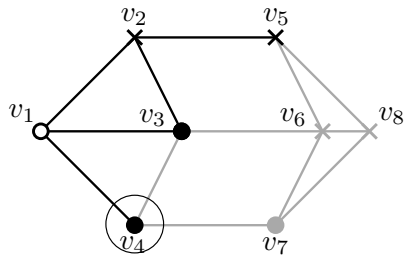
(B) After performing this two moves new possibilities have arisen: from v_3 to v_6 and from v_4 to v_7.

FIGURE 2.4: An example of allowed moves in the monotone search strategy in the edge search model, where each of the four vertices $\{v_1, v_2, v_3, v_8\}$ is occupied by one searcher.

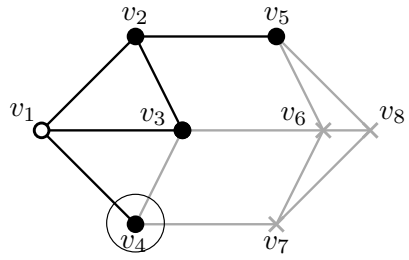
Connectivity and Internality. We call a strategy *connected* if the clean part of the graph forms a connected subnetwork in every step of the strategy. In the *internal* model we forbid searchers to *jump*, i.e., only the moves of sliding a searcher along a link and placing a new searcher on a homebase is allowed. Model with these two properties was first introduced in [7] and called *contiguous*. This assumptions are dictated by the real life problems, which include performing tasks by robots, which do not posses the ability of jumping between two remote places. The minimum number of searchers needed to search a graph G in the edge search model in the connected way is denoted by $\mathtt{cs}(G)$ and internal by $\mathtt{is}(G)$. In order to obtain different search numbers one can combine described properties and denote them according to logical reasoning (e.g., $\mathtt{mis}$ stays for the monotone internal search number and $\mathtt{ics}$ for contiguous one). Note that although connected and contiguous strategies might perform differently (see Figure 2.5 for an example) they are equivalent in terms of used searchers (i.e., $\mathtt{cs}(G) = \mathtt{ics}(G)$), since every jumping move can be replaced by a sequence of sliding moves.

It is natural to study search numbers in the off-line setting since no online algorithms can perform better (in any sense). Although determining

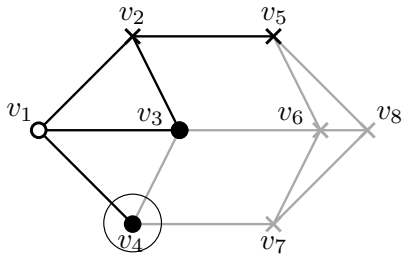
[3]This simple explanation holds only for the strategies optimal in the sense of the number of searchers, where agents dispose of the memory large enough.



(A) In the internal strategy the searcher from v_4 is only allowed to move to any of its neighbors.



(B) In the connected strategy the searcher is forbidden to move to its neighbor v_7, but is allowed to jump to v_2 or v_5.



(C) In the contiguous strategy searchers need to abide both restrictions.

FIGURE 2.5: An example of the internal, connected and contiguous search numbers; *crosses* denote nodes forbidden for the searcher occupying the circled node.

whether $\mathtt{s}(G) \leq k$ for an arbitrary graph G and $k \in \mathbb{N}^+$ belongs to the class of NP-complete problems [118], the distributed and centralized off-line graph searching strategies are well studied and numerous deep results were established, whose summary is presented below. First results can be found in the articles written around the year 1980, where the edge search model was defined [133, 132]. In [118] and [16] authors prove that every graph can be searched in the monotone way and the famous statement: *Recontamination does not help* comes from the article of the same title [109], where LaPaugh shows the monotonicity of the edge search problem:

$$\mathtt{ms}(G) = \mathtt{s}(G). \tag{2.1}$$

Ten years later Barrière *et al.* in [10, 8, 7] present a summary of all properties condensed in two theorems, for an arbitrary graph G

$$\begin{aligned} \mathtt{is}(G) = \mathtt{s}(G) = \mathtt{ms}(G) \leq \mathtt{mis}(G) \leq \\ \leq \mathtt{cs}(G) = \mathtt{ics}(G) \leq \mathtt{mcs}(G) = \mathtt{mics}(G) \end{aligned} \tag{2.2}$$

and tree T

$$\begin{aligned} \mathtt{is}(T) = \mathtt{s}(T) = \mathtt{ms}(T) \leq \mathtt{mis}(T) = \mathtt{cs}(T) = \\ = \mathtt{ics}(T) = \mathtt{mcs}(T) = \mathtt{mics}(T) \leq 2\mathtt{s}(T) - 2. \end{aligned} \tag{2.3}$$

In [77] bounds for a special class of outerplanar graphs are presented, i.e, for any 2-connected outerplanar graph G with its weak dual T^*

$$\mathtt{cs}(T^*)/2 \leq \mathtt{cs}(G) \leq 2\mathtt{cs}(T^*). \tag{2.4}$$

Interestingly the monotonicity condition has no influence on the optimal number of searchers for trees. This does not hold for an arbitrary graph G: in [151] Yang *et al.* constructed special kind of graphs with large clique number for which they proved that the strictness of an inequality $\mathtt{cs}(G) < \mathtt{mcs}(G)$ is possible.

For brevity an algorithm, which computes only monotone (connected) strategies, is called a monotone (connected) algorithm.

The Cost of the Connectivity and Monotonicity. Being aware of the inequality between regular and connected search number, one may wonder how many extra searchers must be provided to ensure the connectivity. The price of the connectivity for an arbitrary graph G was first estimated by Barrière *et al.* in [9, 10] and then improved by Dereniowski in [47] giving a final result $\mathtt{cs}(G) \leq 2\mathtt{s}(G) + 3$, where the factor 2 is tight. In the case of trees, we obtain a tight upper bound of 2: $\mathtt{cs}(T) < 2\mathtt{s}(T)$ [9, 10]. As for the monotonicity, it does not change the search number $\mathtt{ms}(G) = \mathtt{s}(G)$, but influences the connected search number, i.e., $\mathtt{cs}(G) \leq \mathtt{mcs}(G)$ (as it was already mentioned).

The Cost of Knowledge. For an on-line algorithms the following theorem [93] holds: for any monotone connected on-line algorithm A there exists a constant c such that for any sufficiently large n, there exist a n-node graph G, such that a strategy computed by A for G requires at least $c\frac{n}{\log n}\mathtt{mics}(G)$ searchers to clear G, i.e., the competitive ratio $r(A)$ is upper

bounded by

$$r(A) \leq \Omega\left(\frac{n}{\log n}\right). \tag{2.5}$$

Ilcinkas *et al.* [93] propose also an on-line algorithm which for any G uses at most $O\left(\frac{n}{\log n}\right)\texttt{mics}(G)$ searchers thereby making the gap tight. Moreover these results are insensible of the time environment - a lower bound holds in a synchronous setting and the algorithm can be implemented in an asynchronous one. On the other hand, [22] shows that by neglecting the monotone property it is possible to construct an on-line algorithm for an arbitrary graph working in the asynchronous environment that needs only $\texttt{mics}(G)+1$ searchers, giving the same competitive ratio of 1, but performs in exponential time and is not monotone.

2.1.3 Search Numbers for Different Search Models

The search number originally was defined for the edge search model, but as soon as new search variants were defined authors started to distinguish it among them obtaining: the *node search number* $\texttt{ns}(G)$ and the *mixed search number* $\texttt{ens}(G)$, for any graph G. Kirousis and Papadimitriou in 1986 [99] were first who observed that the node and edge search numbers can differ by at most one: $\texttt{ns}(G)-1 \leq \texttt{s}(G) \leq \texttt{ns}(G)+1$, while Takahashu *et al.* [146] completed this results by proving for mixed search model: $\texttt{s}(G)-1 \leq \texttt{ens}(G) \leq \texttt{s}(G)$ and $\texttt{ns}(G)-1 \leq \texttt{ens}(G) \leq \texttt{ns}(G)$.

Interestingly, graph searching number is strictly linked to the *pathwidth* $pw(G)$ and *vertex separation* $vs(G)$ parameters (which are well known to be equal [98]), as it has been shown: $\texttt{ns}(G) = vs(G)+1 = pw(G)+1$ [44, 99]. With the previous observation it leads to the inequality: $vs(G) \leq \texttt{s}(G) \leq vs(G)+2$ formally proven tight in 1994 by Ellis *et al.* [63]. For mixed search, a new parameter, called *proper-pathwidth* and denoted by $ppw(G)$ was defined and following results were established [146]:

$$pw(G) \leq \texttt{ens}(G) = ppw(G) \leq pw(G)+1. \tag{2.6}$$

For more results for different graph parameters (i.e., bandwidth, branchwidth, cutwidth, treewidth) and their connection to graph searching see a comprehensive review on the topic [76].

2.1.4 Monotone Contiguous Search

In this subsection results for monotone contiguous search are presented, where searchers are assumed to start their task from one homebase, which

is a part of an input. Firstly, a closer look at a problem of the amount of information provided *a priori* for searchers is given. Then, in following two paragraphs all known results (to the best of this author's knowledge) for monotone contiguous decontamination problem are described and presented in Tables 2.1-2.4. Models with *visibility* and *cloning* properties are considered, where the first one provides for searchers additional information about the neighborhood nodes of a currently occupied one, and the second gives searchers an ability to make a copy of themselves before they perform a move. The two last paragraphs say about the models with *immunity* (see Tables 2.5-2.7 for the summary) and *exclusivity* properties.

Bits of Advice. The common practice for on-line algorithms is to investigate the amount of additional *a priori* knowledge about the network which has to be provided for searchers in order to accomplish their task efficiently. Fraigniaud *et al.* introduced in [80] a new measure of difficulty for on-line tasks called bits of *advice*, which is the minimum number of bits of information, which have to be provided for searchers to make the decontamination task possible. First results in the subject of searching and exploration can be found in [81], where authors consider one-agent exploration of trees. The most significant results in the multi-agents decontamination task are provided by Nisse and Soguet in [121], where the amount of bits of advice for the problem of finding $\texttt{mics}(G)$ for any unknown graph G in the synchronous environment is investigated. Authors define an oracle using a total of $O(n \log n)$ bits and the algorithm that solves the monotone contiguous search problem with this oracle in $O(n^3)$ time steps. Moreover, no algorithm using an oracle providing $o(n \log n)$ bits of advice permits to clear monotonically using `mics` searchers, so the bound is asymptotically tight. In other words, it is possible to construct an optimal monotone contiguous search strategy for an arbitrary unknown graph G if in every node a whiteboard with $O(\log n)$ bits of information is provided.

An interesting algorithm for monotone contiguous graph search for any unknown weighted graph G can be found in [27], where Borowiecki *et al.* equipped searchers in an additional ability, called *sense of direction*. A partition $(V_1, \ldots, V_t)$ of graph's G node set is given, such that edges are allowed only within each V_i and between two consecutive V_i's. Searchers have no *a priori* knowledge about the graph, but can recognize whether an edge incident to already explored vertex in V_i leads to a vertex in one of V_{i-1}, V_i or V_{i+1}. Giving the size of advice $O(|E|)$ authors present an algorithm, which for G computes a strategy that requires $3 \max_{i=1,\ldots,t} \omega(V_i) + 1$ searchers, where $\omega(V_i)$ is the sum of weights of all nodes in the set V_i.

Every n-node graph can be searched by this technique, in the worst scenario only one-element partition is created and the strategy uses $3n + 1$ searchers. See Figure 2.6 for an example.

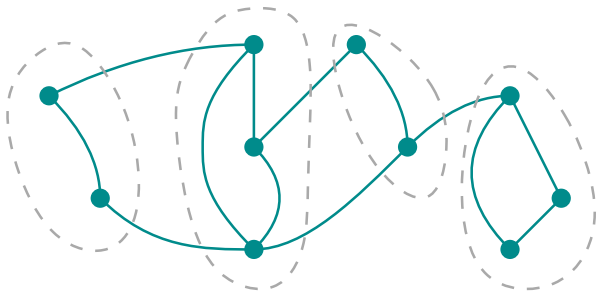


FIGURE 2.6: An example of division of vertices into the four partitions. When a searcher occupies a vertex, it knows if the adjacent edge leads to a vertex lying inside the left, right or the current partition.

Different Topologies. Whereas the problem of finding the search number for arbitrary graphs is NP-hard, it can be solved in polynomial time for specified network topologies. It was Barrière *et al.* who first introduced monotone contiguous search number and provided an algorithm, that for any given tree T computes an optimal search strategy, proving the same equality $\texttt{mics}(T) = \texttt{ics}(T) = \Theta(\log n)$ [7]. Similar results however cannot be transferred to weighted trees, as Dereniowski shows in [46], the problem of connected searching even for node-weighted trees is strongly NP-complete [4]. The same author gives in [45] the 3-approximation algorithm, which can be computed in polynomial time. If the tree is not known in advance, then for every monotone connected algorithm there exist some trees for which $\Omega(n)$ searchers are needed, which holds even for binary trees [93]. These properties are highly unsatisfying, since it is possible to search every graph with $O(n)$ searchers, simply leaving one searcher on each node.

In the off-line setting, other topologies were investigated and monotone contiguous search algorithms were constructed for chordal rings and toroidal meshes [67, 92], hypercubes [68, 92], butterflies [92], Sierpinski graphs [114], meshes [69, 134], pyramids [142], product networks [94] and star graphs [95]. In [134] Qiu looked at the problem from a different perspective of maximizing the number of nodes that can be cleared for a given number of k searchers. Apart from the number of searchers, most of these

[4] Apart from the NP-completeness Dereniowski constructs an FPT algorithm for finding an optimal strategy with the respect to the maximum degree of a tree.

works analyze the completion time and total moves parameters and the influence of cloning and visibility abilities (which are broader discussed in the next paragraph).

As for the on-line setting, authors of [51] present an algorithm for decontaminating partial grids, which for n-node grids computes strategies, which use $O(\sqrt{n})$ searchers and give a lower bound for the competitive ratio $\Omega(\sqrt{n}/\log n)$ (see Chapter 3 for details).

See Tables 2.1-2.4 for the summary of all known (to the best of this author's knowledge) results for the monotone contiguous decontamination problem.

Ref.	Number of Searchers	Computational Complexity	Bits of Messages
[7]	GIVEN n-NODE TREE T		
	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
[22]	UNKNOWN n-NODE TREE T		
	$\Theta(n)$		
[46]	GIVEN n-NODE NODE AND/OR EDGE WEIGHTED TREE T WITH A MAXIMUM NODE DEGREE Δ		
	NP-complete		
[45]	$3\texttt{mics}(T)$	$O(\Delta n^3 \log n)$	
[51]	UNKNOWN n-NODE PARTIAL GRID G		
	$O(\sqrt{n})$		
	$r = \Omega(\sqrt{n}/\log n)$		

TABLE 2.1: Asymptotic results for distributed algorithms (in on-line and off-line settings) for trees and for partial grids in the monotone contiguous model. By Computational Complexity is understood the time needed to compute the algorithm and by Bits of Messages, the amount of bits of messages exchanged by searchers while performing the strategy; r stays for the competitive ratio for any algorithm.

Visibility and Cloning. Property of *visibility d* (or *the local knowledge at distance d*) provides a searcher additional information about all nodes at distance d and less from the one currently being occupied. In particular, it means that a searcher is able to read their whiteboards (if a model assumes ones) and recognize their states (clear, contaminated and occupied). By the strongest assumption of the visibility 0 (*local* model) is understood that

Ref.	Model	Number of Searchers	Total Moves	Completion Time
[69]	GIVEN MESH $M_{m\times n}$, $m \geq n$, $\|V\| = mn$, $\|E\| = 2mn - m - n$			
	NoSyn	$n+1$	$(n^2 + 4mn - 5n - 2)/2$	$mn - 2$
	NoVis	n^*	$(n^2 + 2mn - 3n)/2$	$m + n - 2^*$
	EdSyn	$n+1$	$(n^2 + 4mn - 3n - 2)/2$	$mn + n - 2$
	EdVis	$n+1$	$(n^2 + 4mn - 5n)/2$	$mn - n$
[67]	GIVEN TOROIDAL MESH $Z_{m\times n}$, $m \geq n$, $\|V\| = mn$, $\|E\| = 2mn$			
	No	$2n + 1^*$	$2mn - 4n - 1$	$mn - 2n$
	NoVis	$2n^*$	$mn - 2n^*$	$\lceil \frac{m-2}{2} \rceil^*$
[67]	GIVEN d-DIMENSIONAL TOROIDAL MESH $Z_{n_1\times\ldots\times n_d}$, $n = n_1 \times \ldots \times n_d$, WHERE $n_1 \leq \ldots \leq n_d$			
	No	$2\frac{n}{n_d} + 1$	$2n - 4\frac{n}{d_n} - 1$	$n - 2\frac{n}{n_d}$
	NoVis	$2\frac{n}{n_d}$	$n - 2\frac{n}{n_d}$	$\frac{\lceil n_d - 2 \rceil}{2}$

TABLE 2.2: List of the monotone contiguous distributed off-line search algorithms for meshes and toroidal meshes in the local communication model (face-to-face or whiteboard), where * marks the optimal results. Moves of searchers are being performed in parallel, i.e., combined into steps, which decreases the completion time, understood as the number of steps of the strategy. In the *Model* description: *No* - node-decontamination; *Ed* - edge-decontamination, edge search; *Syn* - synchronous settings (otherwise model is asynchronous); *Vis* - visibility in the distance 1 (otherwise visibility 0) and *Cl* - cloning ability (cloning is performed on demand and it is not considered as a move if it is not specified otherwise).

Ref.	Model	Number of Searchers	Total Moves	Completion Time
[68]	GIVEN d-DIMENSIONAL HYPERCUBE H, $n = 2^d$, $m = d2^{d-1}$			
[92]	No/Ed	$\Theta\left(\frac{n}{\sqrt{\log n}}\right)$	$O(n \log n)$	$O(n \log n)$
	NoCl	$\frac{n}{2}$	$O(n \log n)$	$O(n \log n)$
	NoVis	$\frac{n}{2}$	$\frac{n}{4}(\log n + 1) =$ $= O(n \log n)$	$\Theta(\log n)$
	NoVisCl	$\frac{n}{2}$	$n - 1^*$	$\log n^*$
	NoSynCl	$\frac{n}{2}$	$n - 1^*$	$\log n^*$
	EdSynCl	$\frac{d}{2}\binom{d}{d/2} = O(n)$	$\frac{n}{2} \log n^*$	$\log n^*$
	EdCl	$\frac{d}{2}\binom{d}{d/2} = O(n)$	$\frac{n}{2} \log n^*$	$\log n^*$
[92]	GIVEN d-DIMENSIONAL BUTTERFLY B, WHERE $n = 2^d(d+1)$, $m = d2^{d+1}$			
	Ed/No	$2^d = \frac{n}{d+1}$	$(3d-3)2^d + 4 =$ $= O(n \log n)$	$3d - 1$
	NoCl	$2^d = \frac{n}{d+1}$	$d2^{d+1}$	$3d - 1$
	EdCl	$2^d = \frac{n}{d+1}$	$d2^{d+1*}$	$3d - 1$
	+Syn +Cl	the same		
[67]	GIVEN n-NODE CHORDAL RING $C(\langle d_1 = 1, \ldots, d_k \rangle)$, WHERE $4 \le d_k \le \sqrt{n}$			
	No	$2d_k + 1^*$	$4n - 6d_k - 1$	$3n - 4d_k - 1$
	NoVis	$2d_k^*$	$n - 2d_k^*$	$\lceil \frac{n-2d_k}{2(d_k - d_{k-1})} \rceil$
[92]	WHERE $1 \le d_k \le \lfloor \frac{n}{2} \rfloor$			
	Ed	$2d_k + 1$	$2kn + 2d_k^2 +$ $-2d_k + 2$	$2kn - d_k + 2$
	No	$2d_k + 1$	$4n + 2d_k^2 - 5d_k$	$4n - 4d_k$
	NoSyn	$2d_k + 1$	$2n + 2d_k^2 - 2d_k$	$n + d_k - 1$
	+Cl	the same	$2d_k - 1$	the same
[142]	GIVEN d-LEVEL PYRAMID P, $n = (4^{d+1} - 1)/3$			
	NoVisCl	$2^{d+1} - 1 =$ $= O(\sqrt{n})$		$3 \cdot 2^d - 1 =$ $= O(\sqrt{n})$
	NoVisCl	$2^{2d} = O(n)$		$4d = O(\log n)^*$

TABLE 2.3: Monotone contiguous model with local communication (face-to-face or whiteboard); $n = |V|$ and $m = |E|$ is the number of nodes and edges respectively in the given network (distributed off-line algorithms).

Ref.	Model	Number of Searchers	Total Moves	Completion Time
[114]	GIVEN d-DIMENSIONAL SIERPIŃSKI GRAPH G WHERE $n = \frac{3^d+3}{2}$, $m = 3^d$			
	No	$d+1^*$	$O(d3^d)$	$6 \cdot 3^{d-2} - 1$
	NoVis	$d+1^*$	$O(d3^d)$	$\frac{7 \cdot 3^{d-2}-1}{2} - 1$
[94]	PRODUCT NETWORK OF k GIVEN GRAPHS $P = G_1 \times \ldots \times G_k$, WHERE $n_1, \ldots, n_k \leq n$			
	NoSynCl	$\texttt{mics}(G_1) \prod_{i=2}^{k} n_i$	$\sum_{i=1}^{k} \mathcal{M}(G_i)$	
	NoSynCl	$O(n^k)$	$O(n^k)$	
[95]	GIVEN d-DIMENSIONAL STAR S, $n = \lvert V \rvert = d!$, $\lvert E \rvert = \frac{(d-1)d!}{2}$			
	NoVisCl	$1 + \log_3(n-1)$		$3\lfloor \frac{3(d-1)}{2} \rfloor - 2$

TABLE 2.4: Continuation of Table 2.3. Cloning here is assumed as a separate move, which takes one time step. For any graph G, $\mathcal{M}(G)$ denotes the minimal number of total moves.

searchers have knowledge only about the currently occupied node, which was the case in the previous sections.

The ability of *cloning* allows searchers to make a copy of themselves before they perform a move. Their clones get unique id numbers and start acting like a regular, full-fledged searchers initially based on the node of their creation. In order to limit the number of needed searchers, cloning property is attended with the *terminate* one, which (colloquially saying) equips searchers in the ability to commit a suicide.

Distributed off-line algorithms in models concerning cloning and visibility 1 (or simply: visibility) properties for specified topologies can be found in [67, 68, 69, 92, 94, 95, 114, 134, 142] (see Tables 2.2-2.4 for a summary). Generally, cloning ability allows to minimize the sum of moves of the searchers and the number of steps of a strategy, but may increase the number of searchers. With the visibility property there is no need for an additional searcher (the leader) to coordinate other searchers moves decreasing the same moves and time complexity in the asynchronous environment. Interestingly, in the visibility model, changing the settings from asynchronous to the synchronous ones appears not to provide any benefit, although it is still considered as an open problem [68].

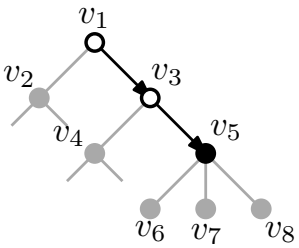
As for the on-line algorithms, an experimental analysis of models with visibility 0, 1 and 2 with and without cloning has been performed in

[71, 72], where a monotone, without jumping BFS algorithm in the synchronous environment is considered. Authors present results and relations between the search number, the total move complexity and the amount of homebases, which they compare to the one obtained by applying genetic algorithms methods.

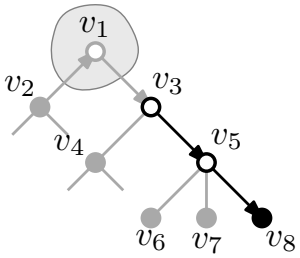
Immunity. An interesting change to the previously described model is the assumption of different conditions to occur in order for a node to be recontaminated. In the literature two concepts were introduced: *temporal* and *threshold immunity*. In the first model, recontamination of a node occurs after constant exposing it to a danger for a given *immunity time* $t \geq 0$ [70]. See Figure 2.7 for an example of a recontamination under the temporal immunity. A monotone contiguous synchronous algorithms and bounds of the number of needed searchers, total moves and bits of exchanged messages by searchers for meshes, toroidal meshes and trees for fixed t are presented in [70, 40] (see Table 2.5 for the summary). For a fixed number of searchers we can ask for the value of smallest t for a given graph, required to make the task of decontamination possible. This parameter was introduced by Daadaa *et al.* in [41] and referred to as the *immunity number* parameter. The authors analyze monotone and nonmonotone strategies for different topologies, including trees and meshes, in the synchronous environment for a single searcher. Let us notice here that in this model a synchronous strategy can not be easily transmitted to an asynchronous one (or local communication to the global one), while the simple leader explanation does not hold due to the time limitations.

The second approach is to recontaminate a node when a specified number m (or greater) of its neighbors is contaminated, which we refer to as the *m-Immunity* property. Monotone contiguous strategies and bounds of the number of needed searchers and total moves in the *local-majority* model (an unoccupied, cleared node is reinfected when more than a half of its neighbors are contaminated) for multi-dimensional toroidal meshes, graphs of vertex degree at most three and tree networks, are presented in [113] (see Table 2.6 for a summary). Broader results for all m for multi-dimensional meshes, toroidal meshes, multi-dimensional hypercubes and trees are provided by [73, 74, 112], where strategies optimal in the sense of the number of searchers and total performed moves are constructed and upper bounds of completion time provided. See Table 2.7 for a summary and Figure 2.8 for an example of a recontamination under the threshold immunity.

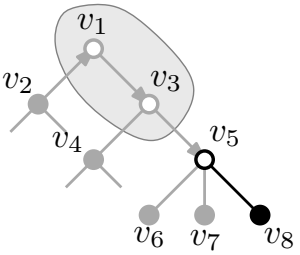
Apart from the upper bound of the minimal number of needed searchers for trees of a given height h [113], all of described results hold



(A) A searcher is placed on the home-base v_1 and slides along to v_3 and v_5.



(B) When the searcher moves to v_8 three unit of time has passed since he left v_1 unprotected, thus v_1 becomes re-contaminated.



(C) In the next step, the searcher stays in v_8 and recontamination spreads to v_3.

FIGURE 2.7: An example of recontamination under the temporal immunity, where $t = 2$.

TEMPORAL TIME IMMUNITY WITH TIME PARAMETER t		
Ref.	Number of Searchers	Total Moves Bits of Messages
[40]	GIVEN MESH $M_{m\times n}$, $m \geq n$	
	$\mathcal{A}(M_{m\times n}) \leq \min\left\{\lceil \frac{m}{\lceil t/2 \rceil} \rceil, \lceil \frac{2m-1}{t} \rceil\right\}$	
[40]	GIVEN TOROIDAL MESH $Z_{m\times n}$, $m \geq n$	
	$\mathcal{A}(Z_{m\times n}) \leq \min\left\{\lceil \frac{m}{\lceil t/2 \rceil} \rceil, \lceil \frac{2m-1}{t} \rceil\right\}$	
[70]	GIVEN n-NODE TREE T	
	$\mathcal{A}(T) = \Omega(\log_3(\frac{n}{t+1}))$	$\mathcal{T}(T) = \Theta(n)$ $\mathcal{B}(T) = \Theta(n)$
	UNKNOWN TREE T OF A GIVEN HEIGHT h	
	$\mathcal{A}(T) \leq \lfloor \frac{2h}{t+2} \rfloor$	

TABLE 2.5: Summary of the results for the monotone contiguous node-decontamination problem in the synchronous setting with the t-time immunity property (an unoccupied, cleared node becomes recontaminated after t time steps of being continuously exposed to danger) in the local communication model (face-to-face or whiteboard); for a graph G: $\mathcal{A}(G)$, $\mathcal{T}(G)$ and $\mathcal{B}(G)$ denote the minimal number of searchers, time steps and bits of messages exchanged by searchers, respectively.

under the assumption of the complete knowledge about a graph. Nevertheless some might be extended for the on-line setting, e.g., it is easy to notice, that for the local-majority model for any binary tree, single searcher is always sufficient, although in order to minimize the number of its moves the diameter of a tree has to be known *a priori* [113].

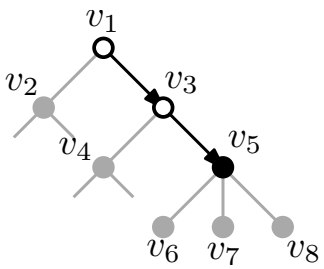
Exclusivity. The *exclusivity* property restricts any two searches to occupy the same node at the same time. This assumption is dictated by the real life problem of placing several searchers at the same point in a physical environment. Also it prevents software agents deployed in a computer network from consuming too much local resources (e.g., memory, computation cycles). In [21] authors prove several significant results for the exclusive search number $\mathtt{xs}$. Firstly, it does not satisfy the subgraph–closeness property, i.e., there exist graphs H and G, where H is a subgraph of G and $\mathtt{xs}(H) > \mathtt{xs}(G)$. Secondly the equality $\mathtt{xs}(G) = \mathtt{xms}(G)$ does not hold, i.e., an optimal strategy does not have to be monotone and finally the disparity between $\mathtt{xs}$ and different search variants (edge search, node search) can be arbitrarily large. Study of exclusive searching in the mixed search model

LOCAL-MAJORITY IMMUNITY		
Ref.	Number of Searchers	Total Moves
[113]	GIVEN d-DIMENSIONAL TOROIDAL MESH Z_n, $n = n_1 \times \cdots \times n_d$	
	SYNCHRONOUS WITH LOCAL COMMUNICATION MODEL	
	$\mathcal{A}(Z_n) = 2^d$	If every n_i, $i = 1, \ldots, d$ is even: $\mathcal{M}(Z_n) \leq (d-2)2^{d-1} + O(n)$, else: $\mathcal{M}(Z_n) \leq (d-2)2^{d-1} + n$. $\mathcal{M}(Z_n) = n - 1$, $d = 1$ $\mathcal{M}(Z_n) = n$, $d = 2$ $\mathcal{M}(Z_n) \geq n + 2^d - 2d - 2$, $d > 3$
[113]	GIVEN n-NODE GRAPH G OF VERTEX DEGREE AT MOST THREE	
	ASYNCHRONOUS WITH GLOBAL COMMUNICATION MODEL	
	If every vertex is of degree > 1: $\mathcal{A}(G) = 2$, else: $\mathcal{A}(G) = 1$.	$\mathcal{M}(G) \leq 2\|E\|$ For binary tree T: $\mathcal{M}(T) = 2(n-1) - diam(T)$
[113]	GIVEN n-NODE TREE T	
	ASYNCHRONOUS WITH GLOBAL COMMUNICATION MODEL	
	Construction of a lower bound for $\mathcal{A}(T)$ and an algorithm, which achieves it.	
[113]	GIVEN n-NODE COMPLETE k-ARY TREE T OF HEIGHT h	
	ASYNCHRONOUS WITH GLOBAL COMMUNICATION MODEL	
	$\mathcal{A}(T) \leq h + 1$, $k \geq 4$ $\mathcal{A}(T) \leq h$, $k = 3$	

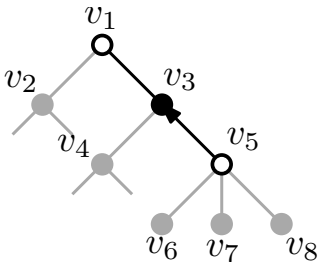
TABLE 2.6: Summary of the results for the monotone contiguous node-decontamination problem with the local-majority property (an unoccupied, cleared node becomes recontaminated if more than a half of its neighbors are infected); for a graph G: $\mathcal{A}(G)$ and $\mathcal{M}(G)$ denote the minimal number of searchers and total moves respectively.

m-IMMUNITY		
Ref.	Number of Searchers	Total Moves Completion Time
[74]	GIVEN d-DIMENSIONAL HYPERCUBE H OF A SIZE $n = n_1 \times \ldots \times n_d$ SYNCHRONOUS MODEL	
	$\mathcal{A}(H,m) \leq 2^{d-m}$	
[74]	GIVEN d-DIMENSIONAL MESH M OF A SIZE $n = n_1 \times \cdots \times n_d$, WHERE $2 \leq n_1 \leq \cdots \leq n_d$ QUASI-SYNCHRONOUS MODEL	
[73]	$\mathcal{A}(M,m) = 1,\ m \geq d$ $\mathcal{A}(M,m) \leq n_1 \cdot \ldots \cdot n_{d-m},\ 1 \leq m < d$ $\mathcal{A}(M,m) \geq n_1 + \ldots + n_{d-m} +$ $-(d-m-1),\ 1 \leq m < d$	$\mathcal{M}(M,m) = \Theta(n)$
[73]	GIVEN d-DIMENSIONAL TOROIDAL MESH Z OF A SIZE $n = n_1 \times \cdots \times n_d$, WHERE $2 \leq n_1 \leq \cdots \leq n_d$ QUASI-SYNCHRONOUS MODEL	
[112]	$\mathcal{A}(Z,m) \leq 2^m n_1 \cdot \ldots \cdot n_{d-m},\ 1 \leq m < d$ $\mathcal{A}(Z,m) \leq 2^{2d-m},\ d \leq m < 2d-3$ $\mathcal{A}(Z,m) = 2^{2d-m},\ 2d-3 \leq m \leq 2d$ $\mathcal{A}(Z,m) \geq 2(n_1 + \ldots n_{d-m} - (d-m-1)),$ $1 \leq m < d$	$\mathcal{M}(Z,m) = \Theta(n)$
[74]	GIVEN n-NODE TREE T ASYNCHRONOUS MODEL	
[73]	Recursively constructed lower bounds for $\mathcal{A}(T,m),\ \mathcal{M}(T,m)$ and an algorithm, which achieves them.	$\mathcal{T}(T,m) \leq 2n-3$

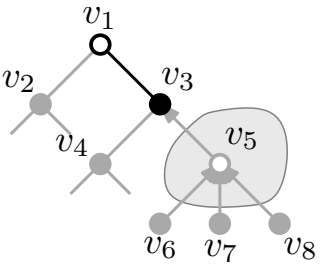
TABLE 2.7: Summary of the results for the monotone contiguous node-decontamination problem with the m-Immunity property (an unoccupied, cleared node becomes recontaminated if m or more of its neighbors are infected) in the local communication model (face-to-face or whiteboard); for a graph G and immunity number $m \geq 0$, $\mathcal{A}(G,m)$, $\mathcal{M}(G,m)$ and $\mathcal{T}(G,m)$ denote the minimal number of searchers, total moves and time steps respectively.



(A) Searcher is placed on the homebase v_1 and slides along to v_3 and v_5.



(B) In the next move searcher comes back to the node v_3 leaving v_5 unguarded.



(C) Vertex v_5 becomes recontaminated, because its three neighbors are contaminated.

FIGURE 2.8: An example of recontamination under the threshold 3-Immunity; one may notice that it is impossible for the node v_1 to be recontaminated.

for trees can be found in [20], where Blin *et al.* show that classical search numbers and the exclusive one can differ exponentially. Relation between exclusive graph searching and pathwidth in edge and mixed search models for planar graphs with bounded maximum degree, split graphs, star-like graphs and cographs is studied in [116].

2.1.5 Computing Search Numbers

As we have already mentioned, computing the node search number of a given graph G is equivalent to computing the pathwidth of G. Moreover, a path decomposition can be easily translated into the corresponding node search strategy that cleans G and vice versa. Thus, the notions of treewidth and pathwidth received growing interest and a vast amount of results has been obtained. Several modifications to pathwidth have been proposed and in this subsection we are interested also in the connected variant, i.e., *connected pathwidth*. Computing this parameter is equivalent to finding the connected node search number.

A lot of research has been done in the direction of obtaining FPT algorithms for pathwidth, parametrized by the pathwidth k. One of the first polynomial-time algorithms was presented in 1983 by Ellis *et al.* and had running time $O(n^{2k^2+4k+2})$ [62]. Later these results were improved (e.g., [23, 24]) leading to the currently fastest FPT algorithm, working in time $2^{O(k^2)}n$ [85]. In these algorithms, in order to produce an optimal solution, an approximate path- or tree-decomposition is pre-computed. It is thus of interest to have good approximations for these problems. Numerous works have been published in this direction [4, 108, 136], leading to the currently fastest algorithm for constant-factor approximation for treewidth, working in time $2^{O(k)}n$ [25], that is, single exponential in the treewidth k. The best known approximation ratio of a polynomial time approximation algorithm for pathwidth is $O(\sqrt{\log(opt)}\log n)$ [64].

There exist exact algorithms for computing pathwidth, whose running times are exponential in the order of the input graph. Pathwidth can be computed in $O^*(2^n)$-time (in $O^*(2^n)$ space) or in $O^*(4^n)$-time with the use of polynomial space, using a simple algorithm from [26]. There is also a faster algorithm with running time $O^*(1.9657^n)$ [144], which has been further improved very recently to $O^*(1.89^n)$ [100]. See [15, 37, 38] for some experimental approaches to pathwidth computation.

For pathwidth, it is known due to [138] that the set of minimal forbidden minors (i.e., the obstruction set) is finite for each fixed k. However, a significant difference between pathwidth and connected pathwidth is that

the latter one is not closed under taking minors and hence it is not known if the set of minimal forbidden minors for connected pathwidth is finite [12].

As much as is done in computing pathwidth, not much has been known about its connected version. During the GRASTA 2017 workshop, Fedor V. Fomin [75] raised an open question, whether we can verify in polynomial time, if the connected pathwidth of a given graph is at most k, for a fixed constant k. We answer this question in the affirmative in Chapter 4. The question whether connected pathwidth is FPT with respect to this parameter remains open.

2.2 Exploration

The goal is to find an algorithm that for any graph G computes a strategy $\mathcal{S}$ which allows agents to *vertex-explore* or *edge-explore* G (all vertices or edges, respectively, have to be visited at least once[5]). We call the exploration *with return* if agents after exploration have to return to their initial positions. We say that an agent *terminates* when it finishes its part of the exploration. If it is not said otherwise graphs are non-weighted and they have only one homebase, in which agents have to terminate. In this subsection we are interested mostly in the on-line setting, although providing results in the off-line one is necessary in order to give a proper perspective.

2.2.1 Completion Time

Completion time is the number of time units required to complete the exploration. We distinguish two models: in the first one (*non-returnable*) an agent once enters an edge has to traverse it fully and in the second one (*returnable*) an agent can stop and/or reverse while traversing an edge. Recall from the introduction to this section, that for edge-weighted graphs a walk along an edge e takes $w(e)$ time units (where $w(e)$ is the weight of the edge e) and for non-weighted graphs takes one time unit. The problem in its two versions is stated as follows:

Time Edge(Vertex)-Exploration

> Find an algorithm that for any number of agents k and a homebase h, computes a time-optimal edge(vertex)-exploration strategy for every *a priori* unknown graph G.

Firstly, let us notice that the edge-exploration can never perform better than the vertex one. Thus, because every graph can be searched by DFS

[5]We omit the prefix when these types of exploration are equivalent, e.g., for trees.

algorithm with a use of one agent, we obtain a trivial upper bound of $2m$ for any exploration strategy. Let D be the h-radius of a graph, understood as a distance from the homebase to the furthest vertex. In order to explore a whole graph and return to the homebase a team of k agents needs at least $\frac{m}{k}$ time units for arbitrary graphs and $2\frac{m}{k}$ for trees. Moreover, at least one of the agents has to explore the furthest vertex, which gives lower bounds of $\max\{\frac{m}{k}, 2D\}$ for arbitrary graphs and $\max\{2\frac{m}{k}, 2D\}$ for trees, which asymptotically is equivalent to $O(D + m/k)$ and $O(D + n/k)$ [31, 83].

Trees. The best known lower bound of the competitive ratio for trees, which holds even in the global communication model, has been given by Dynia *et al.* in [59] (where they introduce so called *Jellyfish* trees) and is equal to $\Omega(\log k / \log\log k)$.

In 2006 Fraigniaud *et al.* in [83] have presented a DFS-based algorithm in the whiteboard communication model, which explores every tree in $O(D + n/\log k)$ time units. This leads to the competitive ratio of $O(k/\log k)$ (comparing to a lower bound of $\Theta(D + n/k)$). These results have been improved by Brass *et al.* in 2011 [31], where an algorithm in the local communication model of the exploration time of $2\frac{m}{k} + O((k + D)^{k-1})$ for general k and $2\frac{m}{k} + 2D^{k-1}$ for $k < D$ has been presented. Moreover, they prove their algorithm to be optimal for $k = 2$. Concurrently, in 2006 Dynia *et al.* [60] have investigated *sparse* trees in the face-to-face communication model and presented an algorithm, which complexity is independent from the number of used agents k. The competitive ratio of $O(D^{1-1/d} \min\{d, \log d \cdot D^{1/2d}\})$ has been achieved, where d is the density parameter defined in the article. When agents know d in advance, then the competitive ratio is equal to $O(D^{1-1/d})$. In 2014 Ortolf and Schidelhauer [126] have constructed two algorithms, where each of them improves the results from [83], but only for a specified relationship between D and k. The competitive ratio of $4D$ for $D = o(k/\log k)$ and of $2^{(2+o(1))\sqrt{(\log D)(\log\log k)}}(\log k)(\log k + \log n)$ for $D \neq o(k/\log k)$ has been established. As for edge-weighted trees, Higashikawa *et al.* in 2014 [90] have given a greedy modification of the algorithm from [83] obtaining the same a lower constant factor and prove the optimality of the competitive ratio of $\Theta(D + n/\log k)$ in the class of greedy algorithms. The algorithm operates in the local communication and returnable model. See Table 2.8 for the summary.

Ref.	Model	Completion Time: Results	Completion Time: Competitive Ratio
[83]	WhB	$O(D + n/\log k)$	$O(k/\log k)$
[31]	Loc	$2m/k + O((k+D)^{k-1})$ $\geq \max\{2m/k,\ 2D\}$	
		TREE FOR TWO AGENTS, $k = 2$	
	Loc	$m + D$	1.5^*
[90]		EDGE-WEIGHTED TREE	
	LocRet		$O(k/\log k)$
		GREEDY ALGORITHM	
	LocRet		$\Theta(k/\log k)^{**}$
[59]		SMALL TEAM SIZE, $k < \sqrt{n}$	
	Glob		$\Omega(k/\log\log k)$
[60]	F2F		$O(H^{1-1/d}\min\{d, \log dH^{1/2d}\})$
		KNOWN DENSITY D	
	F2F		$O(D^{1-1/d})$
[126]	Loc		$\geq 4D$
		IF $D > o(k/logk)$	
	Loc		$\geq 2^{(2+o(1))\sqrt{(\log D)(\log\log k)}} \cdot$ $\cdot(\log k)(\log k + \log n)$

TABLE 2.8: List of the results for the on-line exploration models for trees with the completion time optimization factor, where * marks the optimal ones. We denote D as the diameter (understood as the maximum distance from the root); n as the number of vertices; m as the number of edges and k as the number of agents. In the *Model* description: *Glob* - global communication; *F2F* - face to face communication; *WhB* - whiteboard communication; *Loc* - local communication; *Bo* - bounded communication; *NoC* - none communication; *Ret* - returnable; *NoR* - non-returnable; *Ed* - edge-exploration and *Ve* - vertex-exploration.

General graphs. The best known lower bound of competitive ratio of edge-exploration problem for general graphs is equal to $2 - 1/k$ and holds even in the global communication model [83]. In 2014 Brass *et al.* [30] have generalized the algorithm from [31, 83] for all graphs and proved the competitive ratio of $\left(2 + \frac{2nk(\ln k+1)}{m}\right)$, which is near the optimal $(2 - 1/k)$ for dense graphs (i.e., where m is a super linear function of n).

The results for trees and general graphs presented in the two previous paragraphs are efficient only for small size teams, where $k < \sqrt{n}$, i.e., where in a lower bound of $\Omega(D + n/k)$ the latter component n/k is dominant. Dereniowski *et al.* in 2015 [53] have investigated the vertex-exploration of arbitrary graphs performed by teams of the polynomial size, i.e., $k = Dn^c$, where $c > 1$ is any constant. They provided lower bounds of $D(1 + 2/c - o(1))$ for the face-to-face model and of $D(1 + 1/c - o(1))$ for the global communication model. They have also constructed asymptotically optimal algorithms for both communication models, which explore every graph in $O(D)$ time units. One year later Disser *at al.* [55] have filled the remaining gap for $\sqrt{n} < k < Dn^c$ by extending a lower bound of $\Omega(\log k / \log\log k)$ to the range $\sqrt{n} \leq k < n \log^{c-1} n$ and proving the competitive ratio to be $\omega(1)$ otherwise.

Grids. Ortlof and Schidelhauer examined the vertex-exploration of $p \times p$ grids with rectangular obstacles [127] and have provided a $O(\log^2 p)$-competitive algorithm in the face-to-face model (comparing to a trivial lower bound of $\max\{2p - 1, n/k\}$). They have also presented a lower bound of the competitive ratio (independent of p) of $\Omega(\log k / \log\log k)$ (by using similar methods to [59]), which holds even in the global communication model.

Rings. Higashikawa *et al.* in 2014 [90] have constructed a 1.5-competitive algorithm performed by 2 agents in the global communication non-returnable model for edge-weighted rings, which is optimal.

See Table 2.9 for the summary.

2.2.2 Other Optimization Factors

Apart from minimizing the completion time of a strategy few other optimization factors occurs in the literature, which we describe in this subsection.

		Completion Time	
Ref.	Model	Results	Competitive Ratio
[83]		ANY GRAPH	
	EdNoC		$\Omega(k)$
	EdGlob		$\geq 2 - 1/k$
[31]		ANY GRAPH	
	EdLoc		$\geq \max\{m/k,\ 2D\}$
[90]		EDGE-WEIGHTED RING	
	GlobNoR		1.5^*
[127]		GRID $l \times l$ WITH RECTANGULAR OBSTACLES	
	VeF2F	$O(l \log^2 l + (f \log l)/k))$ $\geq \max\{2l - 1, f/k\}$	$O(\log^2 l)$
	VeGlob		$\Omega(\log k/(\log\log k))$
[30]		ANY GRAPH	
	EdLoc	$\leq \frac{2}{k}(m + nk(\ln k + 1))$	$\leq \left(2 + \frac{2nk(\ln k+1)}{m}\right)$
[53]		ANY GRAPH, $k = Dn^c$	
	VeF2F	$\leq D(1 + 2/(c-1) + o(1))$ $\geq D(1 + 2/(c) - o(1))$	$O(1)$
	VeGlob	$\leq D(1 + 1/(c-1) + o(1))$ $\geq D(1 + 1/(c) - o(1))$	$O(1)$
[55]		ANY GRAPH, $\sqrt{n} < k < n \log^c n$	
	VeGlob		$\Omega(\log k/\log\log k)$
		ANY GRAPH, $n \log^c n \leq k < Dn^c$	
	VeGlob		$\omega(1)$

TABLE 2.9: Continuation of the Table 2.8 with results for different graph topologies; f stays for the number of non-obstacles vertices and $c > 0$ is any constant.

Energy. The energy parameter is defined as the maximum traversed distance by agents in a strategy. The problem for trees is stated as following:

Energy Exploration

> Find an algorithm that for any number of agents k and a homebase h, computes a mimimum-energy exploration strategy for every *a priori* unknown tree T.

In 2006 Dynia *et al.* have presented an 8-competitive algorithm for exploring trees in the bounded communication model and proved a lower bound to be at least 1.5 [58]. One year later in [59] this results have been improved by an algorithm with the competitive ratio of $(4 - 2/k)$ (which holds even in the face-to-face setting). In the strategies computed by both algorithms, in order to minimize the energy, each step consists of only one move. Thus, both algorithms have the high time complexity of $O(kD + n)$, where D is the height of a tree.

Number of Agents. One of the natural assumptions is to restrict the energy that can be consumed by each of the agents during the exploration. Indeed, let us assume that agents are equipped with a battery of a given size B. Firstly this problem occurred in [5] and [13] by the name of the *piecemeal* exploration, where a single agent has to explore the whole graph on its own and can recharge its battery in the homebase node. Notice, that only graphs of the maximum diameter $B/2$ can be searched by such an agent. As for the teams of agents for exploration with return, Das *et al.* in [42] present us two algorithms for trees in global and face-to-face communication settings with the competitive ratio of $O(\log B)$ and prove this results to be optimal. For the exploration, where after finishing agents do not have to return to the homebase, the asymptotically optimal algorithm based on a DFS approach has been given in [43].

Recently, a slightly different approach has been presented: authors of [6] have considered the problem of exploring an unknown tree with a team of k agents with limited energy B. The goal is to maximize the number of nodes collectively visited by all agents during the execution of the strategy. They present a 3-competitive algorithm and a non-trivial lower bound of 2.17 on the competitive ratio of any on-line algorithm.

Total Distance. In the *broadcasting* problem, apart from the homebase vertex, there exist also a *source* one. The task for agents is to deliver the information from the source to every vertex of a graph. Exploration can be seen as a special case of this problem, where a source is located in the homebase.

In [39] edge-weighted trees in the off-line setting have been studied, where a group of k mobile agents has a goal to broadcast the information along the whole tree minimizing the total distance. Two algorithms have been presented: for an arbitrary k of completion time $O(n \log n)$ and a linear one for k big enough (i.e., at least the number of leaves).

The Cost. In [129] a new optimization factor was proposed called the *cost*, which is the total distance traversed by agents coupled with the cost of invoking them (see Chapter 5). Vertex-exploration is analyzed for two edge-weighted graph classes: rings and trees, in the off-line and on-line settings. Agents do not have to terminate in the homebase and can communicate globally. Algorithms that compute the optimal strategies for a given ring and tree of order n are presented. For rings in the on-line setting, author gives a 2-competitive algorithm and proves the lower bound of $3/2$ of the competitive ratio for any on-line algorithm. For every algorithm for trees in the on-line setting, the competitive ratio is proved to be no less than 2, which can be achieved by the DFS algorithm.

See Table 2.10 for the summary.

Ref.	Model	Competitive Ratio	Completion Time
		MINIMAL ENERGY, UNKNOWN TREE	
[58]	BoR	≤ 8	$O(kD+n)$
	BoR	≥ 1.5	
[59]	F2FR	$\leq (4-2/k)$	
		MINIMAL NUMBER OF AGENTS, UNKNOWN TREE	
[42]	F2FR	$\Theta(\log B)$	
[43]	Glob	$O(1)$	
		MINIMAL TOTAL DISTANCE, GIVEN TREE AND k AGENTS	
[39]			$O(n \log n)$
		k big enough	$O(n)$
		MINIMAL COST	
		GIVEN TREE AND RING	
[129]			$O(n)$
		UNKNOWN RING	
	VeGlob	≤ 2	$O(n)$
		$\geq 3/2$	
		UNKNOWN TREE	
	Glob	2	$O(n)$

TABLE 2.10: List of the results for the exploration models with different optimization factors, where R stays for the exploration with return.

Part I

Decontamination

Chapter 3

On-line Search in Two-Dimensional Environment

In this chapter we study the decontamination problem on graphs modeled by partial grids (i.e., connected subgraphs of grids, formally defined later), and the agents operate in a distributed on-line model.

This chapter is constructed as follows: the next section defines the graph searching problem we study, while Section 3.2 introduces the terminology related to the partial grid networks we consider in this chapter. Section 3.3 gives a construction of a class of n-node networks such that each on-line algorithm, which constructs a monotone connected strategy, uses $\Omega(\sqrt{n})$ searchers for some network in the class which turns out to be $\Omega(\sqrt{n}/\log n)$ times more than an optimal off-line algorithm would use.

Section 3.4 describes a distributed algorithm that performs a monotone connected search in partial grids where it is assumed that the algorithm is given an upper bound n on the size of the network. We assume a 'sense of direction' in our model, that is, the grid is embedded into a two-dimensional space by assigning integer coordinates to the nodes. Then, a searcher knows the coordinates of each neighbor of the currently occupied node. More details are given in Section 3.2. We point out that this algorithm uses a distributed procedure from [27] as a subroutine that is called many times to clear selected parts of a grid and it can be seen as a generalization from a 'linear' graph structure studied in [27] to a 2-dimensional structure discussed in this work. Also, although both algorithms are conducted via some greedy rules which dictate how a search should 'expand' to unknown parts of the graph, the analysis of our algorithm is different from the one in [27].

Then, in Section 3.5 we prove the correctness of the algorithm and provide an upper bound on its performance: it is using $O(\sqrt{n})$ searchers for

any partial grid network. In Section 3.6 we consider a modified version of the algorithm, which receives no information on the underlying graph in advance, and we prove that the algorithm also uses $O(\sqrt{n})$ searchers. This result, stated in Theorem 3.6.1, is our main contribution. We finish with conclusions in Section 3.7, giving a few remarks on how our work relates to searching two-dimensional environments, like polygons with holes. As there are many open problems and research directions related to the subject, we list some of them also in Section 3.7.

Applications in robotics. We remark on a potential practical motivation of our setting. Partial grids, which can be seen as a grids with obstacles, are a way of modeling two-dimensional shapes, e.g., polygons. Every search strategy for a polygon can be used to obtain a search strategy for its underlying partial grid and vice versa. The number of searchers in both cases are withing a constant factor of each other. Thus in particular, searching strategies for continuous scenarios like polygons can be obtained by first getting the underlying partial grid and then computing a (discrete) search strategy for the grid by the algorithm we propose in this work. For more details see Section 3.7.1. Thus, our results may be of particular interest not only by providing theoretical insight into searching dynamics in distributed searcher computations, but may also find applications in the field of robotics. Most investigations oriented towards algorithms that can be applied on physical devices need to deal with the problem of modeling of the real world. This can be done either by discretizing it (usually through graphs) or by building algorithms that work in continuous search space and need to address the geometric issues that emerge. Having in mind the vast literature on the subject we point the interested reader to a few references to recent works in this field [36, 57, 91, 101, 135, 139, 140, 141, 143].

3.1 The Model

Let G be a simple, undirected, connected partial-grid (i.e., a subgraph of a grid) with a single homebase h. A *monotone connected k-search strategy* $\mathcal{S}$ for a network G is defined as follows. Initially, k searchers are placed on h of G. Then, $\mathcal{S}$ is a sequence of moves, where each move consists of selecting one searcher present at some node u and sliding the searcher along an edge $\{u, v\}$. (Thus, the searcher moves from its current location to one of the neighbors.) We require $\mathcal{S}$ to decontaminate G, be monotone and *connected*,

i.e., the *clear* subgraph, that is, the subgraph consisting of all clear edges, is connected after each move of the search strategy.

We now state the on-line distributed model we use. All searchers start at the homebase and the network itself is not known in advance to the searchers (except for the fact that the searchers may expect that the network is a partial grid). In fact, the searchers have no information about the network. (We note here that our main algorithmic result will be obtained in two stages: first we describe an algorithm that as an input receives the upper bound n of the size of the network and then we use it to obtain our main result, an algorithm that works without any a priori information about the network.) We assume that nodes are anonymous and searchers have identifiers. The edges incident to each node are marked with unique labels (port numbers) and because only partial grids are considered in this work we assume that labels naturally reflect all possible directions for each edge (i.e., left, right, up and down).

For the searchers, we assume that they communicate locally by exchanging information when present at the same node. Our algorithm is stated as if there existed global communication but it can be easily turned into required one with local communication as follows: we can designate one extra searcher called the *leader* who will be performing the following actions at the beginning of each move of the search strategy to be executed. First, the leader visits all nodes of the subgraph searched to date and gathers complete information about its structure and positions of all other searchers, then the leader computes the next move and finally visits all searchers to pass the information about the next move. Then, the move is performed by the searchers.[1]

Our algorithm is described for the synchronous model in which time is divided into steps, each step having the same unit length duration allowing each searcher to perform its local computations and slide along an edge if the searcher decides to move. We note that this assumption can be lifted and the algorithm can be easily restated to be asynchronous. Indeed, having one searcher that is the leader one can simulate synchronous behavior of the searchers in such a way that the leader waits for the completion of the current move of another searcher and then informs the searcher that is supposed to perform the next move, dictated by the search strategy, to start the move.

[1]Note that the actions of the leader clearly contain a lot of excess work in terms of the number of moves it performs; since the criteria as time or cost (number of sliding moves) are out to scope of this work, we will leave the reader with such a simple leader implementation.

3.2 Partial Grid Notation

We define a *partial grid* $G = (V, E)$ with a set of n nodes V and edges E as a connected subgraph of an $n \times n$ grid. We consider each partial grid to be embedded into two-dimensional Cartesian coordinate system with a horizontal x-axis and vertical y-axis, where each node of G is located at a point with integer coordinates and two nodes are adjacent if and only if the distance between them equals one (in Euclidean metric). This embedding is considered for two reasons. The first one is technical, as it simplifies some statements when we refer to coordinates when pointing nodes of G. The second is that our on-line algorithm relies on the underlying geometric structure. For convenience, the homebase is located at the point $(0,0)$. In order to refer to a node that corresponds to a point with coordinates (x,y) we write $v(x,y)$. In this chapter n denotes an upper bound on the number of nodes of a partial grid, such that $\sqrt{n}$ is an integer.

Informally speaking, our algorithm will conduct a search by expanding the clear part of the graph from one 'checkpoint' to another. These checkpoints (defined formally later) will be subsets of nodes and their potential placements on the partial grid are dictated by the concept of a *frontier*. Take any $x = i\sqrt{n}$ for some integer i, $y = j\sqrt{n}$ for some integer j and take $i', j' \in \{0,1\}, i' \neq j'$. Then, the line segment with endpoints (x,y) and $(x+\sqrt{n}i', y+\sqrt{n}j')$ is called a *frontier* and denoted by $F\left((x,y),\left(x+\sqrt{n}i', y+\sqrt{n}j'\right)\right)$. Whenever the endpoints of a frontier are clear from the context or not important we will omit them. The frontier $F\left((0,0),\left(\sqrt{n},0\right)\right)$ that contains the origin is called the *homebase frontier* and the set of all frontiers is denoted by $\mathcal{F}$. We will also divide frontiers into *vertical* and *horizontal* ones, where coordinates of two extreme nodes do not differ on first and second coordinate, respectively.

Similarly to the graph induced by a set of nodes, we denote the subgraph induced by all nodes that belong to a frontier F of a partial grid G by $G[F]$.

For $i \in \{1,\ldots,\sqrt{n}\}$ and some frontier $F = F((x,y),(x',y'))$, where $x \leq x'$ and $y \leq y'$, we define the *i-th rectangle of* F, denoted by $\mathcal{R}(F,i)$, as the rectangle with corner vertices $(x-i, y-i)$, $(x-i, y+i)$, $(x'+i, y'-i)$, $(x'+i, y'+i)$ if F is horizontal and as the rectangle with corner vertices $(x-i, y-i)$, $(x+i, y-i)$, $(x'-i, y'+i)$, $(x'+i, y'+i)$ if F is vertical. See Figure 3.1 for an example.

Informally speaking, the two above concepts, namely frontiers and rectangles, provide a template on how the search may progress. However, due to the structure of a partial grid it may be possible that only certain nodes, but not all, that lie on a frontier have been reached at some

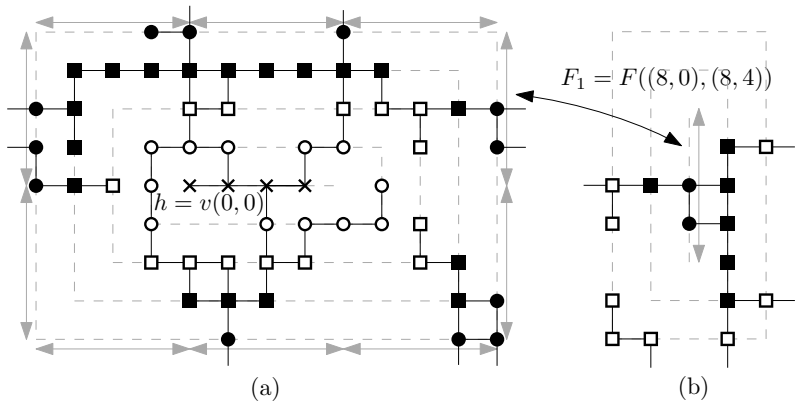


FIGURE 3.1: An illustration of the concept of rectangles (here $\sqrt{n} = 4$). In (a) *crosses* denote nodes that lie on the homebase frontier $F = F((0,0),(4,0))$, *empty circles* denote nodes that lie on $\mathcal{R}(F,1)$, *empty squares* the ones on $\mathcal{R}(F,2)$, *dark squares* the ones on $\mathcal{R}(F,3)$ and *dark dots* denote nodes that lie on $\mathcal{R}(F,4)$. *Gray arrows* stand for the 10 frontiers, that lie on the $\mathcal{R}(F,4)$ (six horizontal and four vertical ones). We denote one of the vertical frontiers that lie on $\mathcal{R}(F,4)$ as $F_1 = F((8,0),(8,4))$. In (b) *dark dots* denote nodes that lie on F_1, *dark squares* the ones on $\mathcal{R}(F_1,1)$ and *empty squares* the ones on $\mathcal{R}(F_1,2)$.

point of a search strategy. For this reason, our notation needs to be extended to subsets of nodes that lie on frontiers and the corresponding rectangles. Any subset C of nodes of G that belong to some frontier F is called a *checkpoint*. *The* 0*-th expansion of a checkpoint* C is C itself and is denoted by $C\langle 0\rangle$. For $i \in \{1,\ldots,\sqrt{n}\}$ we define *the i-th expansion of* C, denoted by $C\langle i\rangle$, recursively as follows: the set $C\langle i\rangle$ consists of all nodes $v \notin C\langle 0\rangle \cup C\langle 1\rangle \cup \cdots \cup C\langle i-1\rangle$ for which there exists a node $u \in C\langle i-1\rangle$, such that there exists a path between v and u in the subgraph of G induced by nodes that lie on the rectangles $\mathcal{R}(F,0), \mathcal{R}(F,1), \ldots, \mathcal{R}(F,i)$. Define

$$C^+\langle i\rangle = C\langle 0\rangle \cup \ldots \cup C\langle i\rangle, \quad i \in \{0,\ldots,\sqrt{n}\}.$$

Informally, $C\langle i\rangle$ consists of only those nodes that belong to the rectangle $\mathcal{R}(F,i)$ that are connected to nodes of C by paths that lie 'inside' of $\mathcal{R}(F,i)$ — this definition captures the behavior of searchers (in our algorithm) that guard the nodes of C and 'expand' from C in all directions: then possible nodes that belong to any of the rectangles $\mathcal{R}(F,0), \mathcal{R}(F,1), \ldots, \mathcal{R}(F,i)$ but do not belong to $C^+\langle i\rangle$ will not be reached by the searchers. See Figure 3.2 for an exemplary checkpoint with its expansions.

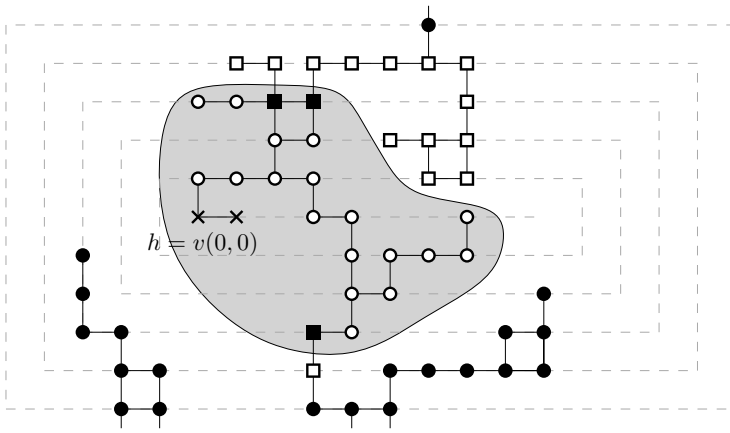


FIGURE 3.2: Some expansions of a checkpoint C (here $\sqrt{n} = 9$); *crosses* denote $C = C\langle 0\rangle$, *thw gray area* covers nodes that belong to $C^+\langle 3\rangle$, *empty squares* denote nodes in $C\langle 4\rangle$ and *dark squares* denote the ones that need to be guarded provided that the gray area consists of the clear nodes. The horizontal dotted line that contains h is the considered frontier.

3.3 Lower Bound

First note that a regular $\sqrt{n} \times \sqrt{n}$ grid requires $\Omega(\sqrt{n})$ searchers even in the off-line setting [61], that is, when the network is known in advance and the searchers may decide on the location of the homebase. Therefore, our on-line algorithm is asymptotically optimal with respect to this worst case measure.

We aim at proving that for *each* on-line algorithm A there exists an n-node partial grid network G with homebase h such that

$$\max_h A(G,h)/A^{opt}(G,h) = \Omega(\sqrt{n}/\log n).^2$$

Define a class of partial grids

$$\mathcal{L} = \bigcup_{l \geq 0} \mathcal{L}_l,$$

[2]We remark that we defined the competitive ratio by taking the worst case homebase for A. However, we note that this does not weaken the result of this section as, informally speaking, one may take two copies of each grid obtained in this section, rotate one copy by 180 degrees and merge the two copies at their homebases. Then, we obtain that for each choice of the homebase any algorithm is forced to use $\Omega(\sqrt{n})$ searchers for some grids since in one copy the search is conducted as in our following analysis.

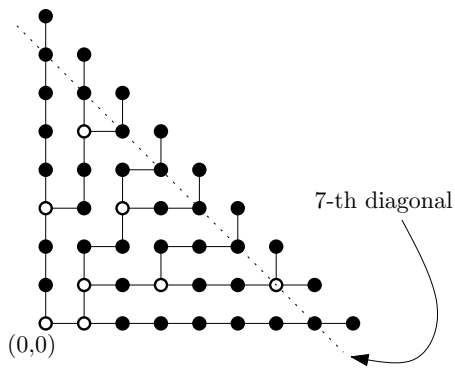


FIGURE 3.3: A network from $\mathcal{L}_8$ obtained from the corresponding network in $\mathcal{L}_7$ by extending it at 6.

where $\mathcal{L}_l$ for $l \geq 0$ is defined recursively as follows. We take $\mathcal{L}_0$ to contain one network that is a single node located at $(0,0)$. Then, in order to describe how $\mathcal{L}_{l+1}$ is obtained from $\mathcal{L}_l$, $l \geq 0$, we introduce an operation of *extending* $G \in \mathcal{L}_l$ *at* i, for $i \in \{0,\ldots,l\}$. In this operation, first take G and add $l+2$ new nodes located at coordinates:

$$(0,l+1),(1,l),\ldots,(j,l+1-j),\ldots,(l+1,0).$$

Call these coordinates the $(l+1)$*-th diagonal*. For each $j \in \{0,\ldots,i\}$ add an edge connecting the nodes $v\,(j,l-j)$ and $v\,(j,l-j+1)$, and for each $j \in \{i,\ldots,l\}$ add an edge connecting the nodes $v\,(j,l-j)$ and $v\,(j+1,l-j)$. Then, obtain $\mathcal{L}_{l+1}$ as follows: initially take $\mathcal{L}_{l+1}$ to be empty and then for each $G \in \mathcal{L}_l$ and for each $i \in \{0,\ldots,l\}$, obtain a network G' by extending G at i and add G' to $\mathcal{L}_{l+1}$. Notice here that a graph constructed this way is not only a partial grid, but also a tree.

Figure 3.3 shows a network that was obtained from the corresponding network in $\mathcal{L}_7$ by extending it at 6.

For a network $G \in \mathcal{L}_l$, $l \geq 0$, we define a *characteristic sequence of* G, $\sigma(G)$, as follows. If $l = 0$, then the characteristic sequence of G is empty. If $l > 0$, then take the network G' such that G has been obtained by extending G' at i. The characteristic sequence of G is $\sigma(G)$, constructed by appending to $\sigma(G')$ a new element $v\,(i,l-i-1)$. Note that the characteristic sequence uniquely defines the corresponding network. In other words, G is a binary tree rooted at $v(0,0)$ with $l+1$ leaves, where only the vertices from $\sigma(G)$ have two children. The network introduced in Figure 3.3 has characteristic sequence $(v\,(0,0), v\,(1,0), v\,(1,1), v\,(0,3), v\,(3,1), v\,(2,3), v\,(1,5), v\,(6,1))$.

Lemma 3.3.1. *For any integer l and for each on-line algorithm A computing a connected monotone search strategy there exists $G \in \mathcal{L}_l$ such that for homebase $v(0,0)$ we have $A(G, v(0,0)) \geq (l+1)/2$.*

Proof. Consider any algorithm A producing a connected monotone search strategy. Run A for each network in $\mathcal{L}_l$ with the homebase $v(0,0)$. Note that for each network in $\mathcal{L}_l$, there exist distinct moves $m_1, \ldots, m_l$ such that till the beginning of move m_j, $j \in \{1, \ldots, l\}$, no node on the j-th diagonal has been occupied by a searcher and at the end of m_j some node $v(x_j, y_j)$ of the j-th diagonal is occupied by a searcher. Consider $G \in \mathcal{L}_l$ such that $\sigma(G) = (v(0,0), v(x_1, y_1), \ldots, v(x_{l-1}, y_{l-1}))$. Informally speaking, whenever the algorithm reaches for the first time a node $v(i, j-i)$ in the j-th diagonal, an *adversary* decides to extend at i the network explored so far, thus always forcing the situation that the first node reached on a diagonal is of degree three.

Note that at the beginning of move m_j, $j \in \{1, \ldots, l\}$, no node of the j-th diagonal has been reached by a searcher and the first j nodes of the characteristic sequence have been reached by searchers. Recall that G is a binary tree.

We analyze the explored part of any graph $G \in \mathcal{L}_l$ at the beginning of the move m_l. All edges incident to the leaves in G are contaminated at this point. On the other hand, all nodes of the characteristic sequence have been visited by searchers till the end of the move $m_l - 1$. Therefore, the contaminated subgraph of G at this point is a collection of paths leading from nodes that are guarded to the leaves. Since there are $l+1$ leaves in G, there are $l+1$ such paths, each such a path needs to have a searcher placed at one of its endpoints (the one that is not a leaf in G) and, by construction of G, any searcher can be present on at most two such endpoints. Thus, at least $(l+1)/2$ nodes need to be occupied by searchers, as required by the lemma. □

Theorem 3.3.1. *For each on-line algorithm A computing a connected monotone search strategy there exists an n-node network G with homebase h such that*

$$\frac{A(G,h)}{A^{opt}(G,h)} = \Omega(\sqrt{n}/\log n).$$

Proof. Observe that each network G in $\mathcal{L}$ is a tree and therefore $A^{opt}(G,h) = O(\log(n))$, $n = |V(G)|$ [9, 118]. The theorem follows hence from Lemma 3.3.1 and the fact that the length of the characteristic sequence of each network in $\mathcal{L}_l$ is $\Omega(\sqrt{n})$. □

3.4 The Algorithm

In this section we describe our algorithm that takes an upper bound on the size of the network as an input. Section 3.4.1 deals with the initialization performed at the beginning of the algorithm. Then, Section 3.4.2 introduces two procedures used by the algorithm and finally Section 3.4.3 states the main algorithm.

We point out that the strategy to be computed is monotone. This means that whenever a new node has been reached by some searcher, the node will be guarded as long as it has some incident contaminated edges. After each move performed by searchers, each searcher that occupies a node that does not need to be guarded is said to be *free*. Each node that needs to be guarded is occupied by at least one searcher; if more searchers occupy such a node then all of them except for one are also *free*. Once all incident edges of a guarded node v become clear, the searcher that has been guarding v becomes immediately free. So we do not express this fact explicitly in the algorithm as the above rule is sufficient to partition the searchers into the free and guarding ones at any point of the strategy computed by the algorithm. Before we start the description of the algorithm, we stress out how we 'reuse' searchers that are free. Whenever the algorithm decides that a searcher needs to perform some action the following decision takes place. If there exists a searcher that is free, then the actions is made by this searcher. If there is no free searcher, then a new one is introduced by the algorithm to perform the action. Thus, in our analysis we will count the number of searchers introduced throughout the execution of the algorithm.

If, at some point, no node of the last expansion of some checkpoint needs to be guarded, then we say that the expansion is *empty*.

3.4.1 Initialization

We start presenting our algorithm by describing initial conditions. Recall that the origin $v\,(0,0)$ of the two-dimensional xy coordinate system is situated in the homebase. The initial checkpoint C_0 is the set of nodes of the connected component of $G[F]$ that contains h, where F is the homebase frontier. Thus, initially $|C_0|$ searchers place themselves on all nodes of C_0 (note that the nodes of C_0 induce a path in G). See Figure 3.4 for an example.

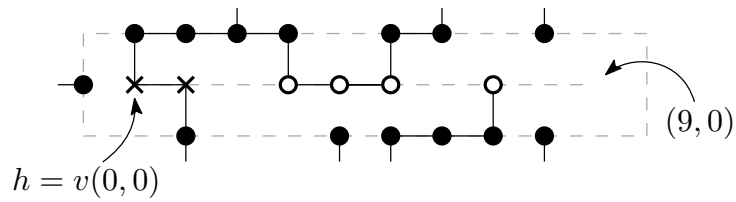


FIGURE 3.4: Exemplary initialization for $\sqrt{n} = 9$; *crosses* denote nodes belonging to the initial checkpoint C_0 and *empty circles* denote nodes that belong to the homebase frontier, but do not fall into C_0.

3.4.2 Procedures

Procedure `ClearExpansion`

We start with an informal description of the procedure. When a new checkpoint C has been reached, our search strategy 'expands' from C by successively clearing subgraphs $G[C^+\langle i\rangle]$ for $i \in \{1, \ldots, \sqrt{n}\}$. Once all nodes in $C^+\langle i-1\rangle$ are clear for some $0 < i \leq \sqrt{n}$, the transition to reaching the state in which all nodes in $C^+\langle i\rangle$ are clear requires clearing all nodes of the i-th expansion of C. This is done by calling for every guarded node u from $C^+\langle i-1\rangle$ a special procedure (`ModConnectedSearching`, described below), which clears nodes which belong to $C\langle i\rangle$ and 'can be accessed' from u. Procedure `ClearExpansion` makes the above-mentioned calls to `ModConnectedSearching` and uses $O(\sqrt{n})$ searchers in the process.

For clearing all nodes of the i-th expansion of C, provided that $G[C^+\langle i-1\rangle]$ is clear we will use a procedure from [27]. That procedure is more general and it is stated in [27] as Procedure `ConnectedSearching` with its performance stated in Theorem 1 in [27]. Here we give its following reformulation that uses our notation.

Theorem 3.4.1 ([27])**.** *Let F be any frontier and let G' be any connected partial grid with nodes lie entirely on the rectangles $\mathcal{R}(F,0), \mathcal{R}(F,1), \ldots, \mathcal{R}(F,i)$, $i \geq 0$. There exists an on-line procedure* `ConnectedSearching` *that, starting at an arbitrarily chosen homebase in G', clears G' in a connected and monotone way using $6i + 4$ searchers.*

We stress out that the above theorem assumes that the partial grid is entirely contained in the area covered by the rectangles. In other words, the subgraph G' in Theorem 3.4.1 has no vertices 'outside' of the specified area. However, while using procedure `ConnectedSearching`, we will be clearing a subgraph of $G[C^+\langle i\rangle]$ that is embedded into the entire partial grid and thus some nodes v of $G[C^+\langle i\rangle]$ have edges leading to neighbors

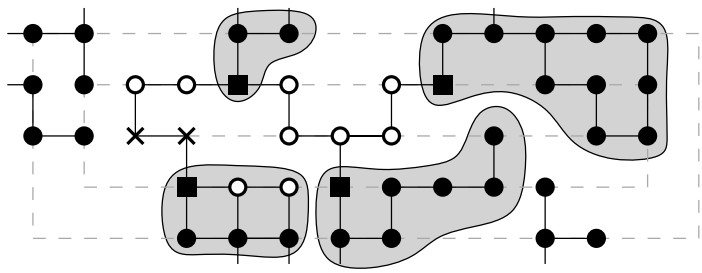


FIGURE 3.5: Example of an execution of procedure `ClearExpansion`; *crosses* denote $C = C\langle 0\rangle$, *empty circles* denote nodes that belong to $C^+\langle 1\rangle$, *dark squares* denote the one that belongs to $C^+\langle 1\rangle$ and for which procedure `ModConnectedSearching` is invoked, *gray areas* show nodes that will be cleared in four calls of `ModConnectedSearching` in order to clear $C\langle 2\rangle$. Note that the empty circles that lie on a gray area are guarded at first, but after one of the calls of `ModConnectedSearching` there is no need to guard them any more, so the procedure is not invoked for them.

that lie outside of $G[C^+\langle i\rangle]$. If such an edge is already clear, then no recontamination happens for the node v and moreover no searcher used by `ConnectedSearching` for the subgraph of $G[C^+\langle i\rangle]$ needs to stay at v. On the other hand, if such an edge is contaminated (and thus not reached yet by our search strategy), then v needs to be guarded and for that end we place an extra searcher on it that guards v during the remaining execution of `ConnectedSearching`. Note that in the latter case, the node v belongs to $\mathcal{R}(F,i)$, where F is the frontier that contains the nodes of C and therefore there exist $O(\sqrt{n})$ such nodes v. In other words, `ConnectedSearching` is called to clear a certain subgraph contained within $\mathcal{R}(F,i)$ and whenever a node on the rectangle $\mathcal{R}(F,i)$ has a contaminated edge leading outside of the rectangle $\mathcal{R}(F,i)$, then an extra searcher, not accommodated by `ConnectedSearching` in Theorem 3.4.1, is introduced to be left behind to guard v. The modification of `ConnectedSearching` that leaves behind a searcher on each such a newly reached node of $\mathcal{R}(F,i)$ will be denoted by `ModConnectedSearching`. Note that this procedure is invoked for every guarded node from $C^+\langle i-1\rangle$ in order to clear $C^+\langle i\rangle$, see Figure 3.5 for an example.

It is enough to provide as an input to `ModConnectedSearching`: a node v in $C^+\langle i-1\rangle$ that plays the role of homebase for `ModConnectedSearching`, the frontier F and i. We stress out that there are possibly many such nodes v and once one of them is selected, some other such nodes in $C^+\langle i-1\rangle$ may no longer have an incident edge that is contaminated since the call

to ModConnectedSearching did clear such an edge. However, we assume that ModConnectedSearching clears only the maximal connected subgraph that contains v and is induced by contaminated edges only. Thus, once its execution is completed, there may exist another vertex v for which a new call to ModConnectedSearching will be made to clear another maximal connected subgraph induced by contaminated edges. See Figure 3.5 that illustrates this process: the shaded areas indicate which subgraphs have been actually cleared by subsequent calls to ModConnectedSearching. We point out that, alternatively, a single call to ModConnectedSearching would suffice if the procedure would 'process' the entire subgraph contained in the expansion $C^+\langle i\rangle$ but this approach would ignore that some subgraph of $C^+\langle i\rangle$ is already clear and hence we present the procedure as having multiple calls to ModConnectedSearching that work on contaminated edges only. We note that each checkpoint used in our final algorithm is obtained as follows: some frontier F is selected and then a checkpoint C is created as some set of nodes that belong to F; thus we assume that with C such a unique frontier F is associated.

Thus, this approach guarantees us using at most $6i+4$ searchers to clear $G[C\langle i\rangle]$ and, in addition to those, $2\sqrt{n}+8i$ searchers for guarding nodes lying on $\mathcal{R}(F,i)$, which will be analyzed in more details in Section 3.5.

To summarize, we give a formal statement of our procedure.

Procedure ClearExpansion

Input: An expansion $C\langle i-1\rangle$ with C contained in the frontier F, $i \geq 1$.
Result: Clearing all nodes of $C\langle i\rangle$.
while there exists a node $v \in C\langle i-1\rangle$ with a contaminated neighbor u in $C\langle i\rangle$ **do**
Place $6i+4+2\sqrt{n}+8i$ free searchers on v to be used by ModConnectedSearching.
Call ModConnectedSearching for v as the homebase, frontier F and integer i.

The following observation summarizes the outcome of an execution of procedure ClearExpansion.

Lemma 3.4.1. *Suppose that $C\langle i-1\rangle$, that is an expansion contained in a frontier F, where $i \geq 1$, is an input to procedure* ClearExpansion. *Suppose that G' is the maximal subgraph contained in $G[C\langle i\rangle]$ and induced by all nodes v such that there exists a path contained in $G[C\langle i\rangle]$ connecting v with a vertex of C. Then, a call to* ClearExpansion *with the above input provides the following:*

– *exactly the edges of G' that are contaminated prior to the call are cleared during this call to* `ClearExpansion`,

– *after the call, each vertex of G' with an incident contaminated edge is guarded by a searcher,*

– *all of the nodes from $C\langle i-1\rangle$ are cleaned and do not have to be guarded.*

We point out that there may be an indirect interaction between different checkpoints. Consider an execution of procedure `ClearExpansion` with an input $C\langle i-1\rangle$. At the point of performing this call, there may exist a different checkpoint C' and a corresponding expansion $C'\langle i'\rangle$ such that some searcher is guarding a node v of $C'\langle i'\rangle$ because v has (assuming for simplicity) a single contaminated edge e incident to it. It may happen that during the execution of `ClearExpansion` the edge e becomes clear as it belongs to $C^+\langle i\rangle$. Therefore, this results in a situation that v is not guarded (since it has no incident contaminated edges) and the corresponding searcher becomes free.

Procedure `UpdateCheckpoints`

By definition, if F is some frontier, then $\mathcal{R}(F, \sqrt{n})$ contains 10 frontiers (see Figure 3.1). Thus, reaching the $\sqrt{n}$-th expansion $C\langle\sqrt{n}\rangle$ of a checkpoint of F provides a possibility of creating one new checkpoint for each of the above frontiers. Procedure `UpdateCheckpoints`, which takes as an input $C\langle\sqrt{n}\rangle$ and a collection $\mathcal{C}$ of currently present checkpoints, generates these new checkpoints and adds them to $\mathcal{C}$ and removes C from $\mathcal{C}$. Also, if it happens that some newly constructed checkpoint belongs to the same frontier as some existing checkpoint in $\mathcal{C}$ and no expansion for the existing one has been performed yet, then both checkpoints are merged into one. Finally, any checkpoint in $\mathcal{C}$ whose lastly performed expansion is empty is removed from $\mathcal{C}$. We remark that procedure `UpdateCheckpoints` only modifies the collection of checkpoints $\mathcal{C}$ and this procedure performs no clearing moves.

Thus, to summarize, the 'lifetime' of a checkpoint is as follows. Once the 1-st expansion of C is performed, the checkpoint will remain in the collection $\mathcal{C}$ and possibly more expansions of C are made (in total at most $\sqrt{n}$ expansion are possible for each checkpoint). A checkpoint C may disappear from $\mathcal{C}$ in three ways:

– when C is in its 0-th expansion and another checkpoint C' appears in the same frontier (thus, C' is in its 0-th expansion) and then the nodes of C are added to C', or

Procedure `UpdateCheckpoints`

Input: $C\langle\sqrt{n}\rangle$ and the collection of all checkpoints $\mathcal{C}$
Result: Updated collection $\mathcal{C}$
$\mathcal{C} \leftarrow \mathcal{C} \setminus \{C\}$
$\mathcal{C}_{\text{new}} \leftarrow \emptyset$
for each frontier F on $\sqrt{n}$-th rectangle of the frontier containing C **do**
Let C' consist of all guarded nodes in F.
If $C' \neq \emptyset$, then $\mathcal{C}_{\text{new}} \leftarrow \mathcal{C}_{\text{new}} \cup \{C'\}$.
for each C'' in $\mathcal{C}$ **do**
if there exists $C' \in \mathcal{C}_{\text{new}}$ that is a subset of the same frontier as C'' **then**
if C'' is in 0-th expansion **then**
$\mathcal{C} \leftarrow \mathcal{C} \setminus \{C''\}$
Replace C' with $C'' \cup C'$ in $\mathcal{C}_{\text{new}}$.
$\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}_{\text{new}}$
for each C in $\mathcal{C}$ **do**
if no node in the last expansion of C is guarded **then**
$\mathcal{C} \leftarrow \mathcal{C} \setminus \{C\}$

– some expansion of C becomes empty (then C is not removed from $\mathcal{C}$ right away but during the subsequent call to `UpdateCheckpoints`), or

– C reaches its $\sqrt{n}$-th expansion and procedure `UpdateCheckpoints` is called for C (in which case C possibly 'gives birth' to new checkpoints during the execution of `UpdateCheckpoints`).

Our algorithm maintains a collection $\mathcal{C}$ of currently used checkpoints.

3.4.3 Procedure `GridSearching`

`GridSearching` is the main algorithm, whose aim it is to clear the entire partial grid G in a connected and monotone way. We start with an informal introduction of the algorithm. The search strategy it produces is divided into phases, which will formally be defined in the next section. In each step of the algorithm, a checkpoint with the highest number of nodes that need to be guarded is chosen and the next expansion is made on it. When one of the checkpoints reaches its $\sqrt{n}$-th expansion, then the current phase ends and the procedure `UpdateCheckpoints` is invoked. Thus, the division of search strategy into phases is dictated by consecutive calls to procedure `UpdateCheckpoints`. For an expansion C, in the pseudocode

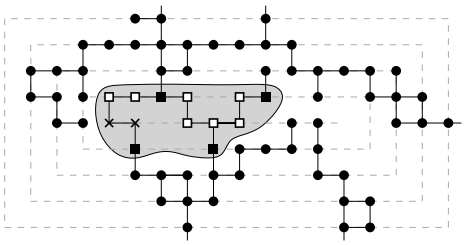
below we write $\delta(C)$ to refer to the set of nodes that belong to the last expansion of C and need to be guarded at a given point.

Procedure `GridSearching`

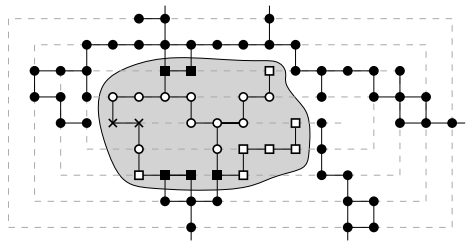
Input: An integer n providing an upper bound on the size of the partial grid G.
Result: A monotone connected search strategy for G.
Perform the initialization (see Section 3.4.1).
while G is not clear **do**
  while no checkpoint has reached its $\sqrt{n}$-th expansion **do**
    Let $C_{\max} \in \mathcal{C}$ be such that $\delta(C_{\max}) \geq \delta(C)$ for each $C \in \mathcal{C}$.
    Let i be the number of expansions of $C_{\max}$ performed so far.
    Invoke `ClearExpansion` for $C_{\max}\langle i \rangle$.
  Invoke `UpdateCheckpoints` for the $\sqrt{n}$-th expansion $C_{\max}\langle \sqrt{n} \rangle$ and for $\mathcal{C}$.

We now introduce a classification of searchers used in our algorithm. This classification will be used in the proof of Theorem 3.5.1 but we place it here as it provides another way of describing several actions that take place in the algorithm. We can divide searchers into three groups: *explorers*, *cleaners* and *guards*. Suppose that procedure `ClearExpansion` performs the i-th expansion of a checkpoint $C_{\max}$. Denote by $F_{\max}$ the frontier that contains the nodes in $C_{\max}$. All searchers located at nodes on the $(i-1)$-th rectangle of $F_{\max}$ that need to be occupied in order to avoid recontamination at the beginning of the call to procedure `ClearExpansion` are named to be guards. The explorers and cleaners are used by algorithm `ModConnectedSearching` called during the execution of procedure `ClearExpansion`. Each time `ModConnectedSearching` reaches a node v on the i-th rectangle of $F_{\max}$ such that v needs to be guarded, the searcher used for guarding v is called an explorer. The searchers used in `ModConnectedSearching` that mimic the movements of searchers in algorithm `ConnectedSearching` are the cleaners. We point out that we do not alter here the behavior of `ClearExpansion` and `ModConnectedSearching` but just assign one of the three categories to each searcher they use. Informally speaking, when explorers protect nodes lying on the i-th rectangle and the guards protect the ones lying on the $(i-1)$-th rectangle of $F_{\max}$, cleaners clear nodes inside the i-th rectangle of $F_{\max}$ (i.e., the remaining nodes of the i-th expansion of $C_{\max}$).

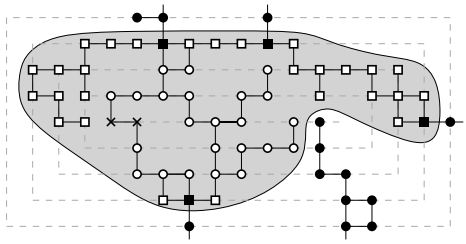
We close this chapter with giving examples of the first three expansions of some checkpoint C, see Figure 3.6, and showing how our algorithm clears an exemplary partial grid network, see Figure 3.7 (for a formal definition of a phase see the first paragraph of Section 3.5).



(A) First expansion of C.



(B) Second expansion of C.



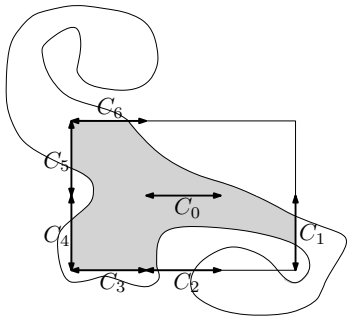
(C) Third expansion of C.

FIGURE 3.6: First three expansions for some checkpoint C (here $\sqrt{n} = 9$); *crosses* denote $C = C\langle 0 \rangle$, *empty circles* denote nodes cleared in previous expansions; *squares* denote nodes explored in the current expansion; *dark circles* are nodes not reached yet by the searchers; and *dark squares* denote nodes that need to be guarded at the end of current expansion. *Gray areas* show the clear part of the graph, i.e., $C^+\langle i \rangle$ for $i \in \{1, 2, 3\}$.

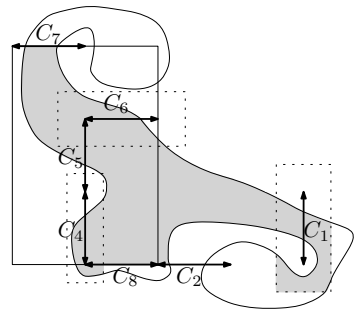
3.5 Analysis of the Algorithm

By a *step of the algorithm*, or simply a *step*, we mean all searching moves performed during a single iteration of the internal 'while' loop of procedure `GridSearching`. Thus, one step of the algorithm includes all moves produced by one call to procedure `ClearExpansion`. A *phase* of an algorithm consists of all its steps between two consecutive calls to procedure `UpdateCheckpoints`. Note that phases may differ with respect to the number of steps they are made of.

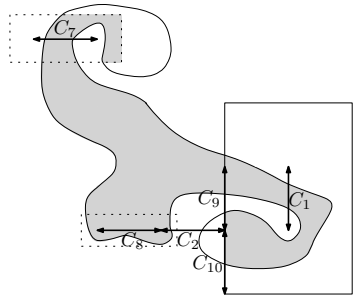
We say that a checkpoint is *present* in a given phase if its last expansion is not empty at the beginning of this phase, i.e., if this checkpoint belongs to $\mathcal{C}$ at the beginning of the phase. Similarly, a checkpoint is *present* in a given step if it is present in the phase to which the step belongs. Thus, in particular, a checkpoint is present in none or in all steps of a given phase. Note that some checkpoints may have empty expansions during a part of a the phase, but they still remain present to the end of the phase; this assumption is made to simplify the analysis of the algorithm.



(A) At the end of the first phase C_0 (initial checkpoint) reaches its $\sqrt{n}$-th expansion. Procedure `UpdateCheckpoints` creates 6 new checkpoints and removes C_0 from $\mathcal{C}$, i.e., $\mathcal{C} = \{C_1, C_2, C_3, C_4, C_5, C_6\}$.



(B) The $\sqrt{n}$-th expansion of C_5 ends the second phase. Checkpoints C_4 and C_6 are removed from $\mathcal{C}$ because (in our example) there is no need to guard any node on theirs expansions. C_3 is in 0-th expansion, so when a new checkpoint C_8 is created on the same frontier, C_3 is removed from $\mathcal{C}$; $\mathcal{C} = \{C_1, C_2, C_7, C_8\}$.

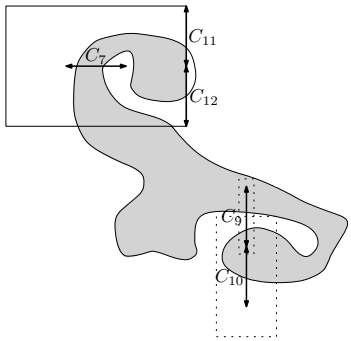


(C) Checkpoint C_1 ends the third phase. Notice that a new checkpoint C_9 emerged 'inside' the already cleared area by C_0; C_2 is removed from $\mathcal{C}$ even if $\sqrt{n}$-th expansion has not been reached but its last expansion has no nodes to be guarded; $\mathcal{C} = \{C_7, C_9, C_{10}\}$.

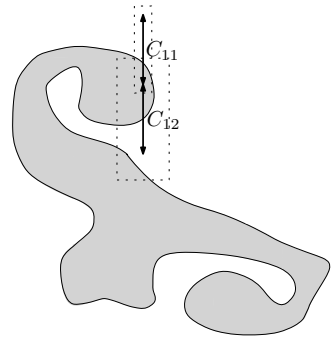
FIGURE 3.7: Clearing an exemplary partial grid by procedure `GridSearching`; *gray areas* denote the clear part, *arrows* denote frontiers on which the marked checkpoints lie, *dotted rectangles* around checkpoints denote their current expansions and *solid rectangles* denote the $\sqrt{n}$-th expansions, which end phases. (The final two phases are presented on Figure 3.8.)

Let t be a step and v be a node, which needs to be guarded at the beginning of step t. We say that the checkpoint C *owns* v *in step* t if:

– either C owns v in step $t-1$ or



(A) Checkpoint C_7 ends the fourth phase. Note that a new checkpoint C_{12} emerged on an edge of C_5's $\sqrt{n}$-th expansion; it was not created at the end of the second phase because then there was no access to the contaminated part; $\mathcal{C} = \{C_{11}, C_{12}\}$.



(B) Last phase, in which the rest of the graph is cleared.

FIGURE 3.8: Continuation of Figure 3.7

– no checkpoint owns v in step $t-1$ and v belongs to the last expansion of C performed till the end of step $t-1$.

(Intuitively, if a node v is reached by searchers in a step in which an expansion of C occurred, then C owns v as long as v is guarded.) We note that any vertex v can be owned by only one checkpoint. This follows from the fact that our strategy is monotone. More precisely, once v is owned by some checkpoint C in some step, then in the following steps it either continues to be owned by C or v does not need to be guarded. In the latter case v will not be owned by any checkpoint till the end of the strategy. Given a checkpoint C present in a step t, we write $\mathcal{E}(C,t)$ to denote the set of nodes that C owns in step t. The *weight of a checkpoint* C present in a step t is $\omega_t(C) = |\mathcal{E}(C,t)|$ and if a checkpoint C is not present in a step t, then we take $\omega_t(C) = 0$. Note that each guarded node is owned by exactly one checkpoint and hence, for a step t, the sum of weights of all checkpoints present in step t equals the number of nodes that need to be guarded.

The checkpoint $C_{\max}$ selected in a step t (see the pseudocode of Procedure `GridSearching`) is called *active in step* t, or simply *active* if the step is clear from the context or not important. All other checkpoints present in this step are called *inactive*. We define an *active interval* of a checkpoint C to be a maximal interval $[t', t'']$ such that C is active in all steps $t \in \{t', \ldots, t''\}$.

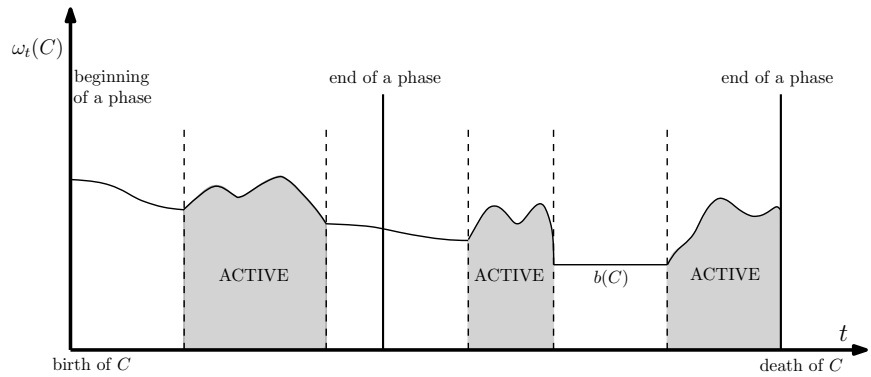


FIGURE 3.9: Exemplary life cycle of a checkpoint C.

3.5.1 Single Phase Analysis

We now prove lemmas that characterize how the weight of a checkpoint changes over time — see Figure 3.9 for an exemplary life cycle of a checkpoint. Informally, the weight of a checkpoint C does not grow in intervals in which C is inactive (Lemma 3.5.1). Also, the weight of C at the end of an active interval is not greater than at the beginning of it (Remark 3.5.1); however, no upper bounds except for the trivial one of $O(\sqrt{n})$ can be concluded for the weight of C inside its active interval.

Lemma 3.5.1. *If a checkpoint C is present and inactive in a step t, then $\omega_{t+1}(C) \leq \omega_t(C)$.*

Proof. It follows directly from procedure `ClearExpansion` and the definitions that the only checkpoint on which an expansion is performed during execution of `ClearExpansion` is the active one. The weight of an inactive checkpoint C can change only in the situation where the active checkpoint in a step t expands on some nodes owned by C. In other words, the weight of C may decrease if C contains in step t nodes that are added to the active checkpoint in step $t+1$. Thus, if t is not the last step of a phase, then the proof is completed.

If t is the last step of some phase, then apart from `ClearExpansion`, procedure `UpdateCheckpoints` is invoked, which affects C in two situations:

– there exists a step t' in the phase that ends such that $\omega_{t'}(C) = 0$. Then, because C cannot be expanded during steps $t', \ldots, t$ of the phase, we get directly that $\omega_{t+1}(C) = \omega_t(C) = 0$.

– C is in its 0-th expansion and a new checkpoint is placed on the same frontier, which implies that C is not present in step $t+1$ and thus $\omega_{t+1}(C) = 0$.

Thus, in all cases we obtain that $\omega_{t+1}(C) \leq \omega_t(C)$. □

We next observe that, informally speaking, once a checkpoint becomes active, it remains active until either the phase ends or its weight decreases. Note that a checkpoint that is active in the last step of the phase is not present in the first step of the next phase, i.e., its weight is then zero, which allows us to state the lemma as follows:

Lemma 3.5.2. *Let C be a checkpoint and let $[t', t'']$ be an active interval of C. For every step $t \in \{t', \ldots, t''\}$ it holds $\omega_t(C) \geq \omega_{t''+1}(C)$.*

Proof. Obviously, t and t'' must belong to the same phase, because at the end of each phase the active checkpoint is removed from $\mathcal{C}$, i.e., it is no longer present in the next phase.

If t'' is the last step of the phase then the lemma follows, because $\omega_{t''+1}(C) = 0 \leq \omega_{t'}(C)$.

We will now prove that lemma holds when t'' is not the last step of the phase. Let us suppose for a contradiction that $\omega_{t''+1}(C) > \omega_t(C)$. From the assumptions of the lemma and definition of an active interval we get that C is not the active checkpoint in step $t''+1$. Because we are still in the same phase, it means that there must exist a checkpoint C^* such that $\omega_{t''+1}(C^*) \geq \omega_{t''+1}(C)$. Moreover from Lemma 3.5.1 we know, that because C^* was inactive from step t' to t'', it holds $\omega_t(C^*) \geq \omega_{t''}(C^*) \geq \omega_{t''+1}(C^*)$. This gives us

$$\omega_t(C^*) \geq \omega_{t''+1}(C^*) \geq \omega_{t''+1}(C) > \omega_t(C), \tag{3.1}$$

which is in a contradiction to the assumption that C is the active checkpoint in step t. □

Remark 3.5.1. *Let C be a checkpoint and let $[t', t'']$ be an active interval of C. Then, $\omega_{t''+1}(C) \leq \omega_{t'}(C)$.*

We now conclude from the two previous lemmas about the weight of inactive checkpoints in the ends of the consecutive phases.

Lemma 3.5.3. *Suppose that a phase ends in a step t' and the next one ends in a step t''. If a checkpoint C is inactive (but present) in steps t' and t'', then $\omega_{t''}(C) \leq \omega_{t'}(C)$.*

Proof. Each checkpoint C can be active or inactive in different steps during the whole phase. If in some step $t \in \{t', \ldots, t''\}$ a checkpoint C is inactive then from Lemma 3.5.1 we have that its weight will not increase, i.e., $\omega_t(C) \geq \omega_{t+1}(C)$. On the other hand, Lemma 3.5.2 guarantees us, that the weight of an active checkpoint cannot be greater after its active interval than at the beginning.

□

3.5.2 How Many Nodes Are Explored by a Checkpoint?

Define a *bottleneck* of a checkpoint C, denoted by $b(C)$ to be its minimum weight taken over all steps in which C was present. (Note that a checkpoint may be present in many consecutive phases, see Figure 3.9.)

Suppose that a node v has been reached by a searcher for the first time in a step t. Let C be the active checkpoint in step t. We say that v has been *explored by* C.

If an expansion of an active checkpoint C reaches in a step t a node u already explored by some checkpoint C', then in most situations u does not need to be guarded. However there might occur a "corner situation" when u still needs to be guarded in order to avoid contamination. In such case, the algorithm clearly needs one searcher on u to guard it and so it is counted in our analysis due to the 'ownership' relation used in the definition of the weight of a checkpoint.

The next lemma states a lower bound on the number of nodes explored by a checkpoint reaching its last expansion.

Lemma 3.5.4. *Suppose that a phase ends in a step t. Let C be the active checkpoint in step t. The number of nodes explored by C in all steps is at least $b(C)\sqrt{n}$.*

Proof. First let us make a remark that nodes can be only explored by C during execution of procedure `ClearExpansion` that took C as an input, i.e., when C is active. Let us denote by S the set of all nodes explored by C.

Because C is active in the last step of the phase, it had to be active in exactly $\sqrt{n}$ steps in total, which can be contained in several past phases. Let $t_1, t_2, \ldots, t_{\sqrt{n}} = t$ be all steps in which C is active. Note that

$$\bigcup_{i=1}^{\sqrt{n}} \mathcal{E}(C, t_i) \subseteq S$$

and $\mathcal{E}(C, t_i) \cap \mathcal{E}(C, t_j) = \emptyset$ for $i \neq j$. The latter follows directly from the fact that nodes in $\mathcal{E}(C, t_i)$ and $\mathcal{E}(C, t_j)$ belong to different rectangles of the frontier containing C for $i \neq j$. (Recall that $|\mathcal{E}(C, t)| = \omega_t(C)$ for each step

t.) Also from the definition of the bottleneck, we get that $b(C) \leq \omega_{t_i}(C)$ for each $i \in \{1, \ldots, \sqrt{n}\}$ and hence we conclude that:

$$|S| \geq \sum_{i=1}^{\sqrt{n}} \omega_{t_i}(C) \geq b(C)\sqrt{n}. \tag{3.2}$$

□

We now give an upper bound on the weight of each inactive checkpoint at the end of a phase.

Lemma 3.5.5. *Suppose that a phase ends in a step t. Let $C_1, \ldots, C_l$ be all checkpoints present in this phase, where C_1 is the active checkpoint in step t. Then, $b(C_1) \geq \omega_t(C_j)$ for each $j \in \{2, \ldots, l\}$.*

Proof. Let us denote by t' the last step in which $\omega_{t'}(C_1) = b(C_1)$. If $t' = t$ then the lemma follows strictly from the definition of an active checkpoint. We will now prove that lemma stands also when $t' < t$.

Suppose that t' and t do not belong to the same active interval of C_1. From the Lemma 3.5.2 we know that $\omega_{t''}(C_1) = b(C_1)$ occurs for some t'' that does not belong to an active interval. Moreover from Remark 3.5.1 we get that every next active interval will need to start and finish on the same weight as the bottleneck, which is in contradiction that t' is the last step when $b(C_1)$ occurred.

Hence t and t' are part of the same active interval of C_1. Then, we get from Lemma 3.5.1 and the fact that C_1 is active in step t':

$$\omega_t(C_j) \leq \omega_{t'}(C_j) \leq \omega_{t'}(C_1) = b(C_1), \quad j \in \{2, \ldots, l\}, \tag{3.3}$$

which finishes our proof. □

Let us introduce a relation $\prec$ on a set of checkpoints. Whenever $C \prec C'$, we say that C is a *predecessor* of C' and C' is a *successor* of C. We stress out that the construction depends on the execution of the algorithm, namely only checkpoints that appear in some step are considered, and the division of the steps into phases shapes the relation. More precisely, the relation is defined only for checkpoints added to the set $\mathcal{C}$ during all executions of procedure `UpdateCheckpoints`. To construct the relation we iterate over the consecutive phases of the algorithm. Initially the relation is empty and once the construction is done for each phase smaller than i, we perform the following for phase i. Let C be the active checkpoint in the last step of phase i. Let $C_1, \ldots, C_l$ be all checkpoints, different from C, that have no

successors so far and were added to $\mathcal{C}$ till the end of phase $i-1$ (including the last step). Then, let $C_j \prec C$ for each $j \in \{1, \ldots, l\}$.

An important property of our algorithm is that each checkpoint may have only a constant number of predecessors:

Lemma 3.5.6. *Each checkpoint has at most* 10 *predecessors.*

Proof. A checkpoint C can only once be active in the last step of some phase i, because after that it will not be present in any later phases. At the end of phase i the only checkpoints that do not have any successors are the ones that were constructed by the procedure `UpdateCheckpoints` at the end of phase $i-1$. There are at most 10 such checkpoints. □

3.5.3 The Algorithm Uses $O(\sqrt{n})$ Searchers in Total

We now bound the total weight of all checkpoints at the end of each phase — note that this bounds the total number of searchers used for guarding at the end of a phase. A high level intuition behind the proof of Lemma 3.5.7 is as follows. Due to Lemma 3.5.4, each checkpoint C that is active in the last step of a phase explores at least $b(C)\sqrt{n}$ nodes in total. Therefore, the sum of bottlenecks of all such checkpoints C cannot exceed $\sqrt{n}$. Moreover, C can have at most 10 predecessors and hence the sum of weights of those predecessors is bounded by $10b(C)$ according to Lemma 3.5.5. Since each checkpoint (except the one that is active in the last step of a given phase) is a predecessor of some checkpoint that is active in the last step of some phase, we bound the sum of all weights of all such checkpoints present in a given phase by $10\sqrt{n}$.

Lemma 3.5.7. *Suppose that $C_1, \ldots, C_l$ are all checkpoints present in a phase that ends in step t, where C_1 is active in step t. Then,*

$$\sum_{i=1}^{l} \omega_t(C_i) \leq \omega_t(C_1) + 10\sqrt{n}.$$

Proof. Suppose that phase j ends in step t. Let t_i be the last step of phase i and let C_i^0 be the active checkpoint in step t_i for each $i \in \{0, \ldots, j\}$. We denote by s the number of nodes visited by searchers till the end of step $t = t_j$. From Lemma 3.5.4 and the fact that the number of all nodes n is at least s we have:

$$n \geq s \geq \sum_{i=0}^{j} b(C_i^0)\sqrt{n} \quad \Rightarrow \quad 10\sqrt{n} \geq 10 \sum_{i=0}^{j} b(C_i^0). \tag{3.4}$$

From Lemma 3.5.6 we have that the checkpoints $C_0^0, \ldots, C_j^0$ can have at most 10 predecessors. From the definition, they are constructed (i.e., added to collection $\mathcal{C}$ during the execution of procedure `UpdateCheckpoints`) at the beginning of the first step of a phase at the end of which their successor is active. Let us denote by $C_i^1, \ldots, C_i^{l_i}$, $0 \leq l_i \leq 10$, the predecessors of C_i^0 for each $i \in \{0, \ldots, j\}$ (by $l_i = 0$ we denote that C_i^0 has no predecessors). From Lemma 3.5.5 we have:

$$\sum_{k=1}^{l_i} \omega_{t_i}(C_i^k) \leq 10b(C_i^0), \quad i \in \{0, \ldots, j\}. \tag{3.5}$$

Lemma 3.5.3 assures us that weights of inactive checkpoints will not be greater at the end of the next phase than they are in the last step of current phase:

$$\begin{aligned} \omega_t(C_i^k) = \omega_{t_j}(C_i^k) \leq \omega_{t_{j-1}}(C_i^k) \leq \cdots \leq \omega_{t_i}(C_i^k), \\ i \in \{0, \ldots, j\};\ k \in \{1, \ldots, l_i\}. \end{aligned} \tag{3.6}$$

Because

$$\{C_1, \ldots, C_l\} \subseteq \{C_j^0\} \cup \left\{ C_i^k \mid k \in \{1, \ldots, l_i\}, i \in \{0, \ldots, j\} \right\},$$

we can conclude from Equations (3.6), (3.5) and (3.4) (in this order) that:

$$\begin{aligned} \sum_{i=1}^{l} \omega_t(C_i) &\leq \omega_t(C_j^0) + \sum_{i=0}^{j} \sum_{k=1}^{l_i} \omega_t(C_i^k) \\ &\leq \omega_t(C_j^0) + \sum_{i=0}^{j} \sum_{k=1}^{l_i} \omega_{t_i}(C_i^k) \\ &\leq \omega_t(C_j^0) + \sum_{i=0}^{j} 10b(C_i^0) \\ &\leq \omega_t(C_j^0) + 10\sqrt{n}. \end{aligned} \tag{3.7}$$

□

Theorem 3.5.1. *Given an upper bound n of the size of the network as an input, the algorithm* `GridSearching` *clears in a connected and monotone way any unknown underlying partial grid network using $O(\sqrt{n})$ searchers.*

Proof. At first let us notice that the algorithm `GridSearching` ends with the whole network cleared. Indeed, as long as there are contaminated nodes,

it will continue clearing next expansions of the checkpoints. Because no recontamination takes place, it eventually terminates. We will bound the number of searchers s used by a single call to procedure `ClearExpansion` and the total number of searchers s' used for guarding at the end of any step of the algorithm. Note that $s + s'$ bounds the total number of searchers used by `GridSearching`. In the proof we refer to the classification of searchers into explorers, cleaners and guards introduced in Section 3.4.

We first analyze procedure `ClearExpansion` to give an upper bound on s. The fact that each rectangle of a frontier contains at most $10\sqrt{n}$ nodes and Theorem 3.4.1 give that:

$$\begin{aligned} \text{number of explorers} &\leq 10\sqrt{n}, \\ \text{number of cleaners} &\leq 6\sqrt{n} + 4. \end{aligned}$$

Thus,

$$s \leq 16\sqrt{n} + 4. \tag{3.8}$$

The guards used to protect nodes lying on the $(i-1)$-th rectangle are accounted for during the estimation of s' below.

We now bound the maximal number of searchers used for guarding at the end of each step t of our search strategy, which we denote by g_t. It is easy to see that $g_t \leq 10\sqrt{n}$ if t belongs to phase 0.

Let us now take any step t that belongs to an i-th phase, where $i > 0$ and denote by t' the last step of the phase $i-1$ and by C the active checkpoint in step t'. From Lemma 3.5.7 we know that $g_{t'} \leq \omega_{t'}(C) + 10\sqrt{n} \leq 20\sqrt{n}$. The latter inequality follows from the fact that all nodes in $\mathcal{E}(C, t')$ belong to the j-th rectangle of the frontier that contains C, $j \leq \sqrt{n}$, and the number of nodes in this rectangle is at most $10\sqrt{n}$.

We know now that every phase starts with at most $20\sqrt{n}$ guards. If t is the first step of an active interval of some checkpoint, then by Lemma 3.5.1 and Remark 3.5.1 we have that $g_t \leq g_{t'} \leq 20\sqrt{n}$. But if t is a step inside some active interval, then an active checkpoint can reach at most $10\sqrt{n}$ new nodes that need to be guarded. Note that by Lemma 2, the nodes of subsequent expansions of a checkpoint that need to be guarded do not accumulate, that is, we only guard the one of the last expansion. Because in one step only one checkpoint can be active that leads us to conclusion that for every step t we have $g_t \leq 30\sqrt{n}$. Therefore, we obtain that $s' \leq 30\sqrt{n}$.

Thus, we obtain $s + s' \leq 46\sqrt{n} + 4 = O(\sqrt{n})$ as required.

□

3.6 Unknown Size of the Graph

The algorithm we have described needs to know an upper bound on the size of the underlying partial grid network G. In this section we design a procedure called `ModGridSearching` that performs the search using $O(\sqrt{n})$ searchers and having no prior information on the network. The procedure is based on a standard technique: guessing an upper bound on n by doubling potential estimate each time. More about applications of the doubling technique in designing on-line and off-line approximation algorithms can be found in [35].

The procedure `ModGridSearching` is composed of a certain number of *rounds*. In round i, procedure `GridSearching` first introduces $c\sqrt{2^i}$ new searchers called *i-th team*, where c is the constant from the asymptotic notation in Theorem 3.5.1. Then, a call to `GridSearching` is made, where procedure `GridSearching` is using only the searchers of the i-th team. The outcome can be twofold. The procedure may succeed in searching the entire graph and in such case the i-th round is the last one and `ModGridSearching` is completed, or the procedure may encounter a situation in which it would be forced to use more than $c\sqrt{2^i}$ searchers to continue. In such case `GridSearching` stops, the i-th round ends and the $(i+1)$-th round will follow. Once the i-th round is completed, the searchers of the i-th team stay idle indefinitely. We point out that during the execution of an i-th round, $i > 1$, procedure `GridSearching` using the searchers of the i-th team is ignoring the fact that the network may be partially clear as a result of the work done in previous rounds. Moreover, the searchers of j-th team for each $j < i$ are not used and thus also ignored during i-th round.

We close this section by giving an upper bound on the number of searchers that need to be used in the presented modified version of our algorithm.

Theorem 3.6.1. *The distributed on-line algorithm* `ModGridSearching` *clears (starting at an arbitrary homebase) in a connected and monotone way any unknown underlying partial grid network using $O(\sqrt{n})$ searchers. The algorithm receives no prior information on the network.*

Proof. Let n be the number of nodes of the partial grid network, which is unknown to our procedure. The number of rounds m fulfills $2^{m-1} < n \leq 2^m$, i.e., $m = \lceil \log_2 n \rceil$. At the end of i-th round, $c\sqrt{2^i}$ searchers need to stay in their last positions till the end of our procedure and are not used in subsequent rounds. This means that the total number of searcher s is

upper bounded by a sum of searchers used in every round:

$$s \leq c\sqrt{2} + c\sqrt{2^2} + \cdots + c\sqrt{2^{\lceil \log_2 n \rceil}} = c \sum_{j=1}^{\lceil \log_2 n \rceil} \left(\sqrt{2}\right)^j$$

$$= \sqrt{2}c\frac{1-\sqrt{2}^{\lceil \log_2 n \rceil}}{1-\sqrt{2}} = \frac{\sqrt{2}c}{\sqrt{2}-1}\left(\sqrt{2^{\lceil \log_2 n \rceil}} - 1\right). \tag{3.9}$$

Because $\sqrt{n} \leq \sqrt{2^{\lceil \log_2 n \rceil}} < \sqrt{2n}$, we conclude

$$s < \frac{\sqrt{2}c}{\sqrt{2}-1}\left(\sqrt{2n} - 1\right) \quad \Rightarrow \quad s = O(\sqrt{n}).$$

□

3.7 Conclusions

3.7.1 Motivation

There exists a number of studies of graph searching problems in the graph-theoretic context. Much less is known for geometric scenarios. It turns out that the geometric (or continuous) analogue of graph searching is challenging to analyze. More precisely, in the recently introduced continuous version [97, 117] the input geometric shape is searched by using line segments or curves (that form a barrier separating contaminated and clear area) instead of searchers. The corresponding optimization criterion is then the total length of this barrier. It can be observed that computing optimal strategies even for some simple shapes turns out to be quite non-trivial [117].

The class of graphs we have selected to study in this chapter is motivated by the following arguments. First, on-line (monotone) searching turns out to be difficult in terms of achievable upper bound on the number of searchers even in simple topologies like trees. This suggest that some additional information is needed to perform on-line search efficiently and our work shows that, informally speaking, a two-dimensional sense of direction is enough to search a graph in asymptotically almost optimal way. Our second motivation comes from approaching the problem of geometric search by considering its discrete analogues, i.e., by modeling via graph theory. We give a short sketch to give an overview as the problem of modeling is out of scope of this work and we only refer to some recent works on the subject [3, 14, 97, 117]. Consider a continuous search problem in which

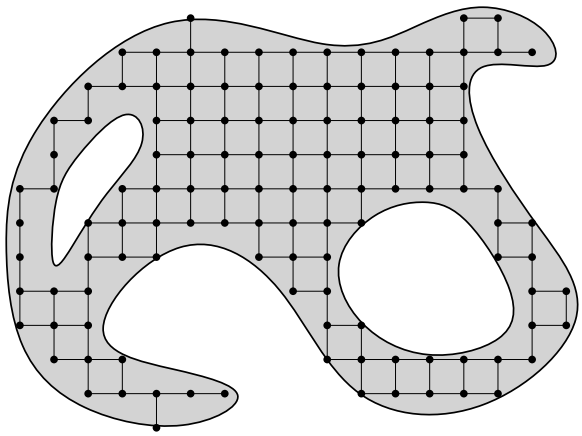


FIGURE 3.10: An example of the construction of a partial grid network.

k searchers initially placed at the same location need to capture the fugitive hiding in an arbitrary polygon that possibly has holes. The polygon is not known a priori to the searchers. The fugitive is considered captured in time t when it is located at distance at most r from some searcher at time point t. (The distance r can be related to physical dimensions of searchers and/or their visibility range, etc.)

Consider the following transition from the above continuous searching problem of a polygon to a discrete one. Overlap the coordinate system with the polygon in such a way that the origin coincides with the original placement of the searchers. Then, place nodes on all points with coordinates, which are multiples of r and lie in the polygon. Connect two nodes with an edge if the edge is contained in the polygon. In this way we obtain a partial grid network. In this brief sketch we omit potential problems that may arise in such modeling, like obtaining disconnected networks or having 'blind spots', i.e., points in the polygon that cannot be cleared by using the above nodes and edges only. We say that a partial grid network G *covers* the polygon if G is connected and for each point p in the polygon there exist a node of G in distance at most r from p. See Figure 3.10 for an example.

Note that any search strategy $\mathcal{S}'$ for a polygon P can be used to obtain a search strategy $\mathcal{S}$ for underlying partial grid network G as follows. For each searcher s used in $\mathcal{S}'$ introduce four searchers $s_1, \ldots, s_4$ that will 'mimic' its movements by going along edges of G. More precisely, the searchers $s_1, \ldots, s_4$ will ensure that at any point, if s is located at a point (x, y), then $s_1, \ldots, s_4$ will reside on nodes with coordinates $(\lfloor x/r \rfloor, \lfloor y/r \rfloor)$,

$(\lfloor x/r \rfloor, \lceil y/r \rceil)$, $(\lceil x/r \rceil, \lfloor y/r \rfloor)$, $(\lceil x/r \rceil, \lceil y/r \rceil)$. In this way, area protected by s in $\mathcal{S}'$ is always protected by four searchers in $\mathcal{S}$. This allows us to state the following.

Observation 3.7.1. *Let P be a polygon and let G by an underlying partial grid network that covers P. Then, there exists a search strategy for G using k searchers such that its execution in G results in clearing P and $k = O(p)$, where p is the minimum number of searchers required for clearing P (in a continuous way).*

3.7.2 Open Problems

In view of the lower bound shown in [93] that even in such simple networks as trees each distributed on-line algorithm may be forced to use $\Omega(n/\log n)$ times more searchers than the connected search number of the underlying network, one possible line of research is to restrict attention to specific topologies that allow to obtain algorithms with good provable upper bounds. This work gives one such an example. An interesting research direction is to find other non-trivial settings in which distributed on-line search can be conducted efficiently. Also, we leave a logarithmic gap in our approximation ratio. Since there exist grids that require $\Omega(\sqrt{n})$ searchers the gap can be possibly closed by analyzing the grids that require few (e.g. $O(\log n)$) searchers.

The above questions related to network topologies can be stated more generally: what properties of the on-line model are crucial for such a search for fast and invisible fugitive to be efficient? This work and also a recent one [27] suggest that a 'sense of direction' may be one such a factor. Possibly interesting directions may be to analyze the influence of visibility on search scenarios.

We finally note that the only optimization criterion that was of interest in this work is the number of searchers. This coincides with the research done in off-line search problems where this was the most important criterion giving nice ties between graph searching theory and structural graph theory. However, one may consider adding different optimization criteria like time or the total distance.

Chapter 4

Finding Small-width Connected Path Decompositions

Since the famous 'graph minor' project by Robertson and Seymour that started with [137], the notions of treewidth and pathwidth received growing interest and a vast amount of results has been obtained. The pathwidth, informally speaking, allows us to say how closely an arbitrary graph resembles a path. This concept proved to be useful in designing algorithms for various graph problems, especially in the case when the pathwidth of an input graph is small (e.g. fixed), in which case quite often a variant of the well-known dynamic programming approach that progresses along a path decomposition of the input graph turns out to be successful.

Several modifications to pathwidth have been proposed and in this chapter we are interested in the connected variant in which one requires that a path decomposition $(X_1, \ldots, X_l)$ of a graph G satisfies: the vertices $X_1 \cup \cdots \cup X_i$ induce a connected subgraph in G for each $i \in \{1, \ldots, l\}$. This version of the classical pathwidth problem is motivated by several pursuit-evasion games, including, but not limited to, edge search, node search or mixed search [99, 132, 146]. More precisely, computing the minimum number of searchers needed to clean a given graph G in the node search game (i.e., computing the node search number of G) is equivalent to computing the connected pathwidth of G. Moreover, a connected path decomposition can be easily translated into the corresponding node search strategy that cleans G and vice versa.

For more about different search numbers, connected search strategies and computing the search numbers see survey on the decontamination problem in Section 2.1 (in particular subsections 2.1.1, 2.1.2 and 2.1.5).

This chapter is constructed as follows: we start with giving the motivation for our research and in the following section we recall the definition

of connected pathwidth and related terms used in this chapter. Section 4.3 provides a polynomial-time algorithm for determining whether the connected pathwidth of an arbitrary input graph G is at most k, where k is a fixed integer. The algorithm is inspired by the algorithms for computing minimum-length path decompositions by Dereniowski, Kubiak, and Zwols [50]. We also use the notation from this paper. Then, Section 4.4 contains the analysis of the algorithm (its correctness and running time). We finish with some open problems in Section 4.5.

4.1 Motivation

The connectivity constraint for pathwidth is natural and useful in graph searching games [9, 68, 82]. The connectivity is in some cases implied by potential applications (e.g., security constraints may enforce the clean, or safe, area to be connected) or it is a necessity, like in distributed or on-line versions of the problem [22, 27, 93, 121].

Our second motivation comes from connections between pathwidth and connected pathwidth. More specifically, [47] implies that for any graph G, these parameters differ multiplicatively only by a small constant. This implies that an approximation algorithm for connected pathwidth immediately provides an approximation algorithm for pathwidth with asymptotically the same approximation ratio. This may potentially lead to obtaining better approximations for pathwidth since, informally speaking, the algorithmic search space for connected pathwidth is for some graphs much smaller than that for pathwidth. On the other hand, we do not know any algorithm computing the connected pathwidth in time $O^*((2-\epsilon)^n)$, for any $\epsilon > 0$. Thus, despite this smaller algorithmic search space, it is not clear how these two problems algorithmically differ in the context of designing exact algorithms.

During the GRASTA 2017 workshop, Fedor V. Fomin [75] raised an open question, whether we can verify in polynomial time, if the connected pathwidth of a given graph is at most k, for a fixed constant k. In this chapter we answer this question in the affirmative.

4.2 Definitions

Let $G = (V(G), E(G))$ be a simple graph. We recall the definition of a path decomposition.

Definition 4.2.1. *A* path decomposition *of a simple graph $G = (V(G), E(G))$ is a sequence $\mathcal{P} = (X_1, \ldots, X_l)$, where $X_i \subseteq V(G)$ for each $i \in \{1, \ldots, l\}$, and*

A. $\bigcup_{i=1}^{l} X_i = V(G)$,

B. *for each* $\{u,v\} \in E(G)$ *there exists* $i \in \{1,\ldots,l\}$ *such that* $u,v \in X_i$,

C. *for each* i,j,k *with* $1 \leq i \leq j \leq k \leq l$ *it holds that* $X_i \cap X_k \subseteq X_j$.

The width *of a path decomposition* $\mathcal{P}$ *is* $\mathrm{width}(\mathcal{P}) = \max_{i\in\{1,\ldots,l\}} |X_i| - 1$. *The* pathwidth *of* G*, denoted by* $\mathrm{pw}(G)$*, is the minimum width over all path decompositions of* G.

We say that a path decomposition $\mathcal{P} = (X_1,\ldots,X_l)$ is *connected* if the subgraph $G[X_1 \cup \cdots \cup X_i]$ is connected for each $i \in \{1,\ldots,l\}$. The *connected pathwidth* of a graph G, denoted by $\mathrm{cpw}(G)$, is the minimum width taken over all connected path decompositions of G.

Finally, a *connected partial path decomposition* of a graph G is a connected path decomposition $(X_1,\ldots,X_i)$ of some subgraph H of G, where $N_G(V(G) \setminus V(H)) \subseteq X_i$. In other words, in the latter condition we require that each vertex of H that has a neighbor outside H belongs to the last bag X_i. Intuitively, a connected partial path decomposition of G can be potentially a prefix of some connected path decomposition of G. Also note that if $(X_1,\ldots,X_l)$ is a connected path decomposition, then for each $i \in \{1,\ldots,l\}$, the sequence $(X_1,\ldots,X_i)$ is a connected partial path decomposition of G, where $H = G[X_1 \cup \cdots \cup X_i]$.

In our analysis we use the following intermediate notion between arbitrary and connected path decompositions.

Definition 4.2.2. *Given* $\mathcal{I} \subseteq V(G)$*, a (partial) path decomposition* $\mathcal{P} = (X_1,\ldots,X_l)$ *of* G *is* $\mathcal{I}$-connected *if for each* $i \in \{1,\ldots,l\}$ *each connected component* H *of* $G[X_1 \cup \cdots \cup X_i]$ *contains a vertex from* $\mathcal{I}$ *and this vertex belongs to the first bag in which* H *appears in* $\mathcal{P}$.

In other words, if $\mathcal{P} = (X_1,\ldots,X_l)$ is a partial path decomposition or a prefix of one, then the subgraph of G induced by the union of bags of $\mathcal{P}$ may have several connected components and each of them must have a vertex in $\mathcal{I}$. Moreover, if one looks at the path decomposition of such a connected component H derived from $\mathcal{P}$, i.e. at the path decomposition obtained from $(X_1 \cap V(H),\ldots,X_l \cap V(H))$ by removing the empty bags, then such a decomposition starts with a bag containing a vertex in $\mathcal{I}$. Note that if $\mathcal{P}$ is connected then there is only one such connected component. See Figure 4.1 for an illustration.

In our analysis, we will assume that the path decompositions we consider have the property that each bag introduces at most one new vertex, i.e., for $\mathcal{P} = (X_1,\ldots,X_l)$ it holds $|X_{i+1} \setminus X_1| \leq 1$ for each $i \in \{1,\ldots,l-1\}$. This property can be easily reestablished whenever needed (a folklore).

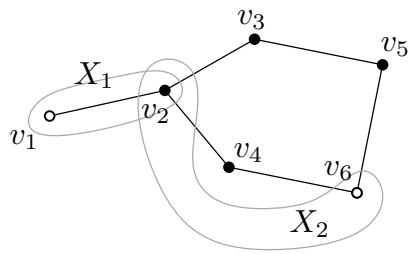


FIGURE 4.1: An illustration of $\mathcal{I}$-connectivity; *empty circles* denote vertices from the set $\mathcal{I}$. A partial path decomposition $\mathcal{P} = (X_1, X_2)$ is $\mathcal{I}$-connected, as every connected subgraph of $G[X_1]$ and $G[X_1 \cup X_2]$ contains a vertex from $\mathcal{I}$ as required in Definition 4.2.2.

4.3 The Algorithm

We will present an algorithm that decides whether, for a connected input graph G and a set $\mathcal{I}$, there exists an $\mathcal{I}$-connected path decomposition of width at most $k-1$ (thus k is the maximum bag size of the connected path decomposition to be computed). In the following, G, $\mathcal{I}$ and k are hence fixed. Note that we may without loss of generality assume that the first bag in a connected path decomposition to be computed has only one vertex.

We start by informally sketching the high-level idea of the algorithm. Then, Sections 4.3.1 and 4.3.2 give the algorithm and Section 4.3.3 gives a summary of our method. We use a dynamic programming approach. To that end we introduce a set of states, so that each state encodes some partial path decomposition of G. If a transition from one state to another is possible, then this certifies that the partial path decomposition corresponding to one state can be extended to (i.e., is a prefix of) a partial path decomposition corresponding to the other state. We will verify whether the transition is possible using a recursive procedure. The need to ensure consistency of solutions found in recursive calls is the reason why we consider a more general problem of finding $\mathcal{I}$-connected path decompositions.

Finally, we will show that each transition can be checked in polynomial time and that it is enough to consider only polynomially many states. This eventually leads to an algorithm with a desired complexity.

For any $S \subseteq V(G)$, we say that a subgraph H of G is an *S-branch* if H is a connected component of $G - S$ and $N_G(V(H)) = S$. For any $S \subseteq V(G)$, define $\mathcal{B}(S)$ to be the set of all S-branches. A set S is called a *bottleneck* if the number of S-branches is at least $2k+1$, as it guarantees us the existence of at least one special branch called an *in-branch*, which will be defined formally in the next section. Observe that each connected component of

$G - X$, for any $X \subseteq V(G)$, is an S-branch for exactly one non-empty subset S of X.

Let us mention that S-branches are also known as *full components associated with* S (see e.g. Bouchitté and Todinca [29]).

4.3.1 States

By a *potential state* we mean a triple $(X, \{B_S\}_{S\subseteq X}, \{f^{B_S}\}_{S\subseteq X})$, consisting of:

- a non-empty set $X \subseteq V(G)$ with $|X| \leq k$,
- a subset $B_S \subseteq \mathcal{B}(S)$ of cardinality at most $2k$, chosen for every non-empty $S \subseteq X$,
- a function $f^{B_S} \colon \mathcal{B}(S) \to \{0,1\}$, chosen for every non-empty $S \subseteq X$. We additionally require that if $H, H' \in \mathcal{B}(S) \setminus B_S$, then $f^{B_S}(H) = f^{B_S}(H')$.

The exact meaning of B_S will be explained later on, but let us present some intuition. In Lemma 4.4.1 we show that for every path decomposition $\mathcal{P}$, the vertices of all but at most $2k$ S-branches appear in bags of $\mathcal{P}$ in a certain, well-structured way. The set B_S and the function f^{B_S} will be used to describe the structure of the remaining, badly behaving S-branches.

Observe that the set X may be chosen in at most n^k ways and the number of choices of S is at most 2^k. For every S, the number of S-branches is at most n, so B_S can be chosen in at most n^{2k} ways. The function f^{B_S} can be chosen in at most $2^{|B_S|} \cdot 2 \leq 2^{2k+1}$ ways. Therefore, the number of potential states is at most $n^k \cdot 2^k \cdot n^{2k} \cdot 2^{2k+1} = O(n^{3k})$, i.e., polynomial in n, where the asymptotic notation hides the factor that depends on k.

With a potential state $s = (X, \{B_S\}_{S\subseteq X}, \{f^{B_S}\}_{S\subseteq X})$ we associate the following notions. By $\mathrm{bag}(s)$ we denote the set X. By $\mathrm{cover}(s)$ we denote the set of vertices

$$X \cup \bigcup_{S\subseteq X, S\neq\emptyset} \left(\bigcup_{H\in\mathcal{B}(S)\colon f^{B_S}(H)=1} V(H) \right).$$

By G_s we denote the subgraph of G induced by $\mathrm{cover}(s)$. We say that two states w, s are *indistinguishable* if $\mathrm{cover}(w) = \mathrm{cover}(s)$ and $\mathrm{bag}(w) = \mathrm{bag}(s)$. Otherwise the states are *distinguishable*. In particular, indistinguishable states can only differ by the choice of B_S. We note that for such two distinguishable states it may hold that $\mathrm{cover}(w) = \mathrm{cover}(s)$ but $\mathrm{bag}(w) \neq \mathrm{bag}(s)$.

Example. *In the example in Figure 4.1 we can consider two states: s and w with bags* $\mathrm{bag}(s) = X_1$ *and* $\mathrm{bag}(w) = X_2$. *There exists one $\{v_1\}$-branch $H_1 = G - \{v_1\}$, two $\{v_2\}$-branches: $H_2 = G[\{v_1\}]$ and $H_3 = G[\{v_3, v_4, v_5, v_6\}]$, one $\{v_6\}$-branch $H_4 = G - \{v_6\}$ and two $\{v_2, v_6\}$-branches: $H_5 = G[\{v_4\}]$ and $H_6 = G[\{v_3, v_5\}]$. We notice that the set of $\{v_1, v_2\}$-branches is empty. The state s is equal to $(X_1, \{\{H_1\}, \{H_2, H_3\}, \emptyset\}, \{f_1, f_2, f_\emptyset\})$ and the state w is $(X_2, \{\{H_2, H_3\}, \{H_4\}, \{H_5, H_6\}\}, \{f_2, f_3, f_4\})$, such that $f_\emptyset$ is a function with empty domain, $f_1(H_1) = f_2(H_3) = f_3(H_4) = f_4(H_5) = f_4(H_6) = 0$ and $f_2(H_2) = 1$. We notice that* $\mathrm{cover}(s) = \{v_1, v_2\}$ *and* $\mathrm{cover}(w) = \{v_1, v_2, v_6\}$.

Let v be a vertex from $\mathrm{cover}(s)$ which has a neighbor $u \notin \mathrm{cover}(s)$. We argue that $v \in \mathrm{bag}(s)$. Otherwise, if $v \notin \mathrm{bag}(s)$, then both v and u belong to the same S-branch for some $S \subseteq \mathrm{bag}(s)$. Thus, they are either both in $\mathrm{cover}(s)$, or outside it. From this it follows that every vertex $v \in \mathrm{cover}(s)$, which has a neighbor $u \notin \mathrm{cover}(s)$, must belong to $\mathrm{bag}(s)$. Let us denote the set of such vertices v by $\mathrm{border}(s)$. We have proved:

Observation 4.3.1. *For each potential state s it holds that* $\mathrm{border}(s) \subseteq \mathrm{bag}(s)$.

We say that a potential state s is a *state* if each connected component of G_s contains a vertex in $\mathcal{I}$.

We introduce a Boolean table *Tab*, indexed by all states. For a state s, the value of $Tab[s]$ will be set to *true* by our algorithm if and only if there exists some $\mathcal{I}$-connected partial path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_l)$ of G with $H = G_s$, such that $\mathrm{width}(\mathcal{P}) \leq k - 1$ and $X_l = \mathrm{bag}(s)$. We will use a dynamic programming to fill the table *Tab*. Then, we will conclude that G has an $\mathcal{I}$-connected path decomposition of width at most $k - 1$ if and only if $Tab[s] = true$ for some state s with $\mathrm{cover}(s) = V(G)$.

Observe that such a final state exists, since for $s = (X, \{B_S\}_S, \{f^{B_S}\}_S)$, we have $\mathrm{cover}(s) = V(G)$ if and only if $f^{B_S}(H) = 1$ for every S and H. However, the astute reader may notice that in our representation we might have not included some $\mathcal{I}$-connected partial path decompositions and it could be possible that we do not find a solution, even though it exists. We will show that if $\mathrm{cpw}(G) \leq k - 1$, then there exists a special type of an $\mathcal{I}$-connected path decomposition of width at most $k - 1$, called a *structured path decomposition* (defined later), which can be found using our algorithm because, as we will argue, our table *Tab* does not 'omit' any structured path decompositions.

4.3.2 Extension Rules

Let us introduce a total ordering $\prec$ on the set of states. We say that $w \prec s$ if $|\mathrm{cover}(w)| < |\mathrm{cover}(s)|$, or $|\mathrm{cover}(w)| = |\mathrm{cover}(s)|$ and $|\mathrm{bag}(w)| >$

$|\text{bag}(s)|$. If $|\text{cover}(w)| = |\text{cover}(s)|$ and $|\text{bag}(w)| = |\text{bag}(s)|$, then we resolve such a tie arbitrarily.

We initialize *Tab* by setting $Tab[s] = true$ for every state s, such that $\text{cover}(s) = \text{bag}(s) = \{v\}$, for some $v \in \mathcal{I}$, while for the remaining states s we initialize $Tab[s]$ to be $false$. In particular for each s with $|\text{cover}(s)| = 1$ and $\text{cover}(s) = \text{bag}(s) \in \mathcal{I}$, we have $Tab[s] = true$. In our dynamic programming algorithm, we process states according to the ordering $\prec$ and fill the table *Tab* using two extension rules: *step extension* and *jump extension*.

Step extension for distinguishable states w, s with $w \prec s$ and $|\text{cover}(s)| > 1$: if $Tab[w] = true$ and

(S1) each connected component of $G[\text{bag}(s)]$ contains a vertex from $\text{bag}(w) \cup \mathcal{I}$,

(S2) $\text{border}(w) \subseteq \text{bag}(s)$,

(S3) $\text{cover}(s) = \text{cover}(w) \cup \text{bag}(s)$,

(S4) $\text{bag}(s) \cap \text{cover}(w) \subseteq \text{bag}(w)$,

then set $Tab[s]$ to *true*.

Jump extension for distinguishable states $w = (X, \{B_S\}_{S \subseteq X}, \{f^{B_S}\}_S)$ and $s = (X, \{B_S\}_{S \subseteq X}, \{g^{B_S}\}_S)$ with $w \prec s$ and $|\text{cover}(s)| > 1$: if $Tab[w] = true$ and there exists a bottleneck set $S' \subseteq X$, such that

(J1) $f^{B_{S'}}(H) = 0$ and $g^{B_{S'}}(H) = 1$ for every $H \in \mathcal{B}(S') \setminus B_{S'}$,

(J2) $f^{B_{S'}}(H) = g^{B_{S'}}(H)$ for every $H \in B_{S'}$,

(J3) $f^{B_S}(H) = g^{B_S}(H)$ for every non-empty $S \neq S'$, $S \subseteq X$, and $H \in \mathcal{B}(S)$,

(J4) for each $H \in \mathcal{B}(S') \setminus B_{S'}$ there exists an $((N_G(S') \cap V(H)) \cup \mathcal{I})$-connected path decomposition $\mathcal{P}_H$ of H of width at most $k - |X| - 1$,

then set $Tab[s]$ to *true*.

Let us present some intuitions behind these extension rules. In step extension, if w corresponds to some $\mathcal{I}$-connected partial path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_l = \text{bag}(w))$, then s corresponds to an $\mathcal{I}$-connected partial path decomposition $\mathcal{P}' = (X_1, X_2, \ldots, X_l, X_{l+1} = \text{bag}(s))$ (we extend $\mathcal{P}$ by adding a single bag, namely $\text{bag}(s)$). Also note, that each new vertex in

X_{l+1}, i.e., one that is not in X_l, is (due to (S1)) connected by a path consisting of vertices from X_{l+1} to a vertex in X_l or $\mathcal{I}$, as required in $\mathcal{I}$-connected path decompositions.

In jump extension, if the state w corresponds to some $\mathcal{I}$-connected partial path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_l)$, then the state s corresponds to an $\mathcal{I}$-connected partial path decomposition $\mathcal{P}' = (X_1, X_2, \ldots, X_l, X_{l+1}, X_{l+2}, \ldots, X_{l+l'})$, where

- $(\bigcap_{i=l}^{l+l'} X_i) = X_l = X_{l+l'}$,
- $\mathcal{P}'' := (X_{l+1} \setminus X_l, X_{l+2} \setminus X_l, \ldots, X_{l+l'} \setminus X_l)$ is a (not necessarily connected) path decomposition of the graph induced by some S'-branches, for some $S' \subseteq X_l$. These are the S'-branches H in (J1) and $\mathcal{P}''$ is obtained by 'concatenating' the path decompositions from (J4).

Note that although a path decomposition $\mathcal{P}_H$ in (J4) may not be connected, we ensure (by definition of $\mathcal{I}$-connectivity) that each connected component of the subgraph induced by each prefix of $\mathcal{P}_H$ has a vertex from $N_G(S') \cap V(H)$ or from $\mathcal{I}$. Hence it contains a neighbor of S' or a vertex from $\mathcal{I}$, ensuring the required $\mathcal{I}$-connectivity of the resulting path decomposition. Figure 4.1 presents an example of step extension from state s to state w, i.e., these states fulfill the conditions of the step extension.

4.3.3 Summing Up

Let us recall how the algorithm works. We introduce a Boolean table *Tab*, indexed by all states. We initialize *Tab* by setting $Tab[s] = true$ for every state s, such that $\text{cover}(s) = \text{bag}(s) = \{v\}$, for some $v \in \mathcal{I}$, while for the remaining states s we initialize $Tab[s]$ to be $false$.

Then we process the states with respect to the ordering $\prec$, checking whether a step extension or a jump extension can be applied to set $Tab[s] = true$. We terminate when we find a state corresponding to a feasible solution (i.e., when we set $Tab[s] = true$ for some state s with $\text{cover}(s) = V(G)$), or when we have processed all states. In the latter case we report that a solution does not exist.

We note that the algorithm will be subsequently switching some entries of *Tab* from $false$ to $true$, and hence until the completion of the algorithm it is understood that the value $false$ of a particular entry of *Tab* does not provide any information as to whether a corresponding path decomposition exists.

4.4 The Analysis

Let us start by introducing some more definitions and additional notation. Let $\mathcal{P} = (X_1, X_2, \ldots, X_l)$ be a path decomposition of G. We say that a connected subgraph H of G is *contained in* an interval $[i, j]$ of $\mathcal{P}$ for some $1 \leq i \leq j \leq l$, if $V(H) \cap X_t \neq \emptyset$ if and only if $t \in \{i, \ldots, j\}$. Note that this definition is valid since it follows that for a connected subgraph H, the subset of indices t such that $V(H) \cap X_t \neq \emptyset$ is indeed an interval. If H is contained in $[i, j]$, then we denote these endpoints of the interval as $i = \alpha(H, \mathcal{P})$ and $j = \beta(H, \mathcal{P})$. If $\mathcal{P}$ is clear from the context, we will often write shortly $\alpha(H) := \alpha(H, \mathcal{P})$ and $\beta(H) := \beta(H, \mathcal{P})$.

For an $\mathcal{I}$-connected path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_l)$ and a set S, we say that an S-branch H is an *in-branch* if $S \subseteq X_{\alpha(H)}$ and $S \subseteq X_{\beta(H)}$. The lemma below gives us a lower bound on the number of in-branches of S.

Lemma 4.4.1. *For every set S and a path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_l)$, at most $2k$ S-branches are not in-branches.*

Proof. Consider an S-branch H, which is not an in-branch. This means that $S \not\subseteq X_{\alpha(H)}$ or $S \not\subseteq X_{\beta(H)}$.

First, consider H, such that $S \not\subseteq X_{\alpha(H)}$. Let t be the minimum index such that $S \subseteq X_1 \cup X_2 \cup \cdots \cup X_t$. Let $v \in S$ be a vertex in $X_t \setminus (X_1 \cup X_2 \cup \cdots \cup X_{t-1})$, it exists by the definition of t. Recall that v is a neighbor of some vertex w of H, so, since $\mathcal{P}$ is a path decomposition, there must be a bag X_i containing both v and w. Since X_t is the first bag, where v appears, we observe that $i \geq t$ and thus $\beta(H) \geq t$.

Suppose now that $\alpha(H) > t$. Note that since $S \not\subseteq X_{\alpha(H)}$, there is some $u \in S \setminus X_{\alpha(H)}$. Recall that u is a neighbor of some vertex from H. However, since $u \in X_1 \cup X_2 \cup \cdots \cup X_t$ and $u \notin X_{\alpha(H)}$, the vertex u does no appear in any bag containing a vertex of H, so $\mathcal{P}$ cannot be a path decomposition of G. Thus $\alpha(H) \leq t$.

Therefore, the bag X_t contains at least one vertex from H. Since S-branches are vertex-disjoint and $|X_t| \leq k$, we observe that there are at most k S-branches H, such that $S \not\subseteq X_{\alpha(H)}$.

Now consider an S-branch H, such that $S \subseteq X_{\alpha(H)}$ and $S \not\subseteq X_{\beta(H)}$. Let t' be the maximum index such that $S \subseteq X_{t'} \cup \cdots \cup X_l$.

Analogously to the previous case, we observe that $\alpha(H) \leq t'$ (by the maximality of t') and $\beta(H) \geq t'$, because $S \not\subseteq X_{\beta(H)}$. Thus the bag $X_{t'}$ contains at least one vertex from H, which shows that the number of S-branches H, such that $S \subseteq X_{\alpha(H)}$ and $S \not\subseteq X_{\beta(H)}$, is at most k. Therefore the

total number of S-branches, that are not in-branches is at most $2k$, which completes the proof. □

Recall that a non-empty set S is a bottleneck if $|\mathcal{B}(S)| > 2k$. Thus Lemma 4.4.1 implies the following:

Corollary 4.4.1. *If $\mathcal{P} = (X_1, \ldots, X_l)$ is a path decomposition and S is a bottleneck, then the following properties hold:*

A. S has at least one in-branch,

B. there is i, such that $S \subseteq X_i$,

C. $|S| \leq k$. □

These properties justify the following definition.

Definition 4.4.1. *For a bottleneck $S \subseteq V(G)$, let $t_1(S, \mathcal{P})$ (respectively $t_2(S, \mathcal{P})$) be the minimum (respectively maximum) index i such that $i = \alpha(H, \mathcal{P})$ ($i = \beta(H, \mathcal{P})$, respectively) for some S-branch H (respectively H), which is an in-branch.*

Note that the definition of an in-branch implies that $S \subseteq X_{t_1(S,\mathcal{P})} \cap X_{t_2(S,\mathcal{P})}$. The interval $I(S, \mathcal{P}) = [t_1(S, \mathcal{P}), t_2(S, \mathcal{P})]$ is called the *interval of S*. Again, we will often write shortly $t_1(S)$, $t_2(S)$, and $I(S)$, if $\mathcal{P}$ is clear from the context.

For a bottleneck S we can refine the classification of S-branches, which are not in-branches. We say that an S-branch H, which is not an in-branch, is

- a *pre-branch* if $\alpha(H) < t_1(S)$,
- a *post-branch* if $\alpha(H) \geq t_1(S)$ and $\beta(H) > t_2(S)$.

By $\mathcal{B}^x(S, \mathcal{P})$ we denote the set of all x-branches for S, where $x \in \{\text{pre}, \text{in}, \text{post}\}$. Again, if $\mathcal{P}$ is clear from the context, we will write $\mathcal{B}^x(S)$ instead of $\mathcal{B}^x(S, \mathcal{P})$. By $\mathcal{C}(S)$ we denote the set of all connected components of $G - S$, that are not S-branches.

Example. *The graph G in Figure 4.2 illustrates the above concepts. The sequence $X_1, X_2, \ldots, X_{16}$ forms a connected path decomposition of G. The only bottleneck set S consists of two vertices denoted by circles. (In this example we take $k = 3$.) There are 13 S-branches: one pre-branch ($G[X_1 \cup X_2 \setminus S]$), eleven in-branches ($G[X_i \setminus S]$ for $i \in \{4, 5, \ldots, 14\}$), and one post-branch ($G[(X_{15} \cup X_{16}) \setminus S]$). The interval of S is equal to $I(S, \mathcal{P}) = [4, 14]$ and the component*

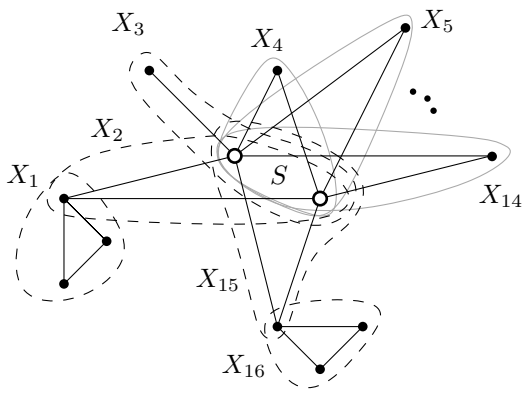


FIGURE 4.2: An illustration of a two-vertex bottleneck set S and the corresponding S-branches.

$G[X_3 \setminus S]$ is not an S-branch, because some vertices in S do not have a neighbor in this component.

For a subgraph H of G, we say that H *waits in* an interval $[i, j]$ of $\mathcal{P}$ if

$$V(H) \cap X_i = V(H) \cap X_{i+1} = \cdots = V(H) \cap X_j.$$

We say that a path decomposition $\mathcal{P}$ is *S-structured* if each subgraph C that is in $\mathcal{C}(S)$ or is a post- or a pre-branch of $\mathcal{S}$ waits in $I(S, \mathcal{P})$. The main technical tool in our approach is the following result concerning the structure of $\mathcal{I}$-connected path decompositions. The proof of the lemma is provided after giving several technical facts that we need.

Lemma 4.4.2. *If there exists an $\mathcal{I}$-connected path decomposition $\mathcal{P}$, then there is also an $\mathcal{I}$-connected path decomposition $\mathcal{P}'$ of width at most* $\mathrm{width}(\mathcal{P})$ *such that $\mathcal{P}'$ is S-structured for every bottleneck S.*

We define $c_{\min}(S, \mathcal{P})$ as the minimum index $i \in I(S, \mathcal{P})$, for which the size of the set

$$X_i \cap \left(\bigcup_{H \in \mathcal{B}^{\mathrm{pre}}(S) \cup \mathcal{B}^{\mathrm{post}}(S) \cup \mathcal{C}(S)} V(H) \right)$$

is minimum. Also, set $X^* := \left(\bigcup_{H \in \mathcal{B}^{\mathrm{pre}}(S) \cup \mathcal{B}^{\mathrm{post}}(S) \cup \mathcal{C}(S)} V(H) \right) \cap X_{c_{\min}(S,\mathcal{P})}$ to be this minimum-size set.

Let Γ be the set of all $\mathcal{I}$-connected path decompositions of G and $\mathcal{S}$ be the set of all bottlenecks of G. We will define a function $F : \mathcal{S} \times \Gamma \to \Gamma$,

which for given $S \in \mathcal{S}$ and $\mathcal{P} \in \Gamma$ transforms $\mathcal{P}$ into an S-structured $\mathcal{I}$-connected path decomposition of width at most width$(\mathcal{P})$. For simplicity of notation, from now on $t_1 = t_1(S,\mathcal{P})$, $t_2 = t_2(S,\mathcal{P})$, and $c_{\min} = c_{\min}(S,\mathcal{P})$, whenever S and $\mathcal{P}$ are clear from the context. Let $\mathcal{B}^{\text{in}}(S,\mathcal{P}) = \{H_1,\ldots,H_{l_{\text{in}}}\}$, where the in-branches are ordered according to their first occurrences in $\mathcal{P}$, that is, $\alpha(H_1) \leq \cdots \leq \alpha(H_{l_{\text{in}}})$. Let

$$d = \sum_{H \in \mathcal{B}^{\text{in}}(S)} (\beta(H) - \alpha(H) + 1)\,.$$

In other words, d is the sum of lengths of intervals in which the in-branches H are contained in the path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_l)$.

For each in-branch H_i, $i \in \{1,\ldots,l_{\text{in}}\}$, we define the following sequence:

$$\mathcal{P}_i := \Big(X_{\alpha(H_i)} \cap V(H_i), \ldots, X_{\beta(H_i)} \cap V(H_i)\Big)\,.$$

Since $\mathcal{P}$ is an $\mathcal{I}$-connected path decomposition of G, it is straightforward to observe that $\mathcal{P}_i$ is a $(N_G(S) \cap V(H_i)) \cup \mathcal{I}$-connected path decomposition of H_i. We will denote the elements of $\mathcal{P}_i$ by

$$\mathcal{P}_i = \Big(X^i_1, \ldots, X^i_{\beta(H_i)-\alpha(H_i)+1}\Big)\,.$$

Then, let us define a sequence $\mathcal{P}^*$ as follows:

$$\mathcal{P}^* := \bigcirc_{i=1}^{l_{\text{in}}} \mathcal{P}_i,$$

where $\bigcirc$ denotes the concatenation of sequences. Observe that the length of $\mathcal{P}^*$ is exactly d. We will denote the elements of $\mathcal{P}^*$ by $\mathcal{P}^* = \big(X^*_1, \ldots, X^*_d\big)$.

Define B^{in} to be the set of vertices of the in-branches of S, i.e., $B^{\text{in}} := \bigcup_{H \in \mathcal{B}^{\text{in}}(S,\mathcal{P})} V(H)$. Now, we define a path decomposition $F(S,\mathcal{P}) = \big(X'_1, \ldots, X'_{l+d+1}\big)$ as follows (see Figure 4.3):

$$X'_i = \begin{cases} X_i & \text{for } i \in \{1,\ldots,t_1-1\}; & (4.1)\\ X_i \setminus B^{\text{in}} & \text{for } i \in \{t_1,\ldots,c_{\min}-1\}; & (4.2)\\ X_{c_{\min}} \setminus B^{\text{in}} & \text{for } i = c_{\min}; & (4.3)\\ X^* \cup X^*_{i-c_{\min}} \cup S & \text{for } i \in \{c_{\min}+1,\ldots,c_{\min}+d\}; & (4.4)\\ X_{c_{\min}} \setminus B^{\text{in}} & \text{for } i = c_{\min}+d+1; & (4.5)\\ X_{i-d-1} \setminus B^{\text{in}} & \text{for } i \in \{c_{\min}+d+2,\ldots,t_2+d+1\}; & (4.6)\\ X_{i-d-1} & i \in \{t_2+d+2,\ldots,l+d+1\}. & (4.7) \end{cases}$$

Observe that $F(S,\mathcal{P})$ is obtained from $\mathcal{P}$ by a modification of the interval

$[t_1, t_2]$ of $\mathcal{P}$. Indeed, we notice that the prefix $(X_1, X_2, \ldots, X_{t_1-1})$ and the suffix $(X_{t_2+1}, X_{t_2+2}, \ldots, X_l)$ of $\mathcal{P}$ are just copied into $F(S, \mathcal{P})$ without any changes (see conditions (4.1) and (4.7)). All components of $G - S$, apart from the in-branches, are covered by bags on positions up to $c_{\min}$, see (4.2) and (4.3), and after position $c_{\min} + d$, see (4.6), and they wait for $d + 1$ steps in the interval $[c_{\min} + 1, c_{\min} + d + 1]$ of $F(S, \mathcal{P})$ — see (4.4)–(4.5). The interval $[c_{\min} + 1, c_{\min} + d]$ is used in (4.4) to cover all in-branches, one by one, in the order of their appearance in $\mathcal{P}$. Intuitively, two bags without any vertices of in-branches are present on positions $c_{\min}$ and $c_{\min} + d + 1$ to ensure that for any other bottleneck S', such that $I(S, \mathcal{P}) \subsetneq I(S', \mathcal{P})$, we have that $c_{\min}(S', \mathcal{P}) \notin I(S, \mathcal{P})$, see (4.3) and (4.5). (The appropriate details are in the proofs.) Figure 4.3 illustrates the conversion from $\mathcal{P}$ to $\mathcal{P}' := F(S, \mathcal{P})$ for a bottleneck S. Notice that the interval of S in $\mathcal{P}'$ is given by $t_1(S, \mathcal{P}') = c_{\min} + 1$ and $t_2(S, \mathcal{P}') = c_{\min} + d$, and it is possibly different than $[t_1, t_2]$. In the next lemma we show that $F(S, \mathcal{P})$ has all necessary properties.

Lemma 4.4.3. *For any $\mathcal{I}$-connected path decomposition $\mathcal{P}$ and a bottleneck S, $\mathcal{P}' = F(S, \mathcal{P})$ is an $\mathcal{I}$-connected path decomposition with* $\mathrm{width}(\mathcal{P}') \leq \mathrm{width}(\mathcal{P})$.

Proof. Let us recall the notation: $\mathcal{P} := (X_1, \ldots, X_l)$, $\mathcal{P}' = F(S, \mathcal{P}) = (X'_1, \ldots, X'_{l'})$, $t_1 = t_1(S, \mathcal{P})$ and $t_2 = t_2(S, \mathcal{P})$, $B^{\text{in}} := \bigcup_{H \in \mathcal{B}^{\text{in}}(S,\mathcal{P})} V(H)$. Moreover, define

$$B^{\text{not-in}} := \left(\bigcup_{H \in \mathcal{B}^{\text{pre}}(S,\mathcal{P}) \cup \mathcal{B}^{\text{post}}(S,\mathcal{P}) \cup \mathcal{C}(S)} V(H) \right).$$

Also, recall that $X'_i = X_i$, $1 \leq i < t_1$ and $X'_i = X_{i-d-1}$, for all $t_2 + d + 2 \leq i \leq l'$.

First, we want to show that $\mathcal{P}'$ satisfies the conditions in Definition 4.2.1.

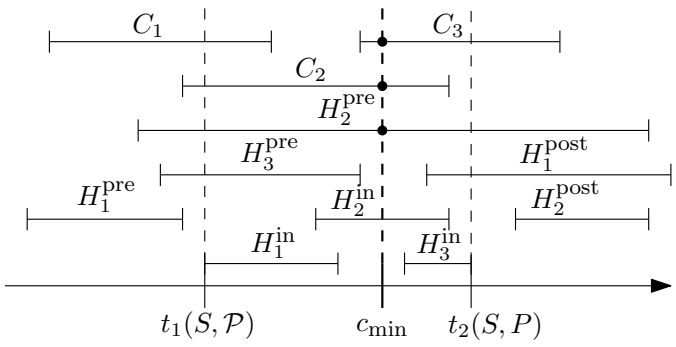
Let $\{u, v\}$ be an edge of G. Since $\mathcal{P}$ is a path decomposition, $u, v \in X_i$ for some i. If $i < t_1$, then $u, v \in X'_i = X_i$. If $i > t_2$, then $u, v \in X'_{i+d+1} = X_i$.

So suppose that $u, v \in X_i$ for $i \in [t_1, t_2]$. Note that this means that $u, v \in B^{\text{in}} \cup S$ or $u, v \in B^{\text{not-in}} \cup S$. If $u, v \in B^{\text{not-in}} \cup S$, then we have

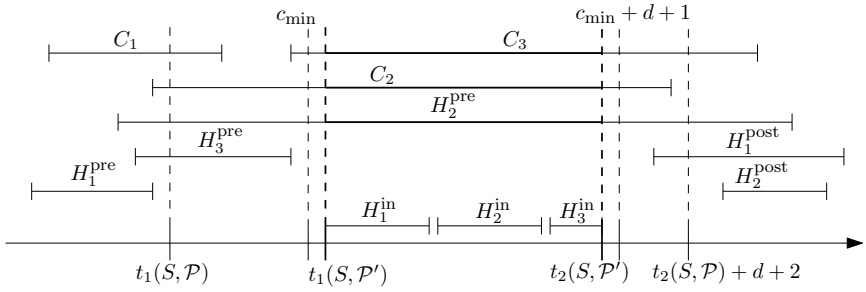
(i) $u, v \in X'_i$, if $i \leq c_{\min}$;

(ii) $u, v \in X'_{i+d+1}$, otherwise.

Finally, consider the case that, say, $u \in B^{\text{in}}$ and $v \in B^{\text{in}} \cup S$ (if both $u, v \in S$, then we are at the previous case). This means that u is a vertex of



(A) S-branches for a path decomposition $\mathcal{P}$ before transformation.



(B) New bags are inserted into $\mathcal{P}$ in which all components apart from in-branches wait in the interval of S in $\mathcal{P}'$, i.e., $C_2, C_3, H_2^{\text{pre}}$ wait in $[t_1(S,\mathcal{P}'), t_2(S,\mathcal{P}')]$.

FIGURE 4.3: Illustration of the conversion from $\mathcal{P}$ to $\mathcal{P}' = F(S,\mathcal{P})$ for a bottleneck S. For simplification, S is assumed to only have 8 S-branches and 3 components, which are denoted by letters H and C, respectively, with appropriate indices.

some in-branch $H_s \in \mathcal{B}^{\text{in}}(S,\mathcal{P})$, so it appears in some bag of $\mathcal{P}_s$ and thus of $\mathcal{P}^*$. This implies that $u, v \in X'_j$ for some $j \in [c_{\min}+1, c_{\min}+d]$. This implies that $\mathcal{P}'$ satisfies conditions A and B from Definition 4.2.1.

Now let us verify that the condition C is also satisfied, i.e., for every vertex v and indices $i < s < j$, such that $v \in X'_i \cap X'_j$ we have $v \in X'_s$. Clearly the condition is satisfied for every v and $j < t_1$ or $i > t_2$, since these parts of $\mathcal{P}'$ are just copied from $\mathcal{P}$ without any modifications. The situation is very similar if $i \leq t_2$ and $j \geq t_1$ and $v \in \left(S \cup B^{\text{not-in}}\right) \setminus X^*$. If $v \in X^*$, then v is included in all bags X'_s for $s \in [c_{\min}, c_{\min}+d+1]$, so the condition C follows from the correctness of $\mathcal{P}$. Finally, if v is a vertex of some $H_a \in \mathcal{B}^{\text{in}}(S,\mathcal{P})$, then the condition C follows from the property C that holds for the path decomposition P_a.

We now show that $\text{width}(\mathcal{P}') \leq \text{width}(\mathcal{P})$. Let $c_1 = |X^*|$ and $c_2 = \max_{i=t_1,\ldots,t_2} \left|B^{\text{in}} \cap X_i\right|$, and let $k = \text{width}(\mathcal{P}) + 1 = \max_{i=1,\ldots,l} |X_i|$. Observe that

each X'_i for $i \notin \{t_1, \ldots, t_2 + d + 1\}$ is an exact copy of some X_j, so $|X'_i| \leq k$. Moreover, each X'_i for $i \in \{t_1, \ldots, c_{\min}\} \cup \{c_{\min} + d + 1, \ldots, t_2 + d + 1\}$ was obtained from some X_j by removal of vertices of B^{in}, so again we have $|X_i| \leq k$. Finally, for $i \in \{c_{\min} + 1, \ldots, c_{\min} + d\}$ we have

$$\left|X'_i\right| = |S| + |X^*| + \left|X^*_{i-c_{\min}}\right| \leq |S| + c_1 + c_2.$$

However, by the definition of $\mathcal{P}^*$, we observe that $|S| + c_1 + c_2 \leq \left|X_j\right|$ for some $j \in \{t_1, \ldots, t_2\}$. Therefore, $\text{width}(\mathcal{P}') \leq \text{width}(\mathcal{P})$.

Finally, recall that $\mathcal{P}$ is $\mathcal{I}$-connected. Consider any connected component H of the subgraph $G[X'_1 \cup \cdots \cup X'_i]$ for an $i \in \{1, \ldots, l'\}$. We argue that H has a vertex in $X'_{\alpha(H,\mathcal{P}')} \cap \mathcal{I}$ as required in an $\mathcal{I}$-connected path decomposition. Denote $j' = \alpha(H, \mathcal{P}')$. We consider a few cases following the definition of $\mathcal{P}'$ in (4.1)-(4.7).

Suppose first that $j' < t_1$ or $j' \geq t_2 + d + 2$. Denote $j = j'$ when $j' < t_1$ and $j = j' - d - 1$ when $j' \geq t_2 + d + 2$. Then, $X'_1 \cup \cdots \cup X'_{j'} = X_1 \cup \cdots \cup X_j$ and $X'_{j'} = X_j$. Thus, for such j',

$$X'_{\alpha(H,\mathcal{P}')} \cap V(H) = X'_{j'} \cap V(H) = X_j \cap V(H) = X_{\alpha(H,\mathcal{P})} \cap V(H).$$

Since $\mathcal{P}$ is $\mathcal{I}$-connected, $X_{\alpha(H,\mathcal{P})} \cap V(H) \cap \mathcal{I} \neq \emptyset$ which completes the proof for this choice of j'.

Suppose now that $t_1 \leq j' < t_2 + d + 2$. Denote for brevity $Z = X'_{j'} \cap V(H)$. Since each bag of $\mathcal{P}$ introduces at most one new vertex, Z has one vertex, call it v. Note that v is not adjacent to a vertex in S. Otherwise the fact that $S \subseteq X'_{j'}$ would imply that v and also its neighbor in S belong to H, which would contradict $|Z| = 1$. Also, $v \notin S$ because $S \subseteq X'_{t_1-1}$ and $v \notin X'_{t_1-1}$. Thus, informally speaking, we have proved that that H is a connected component in $G - S$ that starts in $\mathcal{P}'$ with the vertex v and this vertex is not adjacent to any vertex in S. Note that the path decompositions $\mathcal{P}$ and $\mathcal{P}'$ when restricted to H are identical, $(X_{\alpha(H,\mathcal{P})} \cap V(H), \ldots, X_{\beta(H,\mathcal{P})} \cap V(H)) = (X'_{\alpha(H,\mathcal{P}')} \cap V(H), \ldots, X'_{\beta(H,\mathcal{P}')} \cap V(H))$. This implies in particular that $G[X_1 \cup \cdots \cup X_{\alpha(H,\mathcal{P})}]$ also has a connected component that consists of only the vertex v. Since, $\mathcal{P}$ is $\mathcal{I}$-connected, $v \in \mathcal{I}$. This implies that H has a vertex in $X'_{\alpha(H,\mathcal{P}')} \cap \mathcal{I}$, i.e., this set consists of v. □

Observe that every connected component H of $G - S$ is either contained in $I(S, \mathcal{P}')$ (which means that H is an in-branch) or waits in $I(S, \mathcal{P}')$ (for all other H).

Observation 4.4.1. *For any $\mathcal{P}$ and a bottleneck S, the path decomposition $F(S, \mathcal{P})$ is S-structured.*

Now we want to define a series of transformations, which start with an arbitrary $\mathcal{I}$-connected path decomposition $\mathcal{P}$ and transform it into an $\mathcal{I}$-connected path decomposition with no larger width, which is S-structured for every $S \in \mathcal{S}$. For this, we will apply the F-transformations for all bottlenecks. In order to do this we need some technical lemmas about the structure of bottlenecks and their branches.

Lemma 4.4.4. *Let S and S' be two bottlenecks, such that $S' \subsetneq S$. There exists an S'-branch H such that $\bigcup_{H' \in \mathcal{B}(S)} V(H') \cup S \setminus S' \subseteq V(H)$ and every S'-branch H', different than H, is a non-branch connected component of $G - S$.*

Proof. Let $S, S' \in \mathcal{S}$ such that $S' \subsetneq S$. Clearly S intersects some connected component of $G - S'$. Since every S-branch H'' is connected and $S \setminus S' \subset N_G(V(H''))$, we observe that two connected components of $G - S'$ can not be distinctive, i.e., there exists a connected component H of $G - S'$ such that $\bigcup_{H' \in \mathcal{B}(S)} V(H') \cup S \setminus S' \subseteq V(H)$.

To see that H is an S'-branch, consider a vertex $s' \in S'$. By assumption, $S' \subseteq S$ and hence s' is also a vertex of S. Thus, s' has a neighbor in every S-branch H', and thus in H. See Figure 4.4 for an illustration.

Now consider an S'-branch $H'' \neq H$. If every vertex of $S \setminus S'$ is adjacent to some vertex of H'', then H'' is an S-branch, a contradiction. Thus, H'' is a connected component of $G - S$, which is not an S-branch. □

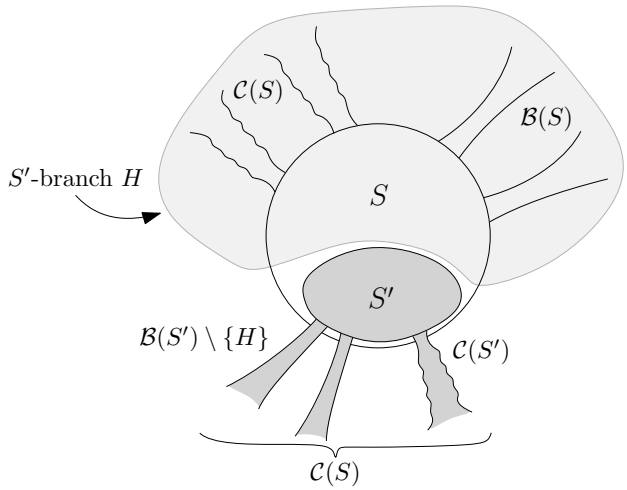


FIGURE 4.4: An illustration for Lemma 4.4.4. S and S' are bottlenecks, such that $S' \subsetneq S$. All S-branches are subgraphs of an S'-branch H and every S'-branch, different than H, is a connected component of $G - S$, which is not an S-branch.

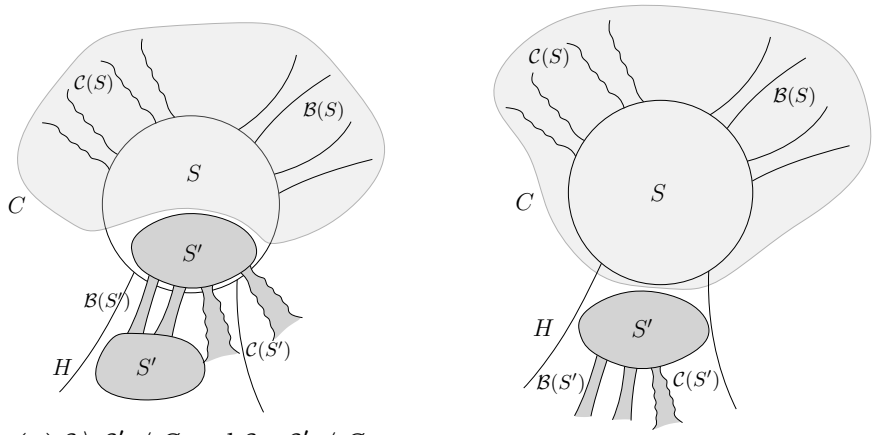
Lemma 4.4.5. *For any two bottlenecks S and S', if $S' \not\subseteq S$, then there exists exactly one connected component H of $G - S$ such that $S' \subseteq S \cup V(H)$.*

Proof. Let $S, S' \in \mathcal{S}$ such that $S' \not\subseteq S$. Clearly S' intersects some connected component of $G - S$. Suppose that S' has a non-empty intersection with two connected components H and H' of $G - S$. Thus, since every S'-branch H'' is connected and $N_G(V(H'')) = S'$, we observe that H'' contains a vertex of S (otherwise H, H' would not be two distinct components). Since S'-branches are vertex-disjoint, this implies that the number of such S'-branches is at most $|S|$, which is in turn at most k by Corollary 4.4.1. However, this contradicts the assumption that S' is a bottleneck. □

The next remark is a straightforward consequence of Lemma 4.4.4 and Lemma 4.4.5.

Remark 4.4.1. *Let S and S' be bottlenecks such that $S' \not\subseteq S$ and $S \not\subseteq S'$. Let H be the connected component of $G - S$, such that $S' \subseteq S \cup V(H)$. There exists exactly one connected component C of $G - S'$ such that $\bigcup_{H' \in \mathcal{B}(S) \setminus H} V(H') \cup S \setminus S' \subseteq V(C)$. Moreover, all S'-branches but possibly C are subgraphs of H.*

See Figure 4.5 for the illustration for Lemma 4.4.5 and Remark 4.4.1.



(A) $S \setminus S' \neq \emptyset$ and $S \cap S' \neq \emptyset$.

(B) $S \cap S' = \emptyset$.

FIGURE 4.5: An illustration of two cases in Lemma 4.4.5 and Remark 4.4.1; S and S' are bottlenecks. All S-branches (apart from H, which is a connected component of $G - S$, which may or may be not an S-branch) are subgraphs of C and $S' \subseteq S \cup V(H)$.

We say two bottlenecks S and S' are *well-nested* in $\mathcal{P}$ if

(i) $I(S', \mathcal{P}) \subsetneq I(S, \mathcal{P})$ or

(ii) $I(S,\mathcal{P}) \subseteq I(S',\mathcal{P})$ or

(iii) $I(S,\mathcal{P}) \cap I(S',\mathcal{P}) = \emptyset$.

Observe that the ordering of S, S' in the definition above matters.

Lemma 4.4.6. *For any $\mathcal{I}$-connected path decomposition $\mathcal{P}$ and bottlenecks S, S', if $\mathcal{P}$ is S'-structured, then S and S' are well-nested in $\mathcal{P}$. Moreover, if $S \subsetneq S'$, then either $I(S,\mathcal{P}) \cap I(S',\mathcal{P}) = \emptyset$ or $I(S',\mathcal{P}) \subseteq I(S,\mathcal{P})$.*

Proof. Let $\mathcal{P} = (X_1, \ldots, X_l)$ be an $\mathcal{I}$-connected path decomposition of G and let $S, S' \in \mathcal{S}$. Suppose $\mathcal{P}$ is S'-structured. Let us assume that $I(S',\mathcal{P}) \cap I(S,\mathcal{P}) \neq \emptyset$, we will show that either $I(S',\mathcal{P}) \subsetneq I(S,\mathcal{P})$ or $I(S,\mathcal{P}) \subseteq I(S',\mathcal{P})$.

Case A: $S' \subsetneq S$**.** By Lemma 4.4.4, there exists an S'-branch H such that for every S-branch H' it holds that $S \setminus S' \cup V(H') \subseteq V(H)$, and thus $[\alpha(H'), \beta(H')] \subseteq [\alpha(H), \beta(H)]$. Recall that $t_1(S) = \alpha(H'_1)$ and $t_2(S) = \beta(H'_2)$ for some in-branches $H'_1, H'_2 \in \mathcal{B}(S)$, so $I(S,\mathcal{P}) = [t_1(S), t_2(S)] \subseteq [\alpha(H), \beta(H)]$. Consider two subcases.

Subcase A1: $\alpha(H) \in I(S',\mathcal{P})$**.** Since $\mathcal{P}$ is S'-structured, we observe that H is an in-branch for S', which implies that $I(S,\mathcal{P}) \subseteq [\alpha(H), \beta(H)] \subseteq I(S',\mathcal{P})$.

Subcase A2: $\alpha(H) \notin I(S',\mathcal{P})$**.** First, observe that if $\alpha(H) > t_2(S')$, then $t_1(S) > t_2(S')$ and thus $I(S,\mathcal{P}) \cap I(S',\mathcal{P}) = \emptyset$, which contradicts our assumption. Analogously, if $\beta(H) < t_1(S')$, we again obtain that $I(S,\mathcal{P}) \cap I(S',\mathcal{P}) = \emptyset$. Therefore assume that $\alpha(H) < t_1(S') \leq \beta(H)$. Observe that this implies that H is a pre-branch for S' and, since $\mathcal{P}$ is S'-structured, H waits in $I(S',\mathcal{P})$. In particular $(S \setminus S') \cap X_{t_1(S')} = (S \setminus S') \cap X_{t_1(S')+1} = \ldots = (S \setminus S') \cap X_{t_2(S')}$. Since $I(S',\mathcal{P}) \cap I(S,\mathcal{P}) \neq \emptyset$, it is necessary that $t_1(S) \leq t_1(S')$ (otherwise $t_1(S) > t_2(S')$). Thus, $t_1(S) \leq t_1(S') \leq t_2(S') \leq t_2(S)$ (recall H waits in $I(S',\mathcal{P})$). Summing up, if $S' \subsetneq S$, then either $I(S',\mathcal{P}) \subsetneq I(S,\mathcal{P})$ or $I(S,\mathcal{P}) \subseteq I(S',\mathcal{P})$, which completes the proof for this case.

Case B: $S' \not\subseteq S$**.** By Lemma 4.4.5, there exists exactly one connected component H of $G - S$ such that $S' \subseteq S \cup V(H)$. Since $S' \not\subseteq S$, we observe that $V(H) \cap S' \neq \emptyset$.

If H is an S-branch that is an in-branch in $\mathcal{P}$, then

$$I(S', \mathcal{P}) \subseteq [\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})] \subseteq I(S, \mathcal{P}).$$

We observe that $I(S', \mathcal{P}) \subseteq I(S, \mathcal{P})$ is equivalent to $I(S', \mathcal{P}) \subsetneq I(S, \mathcal{P})$ or $I(S', \mathcal{P}) = I(S, \mathcal{P})$, thus S and S' are well-nested in $\mathcal{P}$. So assume that H is a connected component of $G - S$, that is not an in-branch for S (it may still be a pre- or a post-branch). We consider now two subcases.

Subcase B1: $S \subsetneq S'$**.** By Lemma 4.4.4, all S-branches possibly except for H are not S'-branches and all S'-branches are subgraphs of H.

Because $\mathcal{P}$ is S'-structured, every S-branch but possibly H waits in $I(S', \mathcal{P})$. In particular, every in-branch H'' for S does wait in $I(S', \mathcal{P})$. Thus, for every such an in-branch H'' it holds that $I(S', \mathcal{P}) \subseteq [\alpha(H''), \beta(H'')]$ or $I(S', \mathcal{P}) \cap [\alpha(H''), \beta(H'')] = \emptyset$. Note that the second condition implies that $I(S', \mathcal{P}) \cap I(S, \mathcal{P}) = \emptyset$, which contradicts our assumption. Therefore we obtain that $I(S', \mathcal{P}) \subseteq [\alpha(H''), \beta(H'')] \subseteq I(S, \mathcal{P})$. Note that this shows the second claim of the lemma.

Subcase B2: $S \setminus S' \neq \emptyset$**.** Let C be the connected component of $G - S'$, for which it holds $\bigcup_{H' \in \mathcal{B}(S) \setminus H} V(H') \cup S \setminus S' \subseteq V(C)$, whose existence is guaranteed by Remark 4.4.1.

If C is an in-branch for S', we observe that $I(S, \mathcal{P}) \subseteq [\alpha(C), \beta(C)] \subseteq I(S', \mathcal{P})$, because $\mathcal{P}$ is S'-structured. On the other hand, if C is not an in-branch for S', then all in-branches of S wait in $I(S', \mathcal{P})$. This is because all subgraphs but in-branches of S' wait in $I(S', \mathcal{P})$, and every in-branch for S is vertex-disjoint with every in-branch for S', since they are all contained in $V(C) \cup S'$. Thus, since $I(S, \mathcal{P}) \cap I(S', \mathcal{P}) \neq \emptyset$, we conclude that $I(S', \mathcal{P}) \subseteq I(S, \mathcal{P})$, which completes the proof. □

In the next lemma, we show that we can apply a series of F-transformations, one for each bottleneck, so that the structure obtained in previous F-transformations is not 'destroyed' during the subsequent F-transformations.

Lemma 4.4.7. *Let S, S' be bottlenecks and let $\mathcal{P}$ be an S-structured $\mathcal{I}$-connected path decomposition. Let $\mathcal{P}' = F(S', \mathcal{P})$.*

A. *If $I(S', \mathcal{P}) \subseteq I(S, \mathcal{P})$, then $\mathcal{P}'$ is S-structured and $I(S', \mathcal{P}') \subseteq I(S, \mathcal{P}')$.*

B. *If $t_2(S, \mathcal{P}) < t_1(S', \mathcal{P})$, then $\mathcal{P}'$ is S-structured and $t_2(S, \mathcal{P}') < t_1(S', \mathcal{P}')$.*

C. *If* $t_2(S', \mathcal{P}) < t_1(S, \mathcal{P})$, *then* $\mathcal{P}'$ *is* S*-structured and* $t_2(S', \mathcal{P}') < t_1(S, \mathcal{P}')$.

Proof. First, let us prove point A, i.e. we assume that $I(S', \mathcal{P}) \subseteq I(S, \mathcal{P})$. If $S' = S$, then the results are obvious, so we consider two cases.

Case A: $S' \subsetneq S$**.** Observe that by Lemma 4.4.6 we obtain that $I(S', \mathcal{P}) \cap I(S, \mathcal{P}) = \emptyset$ or $I(S, \mathcal{P}) \subseteq I(S', \mathcal{P})$, which leads to the equality $I(S, \mathcal{P}) = I(S', \mathcal{P})$. There exists an S'-branch H, such that $\bigcup_{H' \in \mathcal{B}(S)} V(H') \cup S \setminus S' \subseteq V(H)$ (Lemma 4.4.4). Because $I(S, \mathcal{P}) = I(S', \mathcal{P})$, we conclude that H is the only in-branch for S'. The path decomposition $\mathcal{P}$ is S-structured, so every connected component of $G - S'$, different than H, waits in $I(S', \mathcal{P}) = I(S, \mathcal{P})$. Thus, the transformation $F(S', \mathcal{P})$ does not make any changes to bags $X_{t_1(S,\mathcal{P})}, X_{t_1(S,\mathcal{P})+1}, \ldots, X_{t_2(S,\mathcal{P})}$, i.e., $I(S, \mathcal{P}') = I(S', \mathcal{P}')$.

Case B: $S' \not\subseteq S$**.** By Lemma 4.4.5, there exists exactly one connected component H of $G - S$ such that $S' \subseteq S \cup V(H)$. The assumption $I(S', \mathcal{P}) \subseteq I(S, \mathcal{P})$ implies that some vertex of H appears for the first time in $\mathcal{P}$ in the interval $I(S, \mathcal{P})$, i.e, H does not wait in $I(S, \mathcal{P})$. Moreover, it implies that $t_1(S, \mathcal{P}) \leq \alpha(H)\mathcal{P} \leq \beta(H, \mathcal{P}) \leq t_2(S, \mathcal{P})$ due to our general assumption that each bag introduces at most one new vertex. This implies that H is an in-branch for S. We are going to show now that every S'-branch H', which is an in-branch, is a subgraph of H. If $S \subsetneq S'$, then we obtain it immediately from Lemma 4.4.4, so let $S \setminus S' \neq \emptyset$. Let C be a connected component of $G - S'$, such that $\bigcup_{H' \in \mathcal{B}(S) \setminus H} V(H') \cup S \setminus S' \subseteq V(C)$, whose existence is guaranteed by Remark 4.4.1. Recall that C might or might not be an S'-branch, but for sure it is not an in-branch for S', because then $I(S, \mathcal{P}) \subsetneq I(S', \mathcal{P})$, which is a contradiction. Thus, by Remark 4.4.1, every in-branch for S' is a subgraph of H.

Because every in-branch for S' is a subgraph of H, we conclude that $I(S', \mathcal{P}) \subseteq [\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})]$ and the only changes made by the transformation $F(S', \mathcal{P})$ concern the vertices of $H \in \mathcal{B}^{\text{in}}(S)$. Every connected component of $G - S$, apart from in-branches (for S'), waits in $I(S', \mathcal{P}')$, so $F(S', \mathcal{P})$ is S-structured and $I(S', \mathcal{P}') \subseteq [\alpha(H, \mathcal{P}'), \beta(H, \mathcal{P}')] \subseteq I(S, \mathcal{P}')$.

We notice that the prefix $(X_1, X_2, \ldots, X_{t_1(S',\mathcal{P})-1})$ and suffix $(X_{t_2(S',\mathcal{P})+1}, X_{t_2(S',\mathcal{P})+2}, \ldots, X_l)$ of $\mathcal{P}$ are just copied into $\mathcal{P}'$ without any changes. Thus, cases B and C hold as well. □

In the following lemma it is crucial that the path decomposition $\mathcal{P}$ is not only S-structured but has been obtained by applying the transformation described in (4.1)-(4.7) to $\mathcal{P}_0$. In particular, the bags added in (4.3) and (4.5)

will play a crucial role in ensuring that the path decomposition returned by $F(S', \mathcal{P})$ remains S-structured.

Lemma 4.4.8. *Let S, S' be bottlenecks and let $\mathcal{P}_0$ be any $\mathcal{I}$-connected path decomposition. Let $\mathcal{P} = F(S, \mathcal{P}_0)$ and $\mathcal{P}' = F(S', \mathcal{P})$. If $I(S, \mathcal{P}) \subsetneq I(S', \mathcal{P})$, then $\mathcal{P}'$ is S-structured and $I(S, \mathcal{P}') \subseteq I(S', \mathcal{P}')$ or $I(S, \mathcal{P}') \cap I(S', \mathcal{P}') = \emptyset$.*

Proof. Let $\mathcal{P} = (X_1, \ldots, X_l)$ be an $\mathcal{I}$-connected path decomposition of G, let $S, S' \in \mathcal{S}$ and $\mathcal{P}' = F(S', \mathcal{P})$. Moreover, assume that $\mathcal{P} = F(S, \mathcal{P}_0)$ for some $\mathcal{I}$-connected path decomposition $\mathcal{P}_0$. In particular, this implies that $\mathcal{P}$ is S-structured. Finally, assume that $I(S, \mathcal{P}) \subsetneq I(S', \mathcal{P})$.

Case A: $S' \subsetneq S$**.** By Lemma 4.4.4 there exists an S'-branch H such that $\bigcup_{H' \in \mathcal{B}(S)} V(H') \cup S \setminus S' \subseteq V(H)$. If H is an in-branch for S' then $I(S, \mathcal{P}) \subseteq [\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})] \subsetneq I(S', \mathcal{P})$. However, recall that F-transformation applied to S' and $\mathcal{P}$ does not change the structure of the bags restricted to H (or any other in-branch of S'). Therefore we conclude that $I(S, \mathcal{P}') \subseteq [\alpha(H, \mathcal{P}'), \beta(H, \mathcal{P}')] \subsetneq I(S', \mathcal{P}')$ and $\mathcal{P}'$ is S-structured.

Now assume that H is a pre-branch or a post-branch for S'. From the construction of $\mathcal{P}$ we have that $c_{\min}(S', \mathcal{P}) \notin I(S, \mathcal{P})$. Because every in-branch H' for S' either waits in $I(S, \mathcal{P})$ or $[\alpha(H', \mathcal{P}), \beta(H', \mathcal{P})] \cap I(S, \mathcal{P}) = \emptyset$, we obtain that $I(S, \mathcal{P}') \cap I(S', \mathcal{P}') = \emptyset$ and $\mathcal{P}'$ is S-structured.

Case B: $S' \not\subseteq S$ **and** $S \not\subseteq S'$**.** By Lemma 4.4.5 there exists exactly one connected component H of $G - S$ such that $S' \subseteq S \cup V(H)$. Because $V(H) \cap S' \neq \emptyset$ we have that $I(S', \mathcal{P}) \subseteq [\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})]$. We observe that H cannot be an in-branch for S, otherwise $I(S', \mathcal{P}) \subseteq [\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})] \subseteq I(S, \mathcal{P})$, which contradicts the assumption that $I(S, \mathcal{P}) \subsetneq I(S', \mathcal{P})$. Thus, $H \notin \mathcal{B}^{\text{in}}(S)$.

From the facts that $\mathcal{P}$ is S-structured and $I(S', \mathcal{P}) \subseteq [\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})]$, we observe that H waits in $I(S, \mathcal{P})$. By Remark 4.4.1 there exists a connected component C of $G - S'$ such that $\bigcup_{H' \in \mathcal{B}(S) \setminus H} V(H') \cup S \setminus S' \subseteq V(C)$. Because H is not an in-branch for S, all in-branches of S are subgraphs of C. If C is an in-branch for S' then $I(S, \mathcal{P}) \subseteq [\alpha(C, \mathcal{P}), \beta(C, \mathcal{P})] \subsetneq I(S', \mathcal{P})$. Because F-transformation does not change bags inside one in-branch, we have that $I(S, \mathcal{P}') \subseteq [\alpha(C, \mathcal{P}'), \beta(C, \mathcal{P}')] \subsetneq I(S', \mathcal{P}')$ and $\mathcal{P}'$ is S-structured. On the other hand, assume that C is not an in-branch for S'. From construction of $\mathcal{P}$ we have that $c_{\min}(S', \mathcal{P}) \notin I(S, \mathcal{P})$ Because every in-branch H' for S' either waits in $I(S, \mathcal{P})$ or $[\alpha(H', \mathcal{P}), \beta(H', \mathcal{P})] \cap I(S, \mathcal{P}) = \emptyset$, we obtain that $I(S, \mathcal{P}') \cap I(S', \mathcal{P}') = \emptyset$ and $\mathcal{P}'$ is S-structured.

Case C: $S \subsetneq S'$**.** From Lemma 4.4.4 there exists an S-branch H such that $\bigcup_{H' \in \mathcal{B}(S')} V(H') \cup S' \setminus S \subseteq V(H)$, i.e., $I(S', \mathcal{P}) \subseteq [\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})]$. We observe that H cannot be an in-branch for S, otherwise $I(S', \mathcal{P}) \subseteq [\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})] \subseteq I(S, \mathcal{P})$, which contradicts the assumption that $I(S, \mathcal{P}) \subsetneq I(S', \mathcal{P})$.

Let then H be a pre-branch or a post-branch for S, which means that H waits in $I(S, \mathcal{P})$ (notice that it is impossible that $[\alpha(H, \mathcal{P}), \beta(H, \mathcal{P})] \cap I(S, \mathcal{P}) = \emptyset$, because then $I(S', \mathcal{P}) \cap I(S, \mathcal{P}) = \emptyset$). From construction of $\mathcal{P}$ we have that $c_{\min}(S', \mathcal{P}) \notin I(S, \mathcal{P})$. Because every in-branch H' for S' either waits in $I(S, \mathcal{P})$ or $[\alpha(H', \mathcal{P}), \beta(H', \mathcal{P})] \cap I(S, \mathcal{P}) = \emptyset$, we obtain that $I(S, \mathcal{P}') \cap I(S', \mathcal{P}') = \emptyset$ and $\mathcal{P}'$ is S-structured. □

Now we are ready to prove Lemma 4.4.2.

Proof of Lemma 4.4.2. Let $\mathcal{P}$ be an $\mathcal{I}$-connected path decomposition. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_{n'}\}$ be the set of all bottlenecks. We define a path decomposition $\mathcal{P}' := \mathcal{P}_{n'}$ in the following recursive way:

$$\mathcal{P}_0 = \mathcal{P};$$
$$\mathcal{P}_i = F(S_i, \mathcal{P}_{i-1}), \text{ for } i \in \{1, \ldots, n'\}.$$

We are going to prove now that $\mathcal{P}_q$ is S_j-structured and S_i, S_j are well-nested in $\mathcal{P}_q$ for every $1 \leq j \leq q \leq n'$ and $1 \leq i \leq n'$.

Induction on q. If $q = 1$, then obviously $j = 1$. By Observation 4.4.1, $\mathcal{P}_q$ is S_q-structured. Thus, by Lemma 4.4.6, S_q and S_i are well-nested in $\mathcal{P}_q$ for every $1 \leq i \leq n'$. So assume that $q > 1$ and the claim holds for $q - 1$.

Let $j \in \{1, \ldots, q-1\}$. For every S_j we have, by the induction assumption, that S_q, S_j are well-nested in $\mathcal{P}_{q-1}$ and $\mathcal{P}_{q-1}$ is S_j-structured. By Lemma 4.4.8, if $I(S_j, \mathcal{P}_{q-1}) \subsetneq I(S_q, \mathcal{P}_{q-1})$, then $\mathcal{P}_q = F(S_q, \mathcal{P}_{q-1})$ is S_j-structured. By Lemma 4.4.7, if $I(S_q, \mathcal{P}_{q-1}) \subseteq I(S_j, \mathcal{P}_{q-1})$ or $I(S_j, \mathcal{P}_{q-1}) \cap I(S_q, \mathcal{P}_{q-1}) = \emptyset$, then $\mathcal{P}_q = F(S_q, \mathcal{P}_{q-1})$ is S_j-structured.

Because $\mathcal{P}_q$ is S_j-structured, then from Lemma 4.4.6 for any $i \in \{1, \ldots, n'\}$ we have that S_i, S_j are well-nested in $\mathcal{P}_q$.

So $\mathcal{P}'$ is S-structured for every bottleneck $S \in \mathcal{S}$. Note that Lemma 4.4.3 ensures that $\mathcal{P}'$ is an $\mathcal{I}$-connected path decomposition of width at most $\text{width}(\mathcal{P})$, which finishes the proof. □

Now we are ready to show the correctness of our algorithm. We will prove it in two steps.

Lemma 4.4.9. *If G has an $\mathcal{I}$-connected path decomposition of width at most $k-1$, then $Tab[s] = true$ for some state s such that* $\text{cover}(s) = V(G)$.

Proof. Suppose that G has an $\mathcal{I}$-connected path decomposition of width $k-1$. By Lemma 4.4.2, there exists an $\mathcal{I}$-connected path decomposition $\mathcal{P} = (X_1, \ldots, X_l)$ that has width $k-1$ and is S-structured for each bottleneck S.

By Lemma 4.4.6, the set $\mathcal{S}$ of all bottlenecks with relation $S \prec S'$ if and only if $I(S, \mathcal{P}) \subseteq I(S', \mathcal{P})$ forms a partial order (assuming that any ties, i.e. when $I(S, \mathcal{P}) = I(S', \mathcal{P})$, are resolved arbitrarily). Let $S_1, \ldots, S_t$ be the maximal elements with respect to this partial order. Note that for $i \neq j$, $I(S_i, \mathcal{P}) \cap I(S_j, \mathcal{P}) = \emptyset$ and for any bottleneck $S' \notin \{S_1, \ldots, S_t\}$ we have $I(S', \mathcal{P}) \subseteq I(S_i, \mathcal{P})$ for some $i \in \{1, \ldots, t\}$. Assume without loss of generality that the 'maximal' bottlenecks are ordered according to the left endpoints of their intervals,

$$t_1(S_1) \leq t_1(S_2) \leq \cdots \leq t_1(S_t).$$

We show how to arrive at the desired state s. To that end we argue, by induction on j, that for each

$$j \in J := \{1, \ldots, l\} \setminus \bigcup_{i=1}^{t} \{t_1(S_i), \ldots, t_2(S_i) - 1\}$$

there exists a state s_j such that $\text{cover}(s_j) = G[X_1 \cup \cdots \cup X_j]$, $\text{bag}(s_j) = X_j$ and $Tab[s_j] = true$.

Since the first bag of $\mathcal{P}$ consists of a vertex in $\mathcal{I}$, this clearly holds for $j = 1$ so take $j > 1$ and assume that the claim is true for each $j' \in J \cap \{1, \ldots, j-1\}$. We consider two cases.

Case A: $j \notin I(S_i, \mathcal{P})$ for each $i \in \{1, \ldots, t\}$. Hence we have $j - 1 \in J$. This implies, according to Lemma 4.4.1, that for each bottleneck S, either there are at most $2k$ S-branches H such that $\alpha(H) \leq j$, or there are at most $2k$ S-branches such that $j \leq \beta(H)$. Thus, there exists a state s_j such that $\text{cover}(s_j) = G[X_1 \cup \cdots \cup X_j]$ and $\text{bag}(s_j) = X_j$. The step extension rule and $Tab[s_{j-1}] = true$, which holds by the induction hypothesis, imply $Tab[s_j] = true$ as required.

Case B: $j \in I(S_i, \mathcal{P})$ for some bottleneck S_i. By the definition of the set J, $j = t_2(S_i)$. Hence, the preceding index of j in the set J is $j' = t_1(S_i) - 1$. Again, by the definition of $I(S_i, \mathcal{P})$, Lemma 4.4.1, and the maximality of S_i with respect to the partial order, we have that for each bottleneck set S either at most $2k$ S-branches H satisfy $\alpha(H) \leq j'$, or at least $|\mathcal{B}(S_{i'})| - 2k$ S-branches H are contained in $[1, j']$, i.e., satisfy $\beta(H) \leq j'$, depending whether $t_2(S) \leq j'$ or $t_1(S) > j'$. Thus, there exists a state s_j such that

$\text{cover}(s_j) = G[X_1 \cup \cdots \cup X_j]$ and $\text{bag}(s_j) = X_j$. Consider the jump extension rule constructed for $S' = S_i$. For the set $B_{S'}$ in (J1) and (J2) take all S'-branches that are not in-branches, i.e., those that are covered in $[j'+1, j]$ in $\mathcal{P}$. Note that each S-branch of each bottleneck $S \neq S_i$ such that $S \subseteq X_{j'}$ waits in the interval $I(S_i, \mathcal{P})$ because $\mathcal{P}$ is S_i-structured, which ensures the condition (J3). Condition (J4) holds because the decomposition $\mathcal{P}_H$ in (J4) exists which is certified by the decomposition $\mathcal{P}$, namely $\mathcal{P}_H = (X_{\alpha(H)} \cap V(H), \ldots, X_{\beta(H)} \cap V(H))$. Thus, $Tab[j'] = true$ (which holds by the induction hypothesis) ensures that $Tab[j] = true$.

Finally observe that $l \in J$, $\text{cover}(s_l) = G[X_1 \cup \cdots \cup X_l] = V(G)$ and $Tab[s_l] = true$. Thus, $s = s_l$ is the required state. □

Lemma 4.4.10. *For any state s, if $Tab[s] = true$, then G_s has an $\mathcal{I}$-connected path decomposition of width at most $k-1$.*

Proof. Proof by induction on the position of s in the ordering $\prec$. First, let $\text{cover}(s) = \{v\}$ for some $v \in V(G)$ (notice that such states are smallest, according to $\prec$). If $v \in \mathcal{I}$, then $Tab[s]$ was set true in the initialization step. This is justified by considering connected path decomposition consisting of a single bag $\{v\}$, which is a proper connected path decomposition of the single-vertex graph $(\{v\}, \emptyset)$. On the other hand, if $v \notin \mathcal{I}$, then $Tab[s]$ is never set *true*, as the extension rules apply only to states with $|\text{cover}(s)| > 1$.

Now suppose that $|\text{cover}(s)| \geq 2$, and the Lemma holds for all states $w \prec s$. Since $Tab[s] = true$, its value must have been set by one of the extension rules. Consider two cases.

Case 1: $Tab[s]$ **was set by step extension.** Consider the state w. If $\text{bag}(s) \not\subseteq \text{bag}(w)$, then since by (S4), $\text{bag}(s) \cap \text{cover}(w) \subseteq \text{bag}(w)$, we have that $\text{bag}(s) \not\subseteq \text{cover}(w)$. Therefore, $\text{cover}(s) = \text{cover}(w) \cup \text{bag}(s)$ implies that $|\text{cover}(s)| > |\text{cover}(w)|$, which means that $w \prec s$. On the other hand, if $\text{bag}(s) \subseteq \text{bag}(w)$, we have $\text{cover}(s) = \text{cover}(w)$ due to (S3). However, since s and w, are distinguishable, we have $\text{bag}(s) \subsetneq \text{bag}(w)$, so $|\text{bag}(s)| < |\text{bag}(w)|$ and thus $w \prec s$.

So, by the inductive assumption, $Tab[w]$ was set properly and there exists an $\mathcal{I}$-connected path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_l)$ of G_w that has width at most $k-1$, where $X_l = \text{bag}(w)$. Let $X_{l+1} := \text{bag}(s)$ and let $\mathcal{P}' := (X_1, X_2, \ldots, X_l, X_{l+1})$.

We claim that $\mathcal{P}'$ is a path decomposition of G_s. Indeed, $\bigcup_{i=1}^{l+1} X_i = \bigcup_{i=1}^{l} X_i \cup X_{l+1} = \text{cover}(w) \cup \text{bag}(s) = \text{cover}(s)$. Now, consider an edge vu of G_s. If both v, u belong to $\text{cover}(w)$, they appear in some X_i for $i \leq l$

(by the inductive assumption). If both v, u belong to $\mathrm{bag}(s)$, we are done too. Finally, if $v \in \mathrm{cover}(w)$ and $u \in \mathrm{bag}(s) \setminus \mathrm{cover}(w)$, we know from (S2) that $v \in \mathrm{border}(w) \subseteq \mathrm{bag}(s)$, so we are again in the previous case. Now suppose for a contradiction that there are some $1 \leq i < j \leq l$, such that $X_i \cap X_{l+1} \not\subseteq X_j$. This means that $\mathrm{bag}(s) = X_{l+1}$ contains a vertex of $\mathrm{cover}(w) \setminus \mathrm{bag}(w)$, which is a contradiction with (S4).

Moreover, since $\mathrm{width}(\mathcal{P}) \leq k-1$ and $|\mathrm{bag}(s)| \leq k$, we have $\mathrm{width}(\mathcal{P}') \leq k-1$. Finally we prove that $\mathcal{P}'$ is $\mathcal{I}$-connected. Consider any connected component H of $G[X_1 \cup \cdots \cup X_i]$ for some $i \in \{1, \ldots, l+1\}$. If $\alpha(H, \mathcal{P}') \leq l$, then $X_{\alpha(H,\mathcal{P})} \cap V(H) = X_{\alpha(H,\mathcal{P}')} \cap V(H)$ contains a vertex from $\mathcal{I}$ because $\mathcal{P}$ is $\mathcal{I}$-connected. Otherwise $\alpha(H, \mathcal{P}') = l+1$. Then clearly $i = l+1$ and H is a connected component of $G[X_{l+1}]$. By (S1), H contains a vertex from $\mathcal{I}$. Thus, $\mathcal{P}'$ is $\mathcal{I}$-connected. This justifies setting $Tab[s] = true$.

Case 2: $Tab[s]$ **was set by jump extension.** Let $w = (X, \{B_S\}_S, \{f^{B_S}\}_S)$, $s = (X, \{B_S\}_S, \{g^B_S\}_S)$ and let $S' \subseteq X$ be defined as in the definition of the jump extension. To simplify the notation, set $\mathcal{B}' := \mathcal{B}(S') \setminus B_{S'} = \{H_1, H_2, \ldots, H_m\}$. Observe that since S' is a bottleneck, we have $|\mathcal{B}(S')| \geq 2k+1$, thus there is at least one $H \in \mathcal{B}'$. Since $V(H) \not\subseteq \mathrm{cover}(w)$ and $V(H) \subseteq \mathrm{cover}(s)$ by (J1) and (J3), we have $|\mathrm{cover}(w)| < |\mathrm{cover}(s)|$ and thus $w \prec s$. So, by the inductive assumption, $Tab[w]$ was set properly to be *true* and there exists an $\mathcal{I}$-connected path decomposition $\mathcal{P} = (X_1, X_2, \ldots, X_l)$ of G_w with width at most $k-1$, where $X_l = X = \mathrm{bag}(w)$. By (J4), for every $H \in \mathcal{B}'$ there is a path decomposition $\mathcal{P}_H = (X^H_1, X^H_2, \ldots, X^H_{l(H)})$ of width at most $k - |X| - 1$, such that X^H_1 contains a neighbor of S' or a vertex in $\mathcal{I}$, i.e., $\mathcal{P}_H$ is $(N_G(S') \cap V(H)) \cup \mathcal{I}$-connected.

We claim that

$$\mathcal{P}' = \mathcal{P} \circ \left(\prod_{i=1}^{m} \prod_{j=1}^{l(H_i)} (X_l \cup X^{H_i}_j) \right) \circ X_l,$$

where both $\circ$ and $\prod$ denote concatenation of appropriate sequences, is an $\mathcal{I}$-connected path decomposition of G_s of width at most $k-1$.

First, observe that $\mathrm{cover}(s) = \mathrm{cover}(w) \cup \bigcup_{H \in \mathcal{B}'} V(H)$ due to (J2) and (J3). By the definition of $\mathcal{P}$ and decompositions $\mathcal{P}_H$ for $H \in \mathcal{B}'$, we observe that $\mathcal{P}'$ covers exactly $\mathrm{cover}(s)$.

Now consider an edge vu of G_s. If both vertices v, u belong to $\mathrm{cover}(w)$, or to $V(H)$ for some $H \in \mathcal{B}'$, then, by the definition of $\mathcal{P}$ and $\mathcal{P}_H$, both v and u appear in some bag of the decomposition $\mathcal{P}'$. If $v \in \mathrm{cover}(w)$ and $u \in$

$V(H)$ for some $H \in \mathcal{B}'$, then we know that $v \in \text{border}(w)$ and therefore $v \in X_l$, so both vertices appear in every bag containing u. Finally, we observe that there are no edges joining vertices from different S'-branches.

The third condition of the definition of path decomposition follows directly from the definition of $\mathcal{P}$ and $\mathcal{P}_H$ and the fact that subgraphs H are S'-branches.

Observe that $|X_i| \leq k$ for $i \leq l$ (by the definition of $\mathcal{P}$), and since $|X_j^H| \leq k - |X|$ for every H and j, we have $|X \cup X_j^H| \leq k$, so $\text{width}(\mathcal{P}') \leq k-1$.

Denote $\mathcal{P}' = (X_1, \ldots, X_l, X_{l+1}, \ldots, X_{l'})$. Consider any connected component H of $G[X_1 \cup \cdots \cup X_i$ for some $i \in \{1, \ldots, l'\}$. If $i \leq l$, then H has a vertex from $X_{\alpha(H,\mathcal{P}')} \cap \mathcal{I} = X_{\alpha(H,\mathcal{P})} \cap \mathcal{I}$ as required. So, let $i > l$. If H is contained in some subgraph in $\mathcal{B}'$, then by $(J4)$, $(N_G(S') \cap V(H)) \cup \mathcal{I}$-connected and hence H has a vertex from $X_{\alpha(H,\mathcal{P}')} \cap \mathcal{I}$. If H is not contained in any subgraph from $\mathcal{B}'$, then again by $(J4)$, H has a vertex in X_l. But then, by $\mathcal{I}$-connectivity of $\mathcal{P}$, H has a vertex in $X_{\alpha(H,\mathcal{P}')} \cap \mathcal{I}$. Thus, $\mathcal{P}'$ is $\mathcal{I}$-connected.

This completes the proof. □

Combining Lemmas 4.4.9 and 4.4.10, we obtain the following corollary.

Corollary 4.4.2. *The algorithm is correct, i.e., the value of $Tab[s]$ is true for some state s with $\text{cover}(s) = V(G)$ if and only if $\text{cpw}(G) \leq k-1$.* □

Now let us estimate the computational complexity of our algorithm.

Lemma 4.4.11. *For every fixed $k \geq 1$, a graph G with n vertices, and $\mathcal{I} \subseteq V$, there is an algorithm deciding in time $f(k) \cdot n^{O(k^2)}$ whether G has an $\mathcal{I}$-connected path decomposition of width at most $k-1$, where f is a function depending on k only.*

Proof. We do induction on k. First, observe that for a connected graph G, $\text{cpw}(G) = 0$ if and only if G is a single-vertex graph. Moreover, $\text{cpw}(G) = 1$ if and only if G is a caterpillar, and optimal connected path decompositions of caterpillars have very simple structure, so we can verify in polynomial time whether there is an $\mathcal{I}$-connected one.

So assume that $k \geq 2$ and that the claim holds for $k-1$. For given G and $\mathcal{I}$, we run the dynamic programming algorithm that we described in Section 4.3. The correctness of the algorithm follows from Corollary 4.4.2. Now let us estimate its computational complexity. Recall that the total number of states is $O(n^{3k})$, so the total number of pairs of states is $O(n^{6k})$. For each pair of states we check if one of the two extension rules can be applied.

Observe that for each state s, we can compute $\text{cover}(s)$, $\text{bag}(s)$ and $\text{border}(s)$ in polynomial time. Thus checking if the step extension can be applied can also be done in polynomial time.

Now consider the possible jump extension from a state w to a state s. Verifying the first three conditions can be clearly done in polynomial time. We check in (J4) if the appropriate path decomposition $\mathcal{P}_H$ of each S'-branch H exists by calling the algorithm recursively with the initial set $(N_G(S') \cap V(H)) \cup \mathcal{I}$. By the inductive assumption, this can be done in total time bounded by $n^{O(1)} \cdot f'(k-1)n^{c \cdot (k-1)^2}$, for some function f' and a constant c'. This gives total time complexity

$$n^{O(1)} \cdot n^{6k} \cdot f'(k-1) \cdot n^{c'(k-1)^2} = f(k) \cdot n^{O(k^2)}$$

for some function f. □

Now, the main result of the paper follows easily from Lemma 4.4.11.

Theorem 4.4.1. *For every fixed $k \geq 1$, there is an algorithm deciding in time $f(k) \cdot n^{O(k^2)}$ whether $\text{cpw}(G) \leq k-1$, for some function f depending on k only, i.e., in time polynomial in n.*

Proof. For every vertex $s^* \in V$, we run the dynamic programming algorithm for $\mathcal{I} = \{s^*\}$, i.e., we exhaustively guess a vertex in the first bag of some fixed solution. By Lemma 4.4.11, the total running time is as claimed. □

Let us point out that we did not try to optimize the dependence of the degree of the polynomial function in Theorem 4.4.1 on k, as we were only interested in finding a polynomial algorithm.

4.5 Open Problems

As pointed out, both pathwidth and connected pathwidth are asymptotically the same for an arbitrary graph G, namely $\text{cpw}(G)/\text{pw}(G) \leq 2 + o(1)$. However, there are several open questions regarding the complexity of exact algorithms for connected pathwidth. One such immediate question that is a natural next step in the context of our work is whether connected pathwidth is FPT with respect to this parameter. We conjecture that this is indeed the case.

Conjecture 4.5.1. *Determining whether a given graph with n vertices has connected pathwidth at most k can be done in time $f(k) \cdot n^{O(1)}$, for some function f, i.e., the problem is FPT with respect to k.*

Also, it is not known if connected pathwidth can be computed faster than in time $O^*(2^n)$ for an arbitrary n-vertex graph (recall that this is possible for pathwidth).

Finally, let us point out that the notion of connected pathwidth appeared in the context of pursuit-evasion games called node search, edge search or mixed search. A challenging and long-standing open question related to those games is whether their connected variants belong to NP. See [9] for more details regarding this question.

Part II

Exploration

Chapter 5

Minimizing the Cost of Team Exploration

A group of mobile agents has to explore a graph in the *cost-optimal* way, where the cost is understood as the total distance traversed by agents coupled with the cost of invoking them. This model describes well the real life problems, where every traveled unit costs (e.g., used fuel or energy) and entities costs itself (e.g., equipping new machines or software license cost).

This chapter is constructed as follows: in the next section we introduce the necessary notation and formally define the problem. The further two sections present results for rings. In Section 5.2 the cost-optimal algorithm for the off-line setting is presented, whereas in Section 5.3 a 2-competitive algorithm in the on-line setting is described. It is also proved, that for a positive invoking cost and any on-line algorithm there exist a ring, which force the algorithm to produce at least $3/2$ times higher cost than the optimal, off-line one. Section 5.4 contains the cost-optimal algorithm and its analysis for trees in the off-line setting, while Section 5.5 provides the proof that no algorithm can perform better on trees in the on-line setting than DFS. In other words, the competitive ratio for every on-line algorithm is no less than 2. We finish this chapter with the summary and a future outlook, and suggest areas of further research. The extended abstract of the results presented in this chapter has been published in [129].

5.1 The Model

Let $\mathcal{G}$ be a class of simple, undirected, edge-weighted, connected graphs. For every $G = (V(G), E(G), w) \in \mathcal{G}$ we denote the sets of vertices and edges of G as $V = V(G)$ and $E = E(G)$, respectively, $n = |V|$ and a weight function $w : E \to \mathbb{R}^+$. The sum of all weights of a subgraph H of G is

denoted by $w(H) = \sum_{e \in E(H)} w(e)$. For every tree T and the pair of vertices $v, u \in V(T)$ we denote a path between them as $P_T(v, u)$ and omit the bottom index, when a graph is clear from the context. For any tree T and vertex $v \in V(T)$ we define *branch* as a subtree rooted in a child c of v enlarged by the vertex v and edge (v, c).

We assume, for simplicity, that in one step only one agent can perform a move. In other words, a strategy $\mathcal{S}$ is a sequence of moves of the following two types: (1) traversing an edge by an agent and (2) invoking a new agent in the homebase. In this model we consider only vertex-exploration, which we refer shortly as to exploration. The edge-exploration for rings is not an interesting problem (one agent has to traverse simple each edge) and for trees it is identical to the vertex-exploration. Agents are being invoked in one homebase and after the exploration, they do not have to return to their starting position. Let $\mathcal{S}$ be a strategy constructed for some graph G, $k \in \mathbb{N}^+$ be the number of agents used by $\mathcal{S}$ (notice that k is not fixed) and $d_i \in \mathbb{R}^+ \cup \{0\}$ the distance traversed by the i-th agent during the execution of $\mathcal{S}$, $i \in \{1, \ldots, k\}$. Let q be the *invoking cost*. We define the *cost* of $\mathcal{S}$ as $c(\mathcal{S}) = kq + \sum_{i=1}^{k} d_i$. In other words, cost is understood as the sum of invoking costs and the total distance traversed by entities. Intuitively, before exploring any vertex, the algorithm, which computes the strategy needs to decide what is more profitable: invoke a new agent (and pay for it q) or use an agent already present in the graph. The number of agents, that can be invoked, is unbounded. The goal is to find an algorithm, which for any graph G computes the cost-optimal strategy.

In the on-line setting it is assumed that an agent, which occupies the vertex v, knows the length of edges incident to v and the *status* of vertices adjacent to v, i.e., if they have been already explored. We assume that agents can freely communicate with each other regardless of their location.

Our algorithms are stated as centralized, but because they run in the synchronous settings and in the global communication model, they can be easily converted to the distributed ones. Indeed, as every agent knows its id and positions of all other agents on a graph, it can compute its next move using a presented algorithm.

5.2 Rings in the Off-line Setting

Let $\mathcal{C} \subset \mathcal{G}$ be a class of undirected, edge-weighted rings. For every $C = (V, E, w) \in \mathcal{C}$ of order n, we denote the vertices as $V = \{v_i,\ i \in \{0, \ldots, n-1\}\}$ and edges as $E = \{e_i = \{v_i, v_{i+1}\},\ i \in \{0, \ldots, n-2\}\} \cup e_{n-1} =$

$\{v_{n-1}, v_0\}$. Without loss of generality, let a homebase of C be in v_0. We define the problem in the off-line setting as follows:

Off-line Ring Problem Statement

> Find an algorithm that for any given ring C with a homebase h and the invoking cost q computes a strategy of the minimum cost.

Note that, in the cost-optimal solution exactly one of the edges does not have to be traversed. Indeed, because every vertex has to be explored, in order to minimize the cost, agents can omit only one edge. Procedure `RingOffline` finds in $O(n)$ steps, which edge is optimal to omit. If this edge is incident to the homebase, then only one agent is used, which simply traverses the whole ring without it (lines 8-9 and 18-19). Otherwise, depending on the cost q, there might be one or two agents in use. Let e be an omitted edge and let $C' = C \backslash e$, i.e., C' is a tree rooted in v_0 with two leaves. We denote as v^{min} and v^{max} the closer and further, respectively, leaf in C'. If the invoking cost q is lower than $d_{C'}(v_0, v^{min})$, then it is more efficient to invoke two agents, which traverse two paths $P_{C'}(v_0, v^{min})$ and $P_{C'}(v_0, v^{max})$ (lines 11-14). On the other hand, if $q \geq d_{C'}(v_0, v^{min})$, then only one agent is used, which traverses the path $P_{C'}(v_0, v^{min})$ twice (lines 15-17).

We give a formal statement of the procedure `RingOffline` and make an observation about its cost-optimality.

Observation 5.2.1. *For any invoking cost q and ring C, the strategy $\mathcal{S}$, returned by the procedure* `RingOffline` *is cost-optimal.*

Proof. Let C be any ring, q any invoking cost and $\mathcal{S}$ the strategy returned by the procedure `RingOffline` for C and q. Because every vertex has to be explored, exactly one edge does not have to be traversed. Our procedure computes for every edge $e \in E(C)$, the optimal cost c_e of exploring $C' = C \backslash e$. If e is incident to the homebase, then C' is a path with a homebase in one of its ends. Thus, the cost-optimal strategy uses one agent and $c_e = q + w(C')$. If e is not incident to the homebase, then C' is a path with a homebase in one of its internal vertices. Thus, the cost-optimal strategy uses one or two agents. In the first case, an agent has to traverse to the closer end of C' and then along the whole path. In the second case, each of the agents traverses from the homebase to one of the end vertices. Thus, $c_e = \min\{q + d' + w(C'), 2q + w(C')\}$, where d' is the distance between the homebase and the closer end vertex in C'. At the end `RingOffline` chooses an edge, which deleting leads to the lowest cost and sets the corresponding strategy. □

Procedure `RingOffline`

Input: Ring C, homebase v_0, invoking cost q
Result: Strategy $\mathcal{S}$

1: $C_i \leftarrow C \backslash e_i,\ i \in \{0, \ldots, n-1\}$
2: $v_i^{min} \leftarrow$ a vertex $v \in \{v_i, v_{i+1}\}$ for which $d_{C_i}(v_0, v)$ is minimum, $i \in \{1, \ldots, n-2\}$
3: $v_i^{max} \leftarrow$ a vertex $v \in \{v_i, v_{i+1}\}$ for which $d_{C_i}(v_0, v)$ is maximum, $i \in \{1, \ldots, n-2\}$
4: $c_i \leftarrow q + w(C_i),\ i = 0, n-1$
5: $c_i \leftarrow \min\{2q + w(C_i), q + d_{C_i}(v_0, v_i^{min}) + w(C_i)\},\ i \in \{1, \ldots, n-2\}$
6: Let i_{min} be the index of the minimum element of $\{c_i,\ i \in \{0, \ldots, n-1\}\}$
7: Add a move to $\mathcal{S}$: invoke an agent a_1 in v_0
8: **if** $i_{min} == 0$ **then**
9: Add a sequence of moves to $\mathcal{S}$: traverse by a_1 path $P_{C_{i_{min}}}(v_0, v_1)$
10: **else if** $i_{min} > 0$ and $i_{min} < n-1$ **then**
11: **if** $2q + w(C_i) < q + d_{C_i}(v_0, v_{i_{min}}^{min}) + w(C_i)$ **then**
12: Add a sequence of moves to $\mathcal{S}$: traverse by a_1 path $P_{C_{i_{min}}}(v_0, v_{i_{min}}^{min})$
13: Add a move to $\mathcal{S}$: invoke an agent a_2 in v_0
14: Add a sequence of moves to $\mathcal{S}$: traverse by a_2 path $P_{C_{i_{min}}}(v_0, v_{i_{min}}^{max})$
15: **else**
16: Add a sequence of moves to $\mathcal{S}$: traverse by a_1 path $P_{C_{i_{min}}}(v_0, v_{i_{min}}^{min})$
17: Add a sequence of moves to $\mathcal{S}$: traverse by a_1 path $P_{C_{i_{min}}}(v_{i_{min}}^{min}, v_{i_{min}}^{max})$
18: **else**
19: Add a sequence of moves to $\mathcal{S}$: traverse by a_1 path $P_{C_{i_{min}}}(v_0, v_{n-1})$
20: return $\mathcal{S}$

5.3 Rings in the On-line Setting

In this subsection we present a 2-competitive algorithm `RingOnline`, which for any unknown ring C produces in $O(n)$ moves an exploration strategy. We also prove the lower bound of $3/2$ for the competitive ratio for any $q > 0$. We define the problem in the on-line setting as follows:

On-line Ring Problem Statement

> Find an algorithm that for any given invoking cost q computes a cost-optimal strategy for every *a priori* unknown ring C.

We start by giving the informal description of the procedure `RingOnline`. Let $\mathcal{S}$ be the strategy returned by the procedure `RingOnline` for a given homebase v_0, invoking cost q and some ring C. Firstly $\mathcal{S}$ invokes an agent a_1 in v_0 and denotes as e_1 and e_{-1} edges incident to v_0, with the lower and higher weight respectively (lines 4-5). Agent a_1 traverses first e_1 and then continues the exploration process as long as it is profitable, i.e., the cost of traversing the next edge is less or equal to the invoking cost plus $w(e_{-1})$ (lines 7-10). If at some point a new agent is invoked, then it traverses the edge e_{-1} (lines 11-16). We notice here that the lines 11-16 are executed at most once, as these are initial moves for the second agent. Later, the greedy approach is performed: an edge with lesser weight is traversed either by a_1 (lines 7-10) or by a_2 (lines 17-21). Below we give a formal statement of the procedure `RingOnline`.

The next lemma says that for any invoking cost and any ring algorithm `RingOnline` returns the solution at most twice worse than the optimum, which is tight.

Lemma 5.3.1. *The algorithm* `RingOnline` *is* 2*-competitive.*

Proof. Let $C \in \mathcal{C}$ be any ring for which the cost-optimal, off-line strategy $\mathcal{S}^{opt}$ uses two agents or omits an edge incident to the homebase. We notice that procedure `RingOnline` computes the cost-optimal strategy for C. However, the situation is different otherwise.

Let now $C \in \mathcal{C}$ be any ring with the homebase in v_0 for which the cost-optimal strategy uses one agent and omits the edge not incident to the homebase. Let q be any invoking cost and $\mathcal{S}$ be a strategy returned by the procedure `RingOnline`. Denote as e_{max} the edge of C of the maximum weight and as e' the edge incident to v_0 of the bigger weight. Let e be an omitted edge in the cost-optimal off-line strategy and v^{min} be the closer leaf in the tree $C \backslash e$ rooted in v_0. The cost-optimal strategy $\mathcal{S}^{opt}$ uses one agent, that traverses firstly to the v^{min}, then returns to the homebase and

Procedure `RingOnline`

Input: Homebase v_0, invoking cost q
Result: Strategy $\mathcal{S}$

$i_r \leftarrow 1$
$i_l \leftarrow -1$
$s \leftarrow 1$
Add a move to $\mathcal{S}$: invoke an agent a_1 in v_0
Denote as e_1 and e_{-1} edges adjacent to v_0, with the lower and higher weight respectively
while Graph is not explored **do**
    while $(w(e_{i_l}) + q \cdot s) \geq w(e_{i_r})$ and graph is not explored **do**
        Add a move to $\mathcal{S}$: traverse e_{i_r} by a_1
        Denote the unexplored edge incident to the vertex occupied by a_1 as e_{i_r+1}
        $i_r \leftarrow i_r + 1$
    if $s == 1$ and $(w(e_{-1}) + q) < w(e_{i_r})$ **then**
        Add a move to $\mathcal{S}$: invoke an agent a_2 in v_0
        Add a move to $\mathcal{S}$: traverse e_{-1} by a_2
        Denote the unexplored edge incident to the vertex occupied by a_2 as e_{-2}
        $i_l \leftarrow -2$
        $s \leftarrow 0$
    if $s == 0$ **then**
        while $w(e_{i_l}) < w(e_{i_r})$ and graph is not explored **do**
            Add move to $\mathcal{S}$: traverse e_{i_l} by a_2
            Denote the unexplored edge incident to the vertex occupied by a_2 as e_{i_l-1}
            $i_l \leftarrow i_l - 1$
return $\mathcal{S}$

explores the rest of the ring apart from the edge e. Thus, its cost can be lower bounded by:

$$c(\mathcal{S}^{opt}) = q + w(C\backslash e) + d_{C\backslash e}(v_0, v^{min}) > q + w(C\backslash e).$$

If $q + w(e') < w(e_{max})$, then $\mathcal{S}$ uses two agents and omits e_{max}, i.e.,

$$c(\mathcal{S}) = 2q + w(C\backslash e_{max}).$$

On the other hand, if $q + w(e') \geq w(e_{max})$, then $\mathcal{S}$ invokes one agent, which traverses the whole ring apart from e', i.e,

$$\begin{aligned} c(\mathcal{S}) = q + w(C\backslash e') = q + w(C\backslash e_{max}) + w(e_{max}) - w(e') \leq \\ \leq 2q + w(C\backslash e_{max}). \end{aligned}$$

This leads to the following upper bound for the competitive ratio

$$\frac{2q + w(C\backslash e_{max})}{q + w(C\backslash e)} \leq \frac{2q + w(C\backslash e)}{q + w(C\backslash e)} \leq 2. \tag{5.1}$$

The bound of 2 can be reached for the three vertices ring $C' \in \mathcal{C}$, where $w(e_0) = w(e_2) = 1$ and $w(e_1) = q$ for q large enough. Indeed, we notice that although both strategies, $\mathcal{S}$ and $\mathcal{S}^{opt}$, invoke only one agent, in the cost-optimal solution it traverses the edge e_0 twice instead of traversing the edge e_1, i.e., $c(\mathcal{S}^{opt}) = q + 3$ and $c(\mathcal{S}) = 2q + 1$, which finishes the proof.

□

The theorem below shows that for any positive invoking cost and any on-line algorithm there exist a ring for which the strategy computed by this algorithm achieves at least $3/2$ times higher cost than the optimal one.

Theorem 5.3.1. *For any invoking cost $q > 0$ and any on-line algorithm A, which computes strategies for rings, the competitive ratio is at least $3/2$.*

Proof. Let ϵ be a small positive number, such that $\epsilon \ll q$ and $q \mod \epsilon = 0$. Let $\mathcal{C}_1, \mathcal{C}_2 \subset \mathcal{C}$ be two classes of rings constructed as follows. For every $i \in \{3, 4, \ldots\}$ we add to the class $\mathcal{C}_1$ a ring C_1^i of order i with a homebase in v_0 and set the weights of all edges as ϵ. For every positive integer i and $j \in \{i+2, i+3, \ldots\}$ we add to the class $\mathcal{C}_2$ a ring $C_2^{(i,j)}$ of order j with homebase in v_0. We set the weights of all edges, apart from (v_i, v_{i+1}) with the weight $2q$, as ϵ. Let $\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2$.

We are going to show that for any on-line algorithm A, there exists a ring $C \in \mathcal{C}'$, for which there exists a strategy $\mathcal{S}'$, such that $c(\mathcal{S}) \geq \frac{3}{2}c(\mathcal{S}')$, where $\mathcal{S}$ is a strategy computed by A for C. Ring C is being build during the execution of the algorithm by an *adversary*.

Let $G_i, i \in \mathbb{N}$, be an explored part of the graph at the end of the i-th move, starting from $G_0 = v_0$. Firstly, one agent a_1 is invoked in v_0 and explores some part of a ring. Let $j_1 \in \mathbb{N}$ be a move in which the second agent a_2 is invoked (we set j_1 as infinity, if an algorithm decides to use only one agent in a strategy). For every $i < j_1$, if during the i-th move a_1 explores a new edge (v, v') of the weight ϵ, a new vertex u and edge (v', u) are added by an adversary. The weight of (v', u) is set as ϵ, if $w(P_{G_i}(v_0, v')) < q$, and $2q$, if $w(P_{G_i}(v_0, v')) = q$. We consider two cases. In the first one, when the new agent is invoked only edges of the length ϵ are visible. In the second case, a_1 reaches a vertex incident to the edge of the weight $2q$ before the second agent is invoked or a_1 explores the graph on its own.

Case A: *in the move j_1 only edges of the length ϵ are visible.* We can treat G_{j_1} as a tree rooted in v_0, where v_0 has two branches. Denote the number of vertices in them as $1 \leq h_1 < q$ and $0 \leq h_2 \leq h_1$ (where by $h_2 = 0$, we understand that G_{j_1} is a path of the length $\epsilon \cdot h_1$ starting in v_0). We omit the case when a_2 is invoked in the second move (i.e., $h_1 = h_2 = 0$), as then competitive ratio of at least 2 can be easily obtained (e.g., for a triangle with edges of the weight ϵ). An adversary chooses a ring $C = C_1^{h_1+h_2+2}$ from the class $\mathcal{C}_1$. See Figure 5.1a for the illustration.

We notice that $|V(C)| = h_1 + h_2 + 2$ and $|V(G_{j_1})| = h_1 + h_2 + 1$. In order to explore G_{j_1}, a_1 traversed at least twice the path $P_{G_{j_1}}(v_0, v_{h_1+2})$ (possible empty if $h_2 = 0$) and once $P_{G_{j_1}}(v_0, v_{h_1})$. Then, in the j_1-th move the second agent is invoked, which generates the extra cost q. In order to explore the whole C, at least one extra move of cost ϵ has to be done (e.g., a_1 can traverse from v_{h_1} to v_{h_1+1}). Thus, the total cost of exploring C by $\mathcal{S}$ can be lower bounded by

$$c(\mathcal{S}) \geq 2q + \epsilon(2h_2 + h_1 + 1).$$

Let $\mathcal{S}'$ be a strategy, which explores C by visiting all vertices exactly once with one agent. This leads to the following upper bound of the cost-optimal solution

$$c(\mathcal{S}^{opt}) \leq c(\mathcal{S}') = q + \epsilon(h_1 + h_2 + 1).$$

As $\epsilon h_1 < q$ and $h_2 \geq 0$, we obtain the following lower bound of the competitive ratio

$$\lim_{\epsilon \to 0} \frac{c(\mathcal{S})}{c(\mathcal{S}^{opt})} \geq \lim_{\epsilon \to 0} \frac{2q + \epsilon(h_1 + 2h_2 + 1)}{q + \epsilon(h_1 + h_2 + 1)} \geq \lim_{\epsilon \to 0} \frac{2q + q + 2 \cdot 0 + \epsilon}{q + q + 0 + \epsilon} = \\ = \lim_{\epsilon \to 0} \frac{3q + \epsilon}{2q + \epsilon} = \frac{3}{2}. \tag{5.2}$$

Case B: *a_1 reaches a vertex incident to the edge of the weight $2q$ before the second agent is invoked or a_1 explores the graph on its own.* Let j_2 be the move in which a_1 explores the vertex incident to the edge of the weight $2q$. We can treat G_{j_2} as a tree rooted in v_0, where v_0 has two branches. Denote the number of vertices in them as $h_1 = q/\epsilon$ and $0 \leq h_2 < h_1$. An adversary chooses a ring $C = C_2^{(h_1+h_2+2,h_1)}$ from the class $\mathcal{C}_2$. See Figure 5.1b for the illustration. Let $\mathcal{S}'$ be a strategy, which uses one agent, which firstly explores vertices $v_0, v_{h_1+h_2+1}, v_{h_1+h_2}, \ldots, v_{h_1+1}$, then returns to v_0 and explores the rest of C omitting the edge of the weight $2q$. This leads to the following upper bound of the cost-optimal solution

$$c(\mathcal{S}^{opt}) \leq c(\mathcal{S}') = q + 2\epsilon(h_2 + 1) + \epsilon h_1 = 2q + 2\epsilon(h_2 + 1).$$

In the j_2-th move only one vertex of C is not explored by $\mathcal{S}$: v_{h_1+1}, which is incident to the edges of the weights ϵ and $2q$, and agent a_1 occupies the vertex v_{h_1}. The remaining vertex can be either explored by a_1 or by a newly invoked agent a_2.

Subcase B1: *a_1 explores the vertex v_{h_1+1}.* In the cheapest solution agent a_1 returns to v_0 and traverses the path $v_0, v_{h_1+h_2+1}, v_{h_1+h_2}, \ldots, v_{h_1+1}$ of the length $\epsilon(h_2 + 1)$. Thus, the total cost of exploring C by $\mathcal{S}$ can be lower bounded by

$$c(\mathcal{S}) \geq q + \epsilon(2h_2 + 2h_1 + h_2 + 1) = 3q + \epsilon(3h_2 + 1),$$

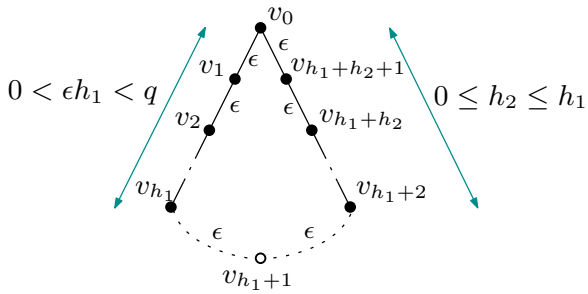
which leads to the following competitive ratio

$$\lim_{\epsilon \to 0} \frac{c(\mathcal{S})}{c(\mathcal{S}^{opt})} \geq \lim_{\epsilon \to 0} \frac{3q + 3\epsilon h_2 + \epsilon}{2q + 2\epsilon h_2 + 2\epsilon} = \frac{3}{2}. \tag{5.3}$$

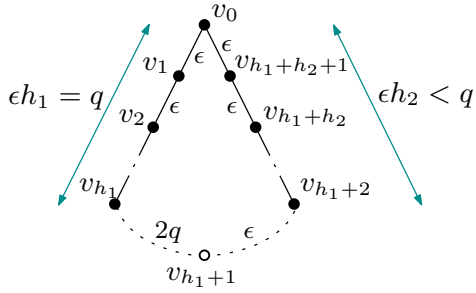
Subcase B2: *a newly invoked agent a_2 explores the vertex v_{h_1+1}.* The total cost of exploring C by $\mathcal{S}$ can be lower bounded by

$$c(\mathcal{S}) \geq (q + 2\epsilon h_2 + \epsilon h_1) + (q + \epsilon h_2 + 1) = 3q + 3\epsilon h_2 + \epsilon,$$

giving the same bound of $\frac{3}{2}$ and finishing the proof.



(A) Ring $C_1^{h_1+h_2+2} \in \mathcal{C}_1$.



(B) Ring $C_2^{(h_1+h_2+2,h_1)} \in \mathcal{C}_2$.

FIGURE 5.1: Illustration of rings from the classes $\mathcal{C}_1$ and $\mathcal{C}_2$; *black dotes* and *solid lines* denote already explored part of the graph, whereas *circles* and *dashed lines* stand for the unvisited part of the rings.

□

At the end we observe, that for $q = 0$ and any ring the strategy returned by the procedure `RingOnline` is cost-optimal.

5.4 Trees in the Off-line Setting

Let $T = (V, E, w) \in \mathcal{G}$ be a tree rooted in a homebase r and $\mathcal{L}(T)$ be the set of all leaves in T. For every $v \in V$ we denote by T_v a subtree of T rooted in v, $c(v)$ list of its children and $p(v)$ its parent vertex.

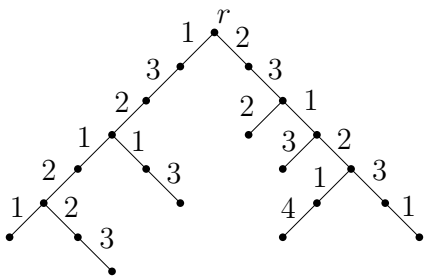
Vertex $v \in V$ is called a *decision vertex* if $|c(v)| \geq 2$ and an *internal vertex* if $|c(v)| = 1$ and v is different from the root. We say that an agent *terminates* in $v \in V$, if v is its last visited vertex. We state the problem in the off-line setting formally:

Off-line Tree Problem Statement

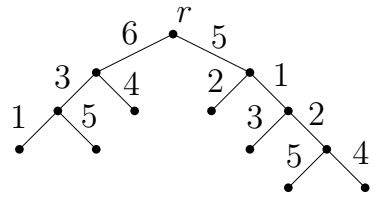
> Find an algorithm that for any given tree T with a homebase at its root and the invoking cost q computes a strategy of the minimum cost.

5.4.1 The Algorithm

In order to simplify our algorithm, a *compressing* operation on a tree T is proceeded. Let $v \in V(T)$ be a decision vertex and $u \in V(T)$ be a decision vertex, a leaf or the root. The new tree T' is obtained by substituting every path $P_T(v, u)$, which apart from u and v consists only internal vertices, with a single edge $e = \{v, u\}$. The weight of e is set as the weight of the whole path, i.e., $w(e) = w(P_T(v, u))$. See Figure 5.2 for an example of the compressing operation.



(A) The original tree T.



(B) Tree T after the compressing operation.

FIGURE 5.2: The compressing operation on an exemplary tree T. The new tree T' has no internal vertices.

Observation 5.4.1. *In every cost-optimal strategy if an agent enters a subtree T_v, it has to explore at least one leaf in it.*

Proof. Let T be any tree, $v \in V$ and a be an agent, which at some point occupies v. If a returns to $p(v)$ or terminates before exploring at least one leaf in T_v, then its moves inside T_v can be omitted. Indeed, in this situation a has reached vertices either (1) already explored, or (2) one lying on the path between v and an unexplored leaf, which will be visited later anyway (by a or any other agent). □

Observation 5.4.2. *In every cost-optimal strategy once an agent leaves any subtree, it never comes back to it.*

Proof. Let T be any tree rooted in r and $v \in V$ different than r. By contradiction, let $\mathcal{S}$ be the cost-optimal strategy for T, in which an agent a after leaving T_v returns to it. Denote as l_a the leaf in which a terminates (not necessarily $l_a \in \mathcal{L}(T_v)$). We split the walk $W(r, l_a)$ traversed by a into parts. We define:

- $W^1(r, p(v))$ as a walk that a traverses until it reaches v, excluding v;
- $W^2(v, v)$ as a walk that a traverses inside T_v before it leaves it;
- $W^3(p(v), v)$ as a walk that a traverses after leaving T_v and before reaching v;
- $W^4(v, l_a)$ as a walk that a traverses after $W^3(p(v), v)$.

In other words

$$W(r, l_a) = W^1(r, p(v)) \circ (p(v), v) \circ W^2(v, v) \circ \\ \circ (v, p(v)) \circ W^3(p(v), v) \circ W^4(v, l_a).$$

The total weight is then the following sum:

$$w(W(r, l_a)) = w(W^1(r, p(v))) + w((p(v), v)) + w(W^2(v, v)) + \\ + w((v, p(v))) + W^3(p(v), v) + w(W^4(v, l_a)).$$

Let S' be a strategy in which a traverses: (1) $W^1(r, p(v))$, (2) $W^3(p(v), v)$, (3) $W^2(v, v)$ and (4) $W^4(v, l_a)$. Then

$$w(W(r, l_a)) = w(W^1(r, p(v))) + W^3(p(v), v) + \\ + w(W^2(v, v)) + w(W^4(v, l_a))$$

and $\mathcal{S}'(T) < \mathcal{S}(T)$, which is a contradiction that $\mathcal{S}$ is cost-optimal. □

Remark 5.4.1. *Let v be any internal vertex. It is never optimal for an agent, which occupies v, to return to the previously occupied vertex in its next move.*

Proof. Let T be any tree and $v \in V$ be any internal vertex. Assume that at some step of the optimal strategy, agent a occupies v. If the last traversed edge of a is $\{p(v), v\}$, then it follows directly from Observation 5.4.1. On the other hand, the remark is true otherwise from Observation 5.4.2. □

In other words, it is always optimal for agents to continue movement along the path once entered. Thus, if we find the optimal strategy for compressed tree T', then we can easily obtain the optimal strategy for T. The only difference is that instead of walking along one edge $\{v, u\}$ in T', the agent has to traverse the whole path $P_T(v, u)$ in T. From now on, till the end of this subsection, whenever we talk about trees, we refer to its compressed version.

For all vertices $v \in V(T)$ we consider a labeling Λ_v, which is a triple (k, u_l, u_c), where k stands for the minimum number of agents needed to explore the whole subtree T_v by any cost-optimal strategy. The second one, u_l, is the furthest leaf from v in T_v (if there is more than one, then v is chosen arbitrary) and u_c is the child of v, such that $u_l \in T_{u_c}$. We will refer to this values using the dot notation, e.g., the number of agents needed to explore tree rooted in v is denoted by $\Lambda_v.k$. The set of labels for all vertices is denoted by $\mathbf{\Lambda} = \{\Lambda_v,\ v \in V(T)\}$.

Procedures

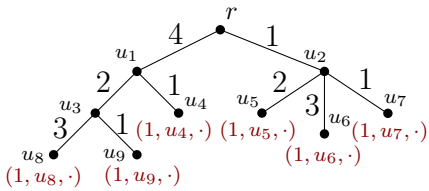
The algorithm is built on the principle of dynamic programming: first the strategy is set for leaves, then gradually for all subtrees and finally for the root. We present three procedures: firstly, labeling $\mathbf{\Lambda}$ is calculated by `SetLabeling`, which is the main core of our algorithm. Once labels for all the vertices are set, the procedure `SetStrategy` builds a strategy based on them. The main procedure `CostExpl` describes the whole algorithm.

Procedure `SetLabeling` for every subtree T_v, calculates and returns labeling Λ_v. We give a formal statement of the procedure and its informal description followed by an example. Firstly, for every leaf v label $\Lambda_v = (1, v, \text{null})$ is set, as one agent is sufficient to explore v. Then, by recursion, labels for the ancestors are set until the root r is reached. Let us describe now how the labeling for the vertex v is established based on the labeling of its children (main loop, lines 9-16). Firstly, the number of needed agents for v is increased by the number of needed agents for its child u (line 10). Then, if the distance between v and the furthest leaf in T_u (i.e., $d(v, \Lambda_u.u_l)$) is less or equal to the distance from the root r to v plus the invoking cost q, the number of required agents is reduced by 1 (lines 12-13). Intuitively, it is more efficient to reuse this agent, than to invoke a new one from r. As we show formally later at most one agent can be returned, and it can happen only if $\Lambda_u.k = 1$. Meanwhile the child of v, which is an ancestor of the furthest leaf in T_v is being set (lines 14-16). See the formal statement of the procedure and an example on the Figure 5.3.

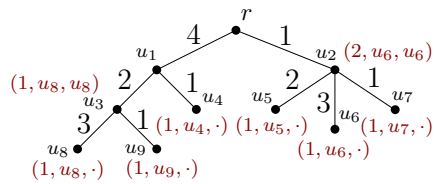
Procedure `SetLabeling`

Input: Tree T, vertex v, invoking cost q, labeling $\mathbf{\Lambda}$
Result: Updated $\mathbf{\Lambda}$

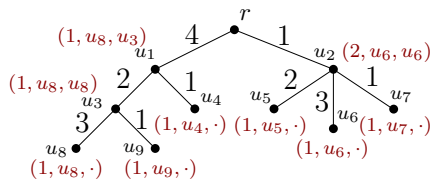
1: **if** $v \in \mathcal{L}(T)$ **then**
2: $\quad \Lambda_v \leftarrow (1, v, \text{null})$
3: $\quad$ return $\mathbf{\Lambda}$
4: **for each** $u \in c(v)$ **do**
5: $\quad$ Invoke Procedure `SetLabeling` for T, u, q and $\mathbf{\Lambda}$
6: $k, d^{max} \leftarrow 0$
7: $u_c^{max} \leftarrow \text{null}$
8: $d_r \leftarrow d(r, v) + q$
9: **for each** $u \in c(v)$ **do**
10: $\quad k \leftarrow k + \Lambda_u.k$
11: $\quad d \leftarrow d(v, \Lambda_u.u_l)$
12: $\quad$ **if** $\Lambda_u.k == 1$ and $d \leq d_r$ **then**
13: $\quad\quad k \leftarrow k - 1$
14: $\quad$ **if** $d > d^{max}$ **then**
15: $\quad\quad d^{max} \leftarrow d$
16: $\quad\quad u_c^{max} \leftarrow u$
17: $k \leftarrow \max\{1, k\}$
18: $\Lambda_v \leftarrow (k, \Lambda_{u_c^{max}}.u_l, u_c^{max})$
19: return $\mathbf{\Lambda}$



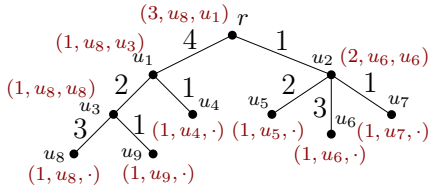
(A) Firstly, labels for leaves are set.



(B) Then, gradually labels are being set for the ancestors, until the root is reached.



(C)



(D) Three agents are required to explore this tree in the cost-optimal way.

FIGURE 5.3: An example of the performing of the procedure `SetLabeling`, where $q = 0$.

Procedure `SetStrategy` builds a strategy for a given subtree T_v based on the labeling $\mathbf{\Lambda}$. If $v \in V(T) \setminus \mathcal{L}(T)$, then for each of its child u, firstly, the required number of agents is sent to u (line 7) and then the strategy is set for u (line 8). Lastly, for all children u of v (apart from the one, which has to be visited as the last one) if it is efficient for the agent, which finished exploration of T_u in $\Lambda_u.u_l$, to come back to v, then the 'return' sequence of moves is added (lines 9-10). It is crucial that for every v the subtree $T_{\Lambda_v.u_c}$ is explored as the last one, but the order of the remaining subtrees is not important (line 5). To summarize, we give a formal statement of the algorithm.

Procedure `CostExpl` consists of two procedures presented in the previous subsections. Firstly, `SetLabeling` is being invoked for the whole tree T. And then the strategy $\mathcal{S}$ is being calculated from the labeling $\mathbf{\Lambda}$ by the procedure `SetStrategy`. We observe that `CostExpl` finds a strategy in $O(n)$ time. To summarize, we give a formal statement of the algorithm.

5.4.2 Analysis of the Algorithm

In this subsection, we analyze the algorithm by providing the necessary observations and lemmas and give the lower and upper bounds. Firstly,

Procedure `SetStrategy`

Input: Tree T, vertex v, invoking cost q, labeling $\mathbf{\Lambda}$, strategy $\mathcal{S}$
Result: Strategy $\mathcal{S}$

if $v \notin \mathcal{L}(T)$ **then**
  if $v == r$ **then**
    Add a move to $\mathcal{S}$: invoke $\Lambda_r.k$ agents in r
  $d_r \leftarrow d(r, v) + q$
  Let $c_1, \ldots, c_l$ be children of v, where $c_l = \Lambda_v.u_c$
  for $i \in \{1, \ldots, l\}$ **do**
    Add a sequence of moves to $\mathcal{S}$: traverse $\{v, c_i\}$ by $\Lambda_{c_i}.k$ agents
    Invoke Procedure `SetStrategy` for $T, c_i, q, \mathbf{\Lambda}$ and $\mathcal{S}$
    if $d(v, \Lambda_{c_i}.u_l) \leq d_r$ and $c_i \neq \Lambda_v.u_c$ **then**
      Add a sequence of moves to $\mathcal{S}$: send an agent back from $\Lambda_{c_i}.u_l$ to v

Procedure `CostExpl`

Input: Tree T, invoking cost q
Result: Strategy $\mathcal{S}$

Invoke Procedure `SetLabeling` for T, r, q and $\emptyset$; set $\mathbf{\Lambda}$ as an output
Invoke Procedure `SetStrategy` for $T, r, q, \mathbf{\Lambda}$ and $\emptyset$; set $\mathcal{S}$ as an output
Return $\mathcal{S}$

let us make a simple observation about the behavior of agents in the cost-optimal strategies.

Observation 5.4.3. *In every cost-optimal strategy all agents terminate in leaves and every leaf is visited exactly once.*

Proof. Let T be any tree and $\mathcal{S}$ be the cost-optimal strategy for T in which an agent a terminates in $v \in V(T) \setminus \mathcal{L}(T)$. By Observation 5.4.1 agent a has explored at least one leaf. Let $u \in \mathcal{L}(T)$ be the last explored leaf by a. Notice, that every vertex visited by a after it leaves u lies on a path between root and some other leaf, which means that either it has been already explored or will be later by some other agents. Thus, these moves are unnecessary and $\mathcal{S}(T)$ is not minimal. The latter part of the observation is obvious. □

In our strategies, subtrees T_v of the maximum height $d(r, v) + q$ are always explored by one agent. The next observation says that in the cost-optimal solution this agent finishes in the furthest leaf of T_v.

Observation 5.4.4. *If $\mathcal{S}$ is a cost-optimal strategy that uses one agent, then, then this agent terminates in one of the furthest leaves.*

Proof. Let T be any tree rooted in r. Let $\mathcal{S}$ be the cost-optimal strategy for T, in which an agent a terminates in leaf l (Observation 5.4.3). Thanks to Observation 5.4.2 we notice that agent a simply performs DFS on T truncated by moves from l to r. In other words, the total cost is equal to $2w(T) - d(r, l)$, thus l should be the furthest leaf. □

Let $v \in V(T)$ different then root. Lemma 5.4.1 guarantees us that after the exploration of T_v at most one agents returns to $p(v)$. Lemma 5.4.2 and Theorem 5.4.1 present our main results.

Lemma 5.4.1. *In every cost-optimal strategy if an agent leaves any subtree, it has explored it on its own.*

Proof. Let T be any tree rooted in r and $v \in V$ different then r. By contradiction, let $\mathcal{S}$ be the cost-optimal strategy for T, in which T_v is explored by at least two agents and at least one of them leaves T_v at some step.

Let A be a group of agents, which terminates in leaves of T_v and B a group of agents, which visits at least one vertex of T_v, but terminates in leaves outside T_v. From the assumption we have that $|A \cup B| \geq 2$ and $B \neq \emptyset$. For every $a \in A \cup B$ let u_a be the last visited leaf from T_v (which existence is guaranteed by Observation 5.4.1) and let a terminates in l_a. Thanks to the Observation 5.4.2 we can split the walk $W_a(r, l_a)$ traversed by every agent $a \in A \cup B$ into parts. We define:

– $W_a^1(r, p(v))$, $a \in A \cup B$ as a walk that a traverses until it reaches v for the first time, excluding v;

– $W_a^2(v, u_a)$, $a \in A \cup B$ as a walk that a traverses inside T_v until it explores u_a;

– $W_a^3(p(v), l_a)$, $a \in B$ as a walk that a traverses after leaving T_v, excluding v.

We obtain that,

$$w(W_a(r, l_a)) = w(W_a(r, u_a)) = w(W_a^1(r, p(v))) + w((p(v), v)) + w(W_a^2(v, u_a)),$$

for every $a \in A$, and

$$w(W_a(r, l_a)) = w(W_a^1(r, p(v)) + w((p(v), v)) + w(W_a^2(v, u_a)) + d(u_a, p(v)) + w(W_a^3(p(v), l_a),$$

for every $a \in B$. We consider two cases.

Case A: $A \neq \emptyset$. We choose and arbitrary agent $a' \in A$ and modify its walk, so after $W_{a'}^1(r, p(v))$ it traverses all the walks $W_a^2(v, u_a)$, $a \in B$ returning every time to v. All of the agents $a \in B$ traverses first $W_a^1(r, p(v))$ and then $W_a^3(p(v), l_a)$, i.e., there is no agent that leaves T_v. Obtained in that way $\mathcal{S}'$ is a proper strategy, which explores the whole tree T. Let now L and L' be the total distances traversed by agents from $A \cup B$ in $\mathcal{S}$ and $\mathcal{S}'$, respectively. We get the following:

$$\begin{aligned} L = &\sum_{a \in A} \Big(w(W_a^1(r, p(v))) + w((p(v), v)) + w(W_a^2(v, u_a)) \Big) + \\ &+ \sum_{a \in B} (w(W_a^1(r, p(v))) + w((p(v), v)) + w(W_a^2(v, u_a)) + \\ &+ d(u_a, p(v)) + w(W_a^3(p(v), l_a)), \qquad (5.4) \\ L' = &\sum_{a \in A} \Big(w(W_a^1(r, p(v))) + w((p(v), v)) + w(W_a^2(v, u_a)) \Big) + \\ &+ \sum_{a \in B} \Big(w(W_a^1(r, p(v)) + w(W_a^3(p(v), l_a)) \Big) + \\ &+ \sum_{a \in B} \big(w(W_a^2(v, u_a)) + d(u_a, v) \big) < L, \qquad (5.5) \end{aligned}$$

which finishes the proof of first case.

Case B: $A = \emptyset$. We choose and arbitrary agent $a' \in B$ and modify its walk, so after $W^1_{a'}(r, p(v))$ it traverses all the walks $W^2_a(v, u_a)$, $a \in B$ returning every time to v. All of the other agents $a \in B$ traverses firstly $W^1_a(r, p(v))$ and then $W^3_a(p(v), l_a)$, i.e., only a' leaves T_v. Obtained in that way $\mathcal{S}'$ is a proper strategy, which explores the whole tree T. Similarly to the previous case one can show that $c(\mathcal{S}') < c(\mathcal{S})$, i.e., we get the contradiction that S is cost-optimal.

□

Lemma 5.4.2. *Let* $\mathbf{\Lambda}$ *be a labeling returned by the procedure* `SetLabeling` *for an arbitrary tree* T. *Every cost-optimal strategy uses at least* $\Lambda_v.k$ *agents to explore* T_v, $v \in V(T)$. [1]

Proof. Let T be any tree of the height H rooted in r. By the induction on the height of a tree h. Firstly, let $h = H$, i.e., we consider labeling of the set $\mathcal{L}(T)$, which is the base case in our procedure. Any cost-optimal strategy uses one agent to explore a leaf, thus $\Lambda_u.k = 1$ is correct.

We assume now, that all the labeling is correct for vertices at levels greater than h and consider vertices on the level h. Notice, that invoking `SetLabeling` in recursive way guarantees, that before the label on any vertex is computed, all of its children's labels are set. Let v be any vertex of T on level $0 \leq h < H$. The algorithm sets

$$\Lambda_v.k = \max\left\{ \sum_{u \in c(v)} \left(\Lambda_u.k - \mathbb{1}_{(d(v,\Lambda_u.u_l) \leq d(r,v)+q)} \right), 1 \right\}.$$

Because once an agent leaves any subtree, it never comes back to it (Observation 5.4.2), subtrees rooted in the children of v can be explored sequentially. Every T_u, $u \in c(v)$ needs $\Lambda_u.k$ agents, which is minimal from the induction assumption. We notice, that after exploring T_u at most one agent might return to v and be used to explore $T_v \setminus T_u$ (Lemma 5.4.1). It can happen only if $\Lambda_u.k = 1$. Thus, this agent has to finish in the furthest leaf of T_u (Observation 5.4.4). The strategy $\mathcal{S}$ reuses all agents (apart from the possible one from the last visited subtree T_u) which finishes the exploration of T_u in the leaf, which is not 'too far', i.e., $d(v, \Lambda_u.u_l) \leq d(r, v) + q$. Indeed, no other agent can be reused, because if $d(v, \Lambda_u.u_l) > d(r, v) + q$, then it is cheaper to call a new agent from the root. Thus, $\Lambda_v.k$ is minimal. □

[1] There exist cost-optimal strategies that can use more than $\Lambda_v.k$ agents. Indeed, if $d(v, \Lambda_v.u_l) = d(r, v) + q$ reusing the agent and calling a new one generates equal cost.

Theorem 5.4.1. *Procedure* `CostExpl` *for every tree T returns a strategy, which explores T in the cost-optimal way.*

Proof. Let T be any tree rooted in r, $\mathbf{\Lambda}$ be the labeling computed by `SetLabeling` and $\mathcal{S}$ be a strategy for T returned by `CostExpl`. Constructing $\mathcal{S}$ for every T_v, $v \in V(T)$ based on $\mathbf{\Lambda}$ is straightforward. Let $u \in c(v) \backslash \{\Lambda_v.u_c\}$. Firstly, traverse $\Lambda_u.k$ agents along $\{v, u\}$. Then, set the strategy for T_u. After the exploration of T_u, if $d(v, \Lambda_u.u_l) \leq d(r, v) + q$, return an agent from $\Lambda_u.u_l$ to v. Repeat for every $u \in c(v) \backslash \{\Lambda_v.u_c\}$ in the random order. For the last child $u = \Lambda_v.u_c$ after the exploration of T_u do not return any agents.

By Lemma 5.4.2 we know that $\Lambda_v.k$ is the minimum number of agents that has to be send to v (or invoked, for $v = r$). The remaining thing to prove is that the order of exploring subtrees does not matter as long as the one with the furthest leaf is visited as the last one. Let

$$\begin{aligned} U_1 &= \{u | u \in c(v),\ d(v, \Lambda_u.u_l) \leq d(r, v) + q\}, \\ U_2 &= c(v) \backslash U_1, \\ \mathcal{T}_i &= \{T_u | u \in U_i\},\ i \in \{1, 2\}, \\ u_c &= \Lambda_v.u_c, \\ T_c &= T_{u_c}. \end{aligned}$$

If $U_1 = \emptyset$, then $\Lambda_v.k = \sum_{u \in c(v)} \Lambda_u.k$ and the order of exploration does not matter. On the other hand, if $U_2 = \emptyset$, then $\Lambda_v.k = 1$ and the agent has to finish in the furthest leaf (Observation 5.4.4), i.e., T_c has to be explored as the last one. Let then $U_1, U_2 \neq \emptyset$, we notice that $T_c \in \mathcal{T}_2$. Our intuition may say that is better to first explore trees from $\mathcal{T}_1$ and then from $\mathcal{T}_2$. But as long as the last tree to visit is from $\mathcal{T}_2$, the order of the rest subtrees does not influence $\Lambda_v.k$ or the cost. Let us consider the strategy, which firstly explores all trees from $\mathcal{T}_1$. The total number of required agents is

$$\Lambda_v.k = 1 + \left(\sum_{u \in U_2 \backslash \{u_c\}} \Lambda_u.k - 1 \right) + \Lambda_{u_c}.k = \sum_{u \in U_2} \Lambda_u.k.$$

This amount does not change for any order of the exploration. Indeed, after the exploration of a tree from $\mathcal{T}_1$ an agent is reused to explore the next tree, decreasing the same $\Lambda_v.k$ by one. See, for example, the strategy in which first are explored trees from $\mathcal{T}_2 \backslash \{T_c\}$. The total number of agents stays the same, i.e,

$$\Lambda_v.k = \sum_{u \in U_2} \Lambda_u.k + 1 + (\Lambda_{u_c}.k - 1) = \sum_{u \in U_2} \Lambda_u.k.$$

At the end we observe, that as trees are explored separately and $\Lambda_v.k$ is the same for any order of the exploration, the total cost is not influenced either. □

Lower and Upper Bounds For any tree T the value of the optimal cost c is bounded by $q + w(T) \leq c \leq q + 2w(T) - H$. A trivial lower bound is achieved on the path graph, where one agent traverses the total distance of $w(T)$. The upper bound can be obtained by performing DFS algorithm by one entity, which set it on $q + 2w(T)$. Let DFS' be the modified version of DFS, such that the agent does not return to the homebase (i.e., terminates in one of the leaves). Then we get an improved upper bound of $q + 2w(T) - H$, where H is the height of T, which is tight (e.g., for paths). It is worth to mention that although DFS' performs well on some graphs, it can be twice worse than `CostExpl`. Let $q \geq 0$ be any invoking cost and $K_{1,n}$ be a star rooted in the internal vertex with edges of the weight $l > q$. While DFS' produces the cost of $c' = q + 2ln - l$, the optimal solution is $c = qn + ln$. The ratio c'/c grows to 2 with the growth of l and n.

5.5 Trees in the On-line Setting

In this subsection we take a closer look at the algorithms for trees in the on-line setting. Because the height of tree T is not known, the upper bound of the cost, set by DFS', is $q + 2w(T) - \epsilon$, where ϵ is some small positive constant. This leads to the upper bound of 2 for the competitive ratio. We are going to prove that it is impossible to construct an algorithm that achieves better competitive ratio than 2. We state the problem in the on-line setting formally:

On-line Tree Problem Statement

> Find an algorithm that for any given invoking cost q computes a cost-optimal strategy for every *a priori* unknown tree T.

Denote by $\mathcal{T} \subset \mathcal{G}$ an infinite class of rooted in v_0 trees, where every edge has weight equal to 1. For every integer $l \in \mathbb{N}^+$, $i \in \{1, \ldots, l\}$ and $l_i \in \{1, \ldots, l\}$, we add to the class $\mathcal{T}$ a tree constructed in the following way:

- construct $l + 1$ paths $P(v_i, v_{i+1})$, $i \in \{0, \ldots, l\}$ of the length l;
- for every $i \in \{1, \ldots, l\}$ construct a path $P(u'_i, u_i)$ of the length $l_i - 1$ (if $l_i = 1$, then $u_i = u'_i$) and add edge $\{v_i, u'_i\}$.

In other words, every graph in $\mathcal{T}$ has a set of decision vertices $\{v_1, \ldots, v_l\}$ and set of leaves $\{v_{l+1}, u_1, \ldots, u_l\}$. Every decision vertex has exactly two children, v_i is an ancestor of v_j and $d(v_i, v_{i+1}) = l$ for every $0 \leq i < j \leq l + 1$. See Figure 5.4.

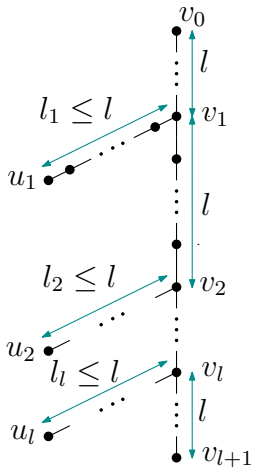


FIGURE 5.4: Illustration of graphs from the class $\mathcal{T}$, where $l \in \mathbb{N}^+$ and $l_i \in \{1, \ldots, l\}, i \in \{1, \ldots, l\}$.

Theorem 5.5.1. *For any invoking cost $q \geq 0$ and any on-line algorithm A, which computes strategies for trees, the competitive ratio is at least* 2.

Proof. DFS' is an example of an algorithm at most twice worse than the best solution, which sets the upper bound. We are going to show that for any invoking cost $q \geq 0$ and any on-line algorithm A, there exists a ring $T \in \mathcal{T}$, for which there exists a strategy $\mathcal{S}'$, such that $c(\mathcal{S}) \geq 2c(\mathcal{S}')$, where $\mathcal{S}$ is a strategy computed by A for T. Tree T is being build during the execution of the algorithm by an *adversary*. Let $l \in \mathbb{N}^+$ be any integer. Values of l_i, $i \in \{1, \ldots, l\}$ are set during the computation of $\mathcal{S}$. For every v_i, $i \in \{1, \ldots, l\}$ three cases can occur.

Case A: *More than one agent reaches v_i before any child of v_i is explored.* The value of l_i is set as 1.

Case B: *One of the agents explores one of the branches of v_i at the depth $0 \leq h < l$ and the second branch at the depth l, before any other agent reaches v_i for the first time.* In this situation an adversary chooses a set of graphs from $\mathcal{T}$ for which the explored vertex at the depth l is v_{i+1} and $l_i = h + 1$. The value of h might be 0, as it takes place e.g., for DFS.

Case C: *One of the agents explores two branches of v_i at the depth $0 \leq h_1 < l$, $1 \leq h_2 < l$, before any other agent reaches v_i for the first time.* Without loss of generality, we assume that the branch explored to the level h_2 is visited as the last one. In this situation an adversary chooses a set of graphs from $\mathcal{T}$ for which vertex v_{i+1} belongs to the branch of v_i explored till the level h_2 and $l_i = h_1 + 1$. Once again, $h_1 = 0$ means that the branch was not explored at all.

When v_{l+1} is explored, all l_i are defined and the set of graphs is narrowed to the exactly one graph, which we denote as T. We claim first that the distance d_0 traversed along the path $P(v_1, v_{l+1})$ is at least $2l^2 - l$ in any $\mathcal{S}$. Let agent a_1 be the one, which explores v_{l+1} and let $k \in \{0, \ldots, l\}$ be the number of decision vertices visited by more than one agent.

Case A': $k = 0$, *i.e., T is explored by one agent.* In other words, for all v_i holds the Case B. We notice that, whenever a strategy $\mathcal{S}$ explores v_{i+1}, $i \in \{1, \ldots, l\}$, exactly one vertex (i.e., leaf u_i) on the path $P(v_i, u_i)$ is unexplored. Thus, $P(v_1, v_{l+1})$ has to be traversed at least twice and $d_0 \geq 2l^2$.

Case B': $k = l$. Path $P(v_1, v_l)$ has to be obviously traversed at least twice and $P(v_l, v_{l+1})$ once, i.e., $d_0 \geq 2l(l-1) + l = 2l^2 - l$.

Case C': $0 < k < l$. In other words, $T_{v_{k+1}}$ is explored by one agent. Paths $P(v_1, v_k)$ and $P(v_{k+1}, v_{l+1})$ are traversed at least twice and $P(v_k, v_{k+1})$ at least once. Thus, $d_0 \geq 2l(k-1) + 2l(l-k) + l = 2l^2 - l$.

Now, we have to analyze paths $P(v_i, u_i)$, $i \in \{1, \ldots, l\}$. We divide decision vertices into the four groups based on the performance of $\mathcal{S}$:

- $V_1 = \{v_i | l_i = 1$, no agent terminates in u_i, $i \in \{1, \ldots, l\}\}$;
- $V_2 = \{v_i | l_i = 1$, at least one agent terminates in u_i, $i \in \{1, \ldots, l\}\}$;
- $V_3 = \{v_i | l_i > 1$, no agent terminates in any vertices of the path $P(v_i, u_i)$, $i \in \{1, \ldots, l\}\}$;
- $V_4 = \{v_i | l_i > 1$, at least one agent terminates in a vertex from the path $P(v_i, u_i)$, $i \in \{1, \ldots, l\}\}$.

Notice that V_1, V_2, V_3 and V_4 form a partition of decision vertices. Let us denote as d_i the total distance traversed by all the agents along $P(v_i, u_i)$ in $\mathcal{S}$. For any $v_i \in V_1$ we have $d_i \geq 2$ and $v_i \in V_2$ we have $d_i \geq 1$. From the way how T is constructed follows, that if $l_i > 1$, then either holds Case B and $h > 0$ or Case C and $h_1 > 0$. In both situations path $P(v_i, p(u_i))$ is first

traversed at least twice, leaving u_i unexplored. If now, no agent terminates in any vertex of $P(v_i, u_i)$, then $P(v_i, p(u_i))$ has to be traversed at least twice more. Thus,

$$d_i \geq 4(l_i - 1) + 2 \geq 4l_i - 2, \qquad v_i \in V_3.$$

On the other hand, if at least one agent terminates in any vertex of $P(v_i, u_i)$, then $P(v_i, p(u_i))$ can be traversed only one extra time. Which leads to,

$$d_i \geq 3(l_i - 1) + 1 \geq 3l_i - 2, \qquad v_i \in V_4.$$

Lastly, we have to consider the extra cost d' generated by agents. Every invoked agent, which terminates on some path $P(v_i, u_i)$ has to traverse the edge $\{v_0, v_1\}$, thus

$$d' \geq (q + l)\,(|V_2| + |V_4|) \geq |V_2| + l|V_4|.$$

The total cost of exploring T by $\mathcal{S}$ can be lower bounded by

$$\begin{aligned} c(\mathcal{S}) &\geq 2l^2 - l + 2|V_1| + |V_2| + \sum_{v_i \in V_3} (4l_i - 2) + \sum_{v_i \in V_4} (3l_i - 2) + |V_2| + \\ &+ l|V_4| \geq 2l^2 - l + 2|V_1| + 2|V_2| + 4 \sum_{v_i \in V_3} l_i - 2|V_3| + 4 \sum_{v_i \in V_4} l_i - 2|V_4| = \\ &= 2l^2 - l + 4 \sum_{i=1}^{l} l_i - 2(|V_1| + |V_2| + |V_3| + |V_4|) = 2l^2 + 4 \sum_{i=1}^{l} l_i - 3l. \end{aligned}$$

Consider now the following off-line strategy $\mathcal{S}'$, which explores the same graph T by using one agent, which after reaching the decision vertex v_i, $i \in \{1, \ldots, l\}$, firstly traverses the path $P(v_i, u_i)$, then returns to v_i and explores further the tree. The agent finally terminates in v_{l+1}. Thus, the path $P(v_0, v_{l+1})$ of the length $(l + 1)l$ is traversed only once and paths $P(v_i, u_i)$, $i \in \{1, \ldots, l\}$ twice. The optimal strategy can be then upper bounded by

$$c(\mathcal{S}^{opt}) \leq c(\mathcal{S}') = q + l^2 + 2 \sum_{i=1}^{l} l_i + l.$$

This leads to the following competitive ratio

$$\lim_{l\to\infty}\frac{c(\mathcal{S})}{c(\mathcal{S}^{opt})} \geq \lim_{l\to\infty}\frac{2l^2+4\sum_{i=1}^{l} l_i - 3l}{q+l^2+2\sum_{i=1}^{l} l_i + l} = \\ = 2 - \lim_{l\to\infty}\frac{5l+2q}{q+l^2+2\sum_{i=1}^{l} l_i + l} = 2, \tag{5.6}$$

which finishes the proof. □

5.6 Conclusions

In this chapter we have proposed a new cost of the team exploration, which is the sum of total traversed distances by agents and the invoking cost which has to be paid for every agent. This model describes well the real life problems, where every traveled unit costs (e.g., used fuel or energy) and entities costs itself (e.g., equipping new machines or software license cost). The algorithms, which construct the cost-optimal strategies for any given edge-weighted ring or tree have been presented. As for the on-line setting a 2-competitive algorithm for rings is given and lower bounds of 3/2 and 2 of the competitive ratio for rings and trees, respectively, are proved. While there is very little done in this area, a lot of new questions have been pondered. Firstly, it would be interesting to consider edge- and vertex-exploration for other classes of graphs. Intuitively, for some of them, the problem would be easy and for some might be NP-hard (e.g., cliques). Another direction is to look more into the problem in the on-line setting. It would be highly interesting to close the gap between lower and upper bounds of the competitive ratio for rings. Another idea is to bound communication for agents, which will make this model truly distributed. Lastly, different variation of this model might be proposed, e.g., the invoking cost might increase/decrease with the number of agents in use or time might be taken under consideration as the third minimization parameter.

Chapter 6

Clearing Directed Subgraphs by Mobile Agents

Consider a network, after a hacker attack, where all terminals are not functioning properly, leaving a number of facilities disconnected. For a team of mobile agents (e.g., recovery software programs), distributed within the network on bases, the main battle is usually first to re-establish connectivity between these facilities. This motivates us to introduce a number of (theoretical) *Link Up* problems in digraphs.

This chapter is constructed as follows: in the next section we introduce the necessary notation and formally define the problem. In the further Section 6.2 it is proved that the Link Up (LU) problem (together with its different variations) is fixed-parameter tractable. In particular, we prove that the LU problem admits a fixed-parameter randomized algorithm with respect to the total number l of facilities and bases, running in $2^{O(l)} \cdot \text{poly}(n)$ time, where $\text{poly}(n)$ is a polynomial in the order n of the input graph. The proof relies on the algebraic framework introduced by Koutis in [103], i.e., first so called the Tree Pattern Embedding Problem is solve, which is described in Subsection 6.2.1. On the other hand, we show that the LU problem (as well as some of its variants) is NP-complete, by a reduction from the Set Cover problem [87] (Section 6.3). Our result on NP-completeness of the LU problem implies NP-completeness of the Agent Clearing Tree problem studied in [110], where the complexity status of the latter has been posed as an open problem. Results presented in this chapter have been published in [52].

6.1 The Model

Let D be a vertex-weighted digraph with an additional subset of vertices, such that its underlying graph is connected. In other words,

$D = (V(D), A(D), F, B)$ is a quadruple, where $V = V(D)$ is a set of all vertices, $A = A(D)$ set of arcs, $F \subseteq V$ and $B : V \to \mathbb{N}$ a vertex-weight function. We define $n = |V|$ and $m = |A|$.

Recall that the vertices of D correspond to terminals while its arcs correspond to (one-way) transmission links, the set F corresponds to locations of facilities, and the set $\mathcal{B} = B^{-1}(\mathbb{N}^+)$ corresponds to vertices, called from now on *bases*, where a (positive) number of agents is placed (so we shall refer to the function B as an *agent-quantity* function). Let $k = \sum_{v \in V} B(v)$ be the total number of agents placed in the digraph.

The Link Up Problem (LU)

> Do there exist k directed walks in D, with exactly $B(v)$ starting points at each vertex $v \in V$, whose edges induce a subgraph H of D such that all vertices in F belong to one connected component of the underlying graph of H? Note that the k directed walks may overlap in vertices and even edges.

The LU problem may be understood as a question, whether for a team of size k, initially located at bases in $\mathcal{B} = B^{-1}(\mathbb{N}^+)$, where the number of agents located at $v \in \mathcal{B}$ is equal to $B(v)$, it is possible to follow k walks in D clearing their arcs so that the underlying graph of the union of cleared walks includes a Steiner tree for all facilities in F.

Related work. The Link Up problem is related to the problems of clearing connections by mobile agents placed at some vertices in a digraph, introduced by Levcopoulos et al. in [110]. In particular, the LU problem is a generalized variant of the Agent Clearing Tree (ACT) problem where one wants to determine a placement of the minimum number of mobile agents in a digraph D such that agents, allowed to move only along directed walks, can simultaneously clear some subgraph of D whose underlying graph includes a spanning tree of the underlying graph of D. In [110], the authors provided a simple 2-approximation algorithm for solving the Agent Clearing Tree problem, leaving its complexity status open.

All the aforementioned clearing problems are variants of the path cover problem in digraphs, where the objective is to find a minimum number of directed walks that cover all vertices (or edges) of a given digraph. Without any additional constraints, the problem was shown to be polynomially tractable by Ntafos and Hakimi in [124]. Several other variants involve additional constraints on walks as the part of the input, see [11, 86, 102, 105, 123, 124, 125] to mention just a few, some of them combined with relaxing the condition that all vertices of the digraph have to be covered by

walks. In particular, we may be interested in covering only a given set of walks that themselves should appear as subwalks of some covering walks (polynomially tractable [124]). We may also be interested in covering only a given set of vertex pairs, where both elements of a pair should appear in the same order and in the same path in a solution (NP-complete even in acyclic digraphs [125]). Finally, for a given family $\mathcal{S}$ of vertex subsets of D, we may be interested in covering only a representative from each of the subsets (NP-complete [125]).

A wider perspective locates our link up problems as variants of the directed Steiner tree problem, where for a given edge-weighted directed graph $D = (V, A)$, a root $r \in V$ and a set of facilities $X \subseteq V$, the objective is to find a minimum cost arborescence rooted at r and spanning all facilities in X (equivalently, there exists a directed path from r to each facility in X) [33, 153]. For some recent works and results related to this problem, see e.g. [1, 84, 96]. We also point out to a generalization of the Steiner tree problem in which pairs of facilities are given as an input and the goal is to find a minimum cost subgraph which provides a connection for each pair [33, 65]. For some other generalizations, see e.g. [34, 107, 145, 148, 149]. Finally, we also remark a different cleaning problem introduced in [119] and related to the variants we study: cleaning a graph with *brushes* — for some recent works, see e.g. [28, 32, 88, 120].

Remark. Note that a weaker version of the LU problem with the connectivity requirement removed, that is, when we require each facility only to be connected to some agents base, admits a polynomial-time solution by a straightforward reduction to the minimum path cover problem in digraphs [124].

Observe that in a border case, all non-zero length walks of agents start at the same vertex of the input digraph $D = (V, A, F, B)$. Therefore, we may assume that the number of agents at any vertex is at most $n - 1$, that is, $B(v) \leq n - 1$ for any $v \in V$, and so the description of any input requires $O(n \log n + m)$ space (recall $m \geq n - 1$).

6.2 The Link Up Problem is Fixed-Parameter Tractable

In this section, we prove that the Link Up problem is fixed-parameter tractable with respect to the number of facilities and bases. The proof relies on the key fact (see Lemmas 6.2.1 and 6.2.2 below) that a restricted variant of the LU problem with the input D can be reduced to the detection of a

particular directed subtree of 'small' order in the transitive closure $\mathrm{TC}(D)$ of D.

We solve the latter tree detection problem by a reduction to the problem of testing whether some properly defined multivariate polynomial has a monomial with specific properties, essentially modifying the construction in [104] designed for undirected trees/graphs.

Let us consider the variant of the LU problem, which we shall refer to as the All-LU problem, where we restrict the input only to digraphs $D = (V, A, F, B)$ that satisfy $\mathcal{B} = B^{-1}(\mathbb{N}^+) \subseteq F$. (In other words, bases can be located only at some facilities.) Observe that D admits a positive answer to the LU problem if and only if there exists a subset $\mathcal{B}'$ of $\mathcal{B} \setminus F$ such that the digraph $D' = (V, A, F', B')$, where $v \in F'$ for $v \in F \cup \mathcal{B}'$ and $v \notin F'$ otherwise, and $B'(v) = B(v)$ for $v \in \mathcal{B}' \cup (F \cap \mathcal{B})$ and $B'(v) = 0$ otherwise, admits a positive answer to the All-LU problem. Therefore, we can immediately conclude with the following lemma.

Lemma 6.2.1. *Suppose that the* All-LU *problem can be solved in* $2^{O(l)} \cdot \mathrm{poly}(n)$ *time, where* l *is the number of facilities in the input (restricted) digraph of order* n*. Then, the* LU *problem can be solved in* $2^{O(l')} \cdot \mathrm{poly}(n)$ *time, where* l' *is the total number of facilities and bases in the input digraph of order* n*.* □

Taking into account the above lemma, we now focus on constructing an efficient fixed-parameter algorithm for the All-LU problem, with the *restricted* input digraph $D = (V, A, F, B)$ satisfying $\mathcal{B} = B^{-1}(\mathbb{N}^+) \subseteq F$. Let $\mathcal{W}$ be a set of walks (if any) that constitute a positive answer to the All-LU problem in D. We say that $\mathcal{W}$ is *tree-like* if all walks in $\mathcal{W}$ are strongly arc-distinct, that is, they are arc-distinct and if there is a walk in $\mathcal{W}$ traversing the arc $(u, v) \in A$, then there is no walk in $\mathcal{W}$ traversing its complement $(v, u) \in A$, and the underlying graph of their union is acyclic and includes a Steiner tree for F. Notice that if $\mathcal{W}$ is tree-like, then all walks in $\mathcal{W}$ are just (simple) paths. See Figure 6.1 for an example.

Lemma 6.2.2. *A (restricted) instance* $D = (V, A, F, B)$ *admits a positive answer to the* All-LU *problem if and only if the transitive closure* $\mathrm{TC}(D) = (V, A', F, B)$ *of* D*, with the same subset* F *and vertex-weight function* B*, admits a positive answer to the* All-LU *problem with a tree-like set of walks whose underlying graph is of order at most* $2|F| - 1$*.*

Since the transitive closure $\mathrm{TC}(D) = (V, A', F, B)$ inherits the subset F and the function B from the restricted instance D, we emphasize that $\mathrm{TC}(D)$ is a proper (restricted) instance to the All-LU problem.

Proof. ($\Leftarrow$) It follows from the fact that a directed walk in the transitive closure $\mathrm{TC}(D)$ corresponds to a directed walk in D.

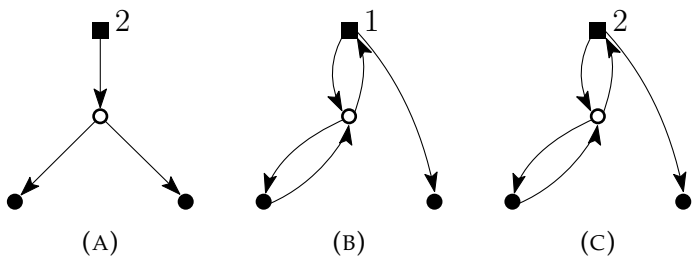


FIGURE 6.1: Example of the definition of tree-like set of walks; *black squares* denote bases and *black dots* denote facilities, which are not bases. There exists a tree-like set of walks on 6.1c, but not on 6.1a and 6.1b.

($\Rightarrow$) Assume that the agents initially located at vertices in $\mathcal{B}$, with respect to the agent-quantity function B, can simultaneously follow k directed walks $\pi_1, \ldots, \pi_k$ whose edges induce a subgraph H of D such that the underlying graph of H includes a Steiner tree of F. Consider now the same walks in the transitive closure $\mathrm{TC}(D)$. To prove the existence of a tree-like solution of 'small' order, the idea is to transform these k walks (if ever needed) into another strongly arc-distinct k walks. The latter walks have the same starting points as the original ones (thus preserving the agent-quantity function B) and the underlying graph of their union is a Steiner tree of F (in the underlying graph of $\mathrm{TC}(D)$) having at most $|F| - 1$ non-terminal vertices.

Our transforming process is based on the following 2-step modification. First, assume without loss of generality that the walk $\pi_1 = (v_1, \ldots, v_{|\pi_1|+1})$ has an arc (v_t, v_{t+1}) corresponding to an edge in the underlying graph H of $\bigcup_{i=1}^{k} \pi_i$ that belongs to a cycle (in H), or both (v_t, v_{t+1}) and its complement (v_{t+1}, v_t) are traversed by π_1, or there is another walk traversing (v_t, v_{t+1}) or (v_{t+1}, v_t), see Figure 6.2 for an illustration. If $t = |\pi_1|$, then we shorten π_1 by deleting its last arc (v_t, v_{t+1}). Otherwise, if $t < |\pi_1|$, then we replace the arcs (v_t, v_{t+1}) and (v_{t+1}, v_{t+2}) in π_1 with the arc (v_t, v_{t+2}) that exists because of the transitive closure. One can observe that the underlying graph of the new set of walks is connected, includes a Steiner tree of F, and the vertex v_1 remains the starting vertex of (the new) π_1. But, making a walk cycle-free or strongly arc-distinct may introduce another cycle in the underlying graph, or another multiply traversed arc, or another arc a such that both a and its complement are traversed. However, the length of the modified walk always decreases by one. Consequently, since the initial walks are of the finite lengths, we conclude that applying the above procedure multiple times eventually results in a tree-like set $\Pi = \{\pi_1, \ldots, \pi_k\}$ of walks, being (simple) paths.

Assume now that in this set Π of strongly arc disjoint paths, there is

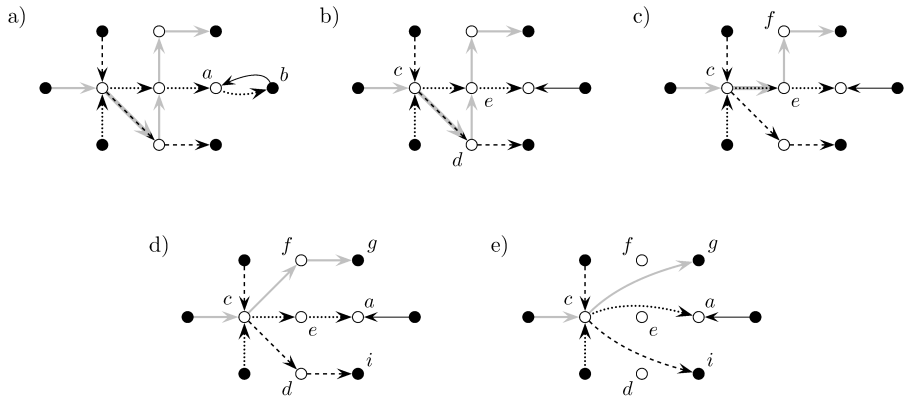


FIGURE 6.2: Transforming walks into a tree-like set of walks. In the first step (a-d), applied three times, we first delete the arc (a, b), then replace two arcs (c, d) and (d, e) with the arc (c, e), and then two arcs (c, e) and (e, f) with the arc (c, f). In the second step (d-e), we replace two arcs (c, d) and (d, i), two arcs (c, e) and (e, a), and two arcs (c, f) and (f, g), with the arcs (c, i), (c, a), and (c, g), respectively.

a non-terminal vertex v of degree at most two in the underlying graph H of $\bigcup_{i=1}^{k} \pi_i$. Without loss of generality assume that v belongs to the path π_1. Similarly as above, if $\deg_H(v) = 1$, then we shorten π_1 by deleting its last arc. Otherwise, if $\deg_H(v) = 2$ and v is not the endpoint of π_1, then modify π_1 be replacing v together with the two arcs of π_1 incident to it by the arc connecting the predecessor and successor of v in π_1, respectively. Observe that since v was a non-terminal vertex in the underlying graph, the underlying graph of (the new) $\bigcup_{i=1}^{k} \pi_i$ is another Steiner tree of F (and does not include v). Moreover, the above modification keeps paths strongly arc-distinct and does not change the starting vertex of π_1. Therefore, by subsequently replacing all such non-terminal vertices of degree at most two, we obtain a tree-like set of k paths in the transitive closure $\mathrm{TC}(D)$ such that the underlying graph of their union is a Steiner tree of F with no degree two vertices except those either belonging to F or being end-vertices of exactly two paths (in $\mathrm{TC}(D)$). Therefore, we conclude that the number of non-terminal vertices in this underlying graph is at most $|F| - 1$, which completes our proof of the lemma.

Indeed, the bound is obvious if $|F| \leq 2$. So assume now that $|F| \geq 3$, the statement is valid for any set $F' \subset V$ with $0 \leq |F'| < |F|$, and let Π be a tree-like set of paths in the transitive closure $\mathrm{TC}(D)$ such that the underlying graph H of their union is a Steiner tree T of F with no degree two

vertices except those either belonging to F or being end-vertices of exactly two paths. Let v be a non-terminal vertex in H (if no such v exists, then there is nothing to prove), and let $\Pi' \subseteq \Pi$ be the set of paths ending at vertex v. By deleting all path arcs with endpoint v for paths in Π' and by replacing two consecutive path arcs incident to v by the relevant arc connecting the predecessor and successor of v in π, respectively, for any path $\pi \in \Pi \setminus \Pi'$, we obtain the set Π' of strongly arc-distinct paths and a non-trivial partition $F_1 \cup \cdots \cup F_r = F$ such that the underlying graph of their union consists of $r \geq 2$ Steiner trees $T_1, \ldots, T_r$ of $F_1, \ldots, F_r$, respectively, all of them with no degree two vertices except those either belonging to F or being end-vertices of exactly two paths. By induction assumption, each tree T_i has at most $|F_i| - 1$ non-terminal vertices, $i = 1, \ldots, r$, and so T has at most $1 + \sum_{i=1}^{r}(|F_i| - 1) \leq |F| - 1$ non-terminal vertices since $r \geq 2$. □

Taking into account the above lemma, a given (restricted) instance $D = (V, A, F, B)$ of the All-LU problem can be transformed (in polynomial time) into the answer-equivalent (restricted) instance $\mathrm{TC}(D) = (V, A', F, B)$ of the *tree-like-restricted* variant of the All-LU problem in which only tree-like paths that together visit at most $2|F| - 1$ vertices are allowed. Observe that $\mathrm{TC}(D) = (V, A', F, B)$ admits a positive answer to the tree-like-restricted All-LU problem if and only if $\mathrm{TC}(D)$ has a subtree $T = (V(T), A(T))$ of order at most $2|F| - 1$ and such that $F \subseteq V(T)$ and all edges of T can be traversed by at most k agents following arc-distinct paths starting at vertices in $\mathcal{B}$ (obeying the agent-quantity function B). This motivates us to consider the following problem.

Let $D = (V, A, F, B)$ be a directed graph of order n and size m, with the subset F of V and a vertex-weight function $B\colon V \to \mathbb{N}$ such that $B^{-1}(\mathbb{N}^+) \subseteq F$, and let $T = (V(T), A(T), L)$ be a directed vertex-weighted tree of order η, with a vertex-weight function $L\colon V(T) \to \mathbb{N}$.

The Tree Pattern Embedding Problem (TPE)

> Does D have a subgraph $H = (V(H), A(H))$ isomorphic to T such that $F \subseteq V(H)$ and $L(v) \leq B(h(v))$ for any vertex $v \in V(T)$, where h is an isomorphism of T and H?

In Subsection 6.2.1, we prove Theorem 6.2.1 given below which states that there is a randomized algorithm that solves the TPE problem in $O^*(2^\eta)$ time, where the notation O^* suppresses polynomial terms in the order n of the input graph D. We point out that if the order η of T is less than $|F|$ or at least $n + 1$, then the problem becomes trivial, and so, in the following, we assume $|F| \leq \eta \leq n$.

Theorem 6.2.1. *There is a randomized algorithm that solves the* TPE *problem with high probability in $O^*(2^\eta)$ time.*

Suppose that for each vertex $v \in V(T)$, the value $L(v)$ corresponds to the number of agents located at v that are required to simultaneously traverse (clear) all arcs of T, in an arc-distinct manner, and T admits a positive answer to the TPE problem in the transitive closure $\mathrm{TC}(D) = (V, A', F, B)$. Then $\mathrm{TC}(D)$ admits a positive answer to the tree-like-restricted All-LU problem, which immediately implies that D admits a positive answer to the All-LU problem (by Lemma 6.2.2). Therefore, taking into account Theorem 6.2.1, we are now ready to present the main theorem of this section. For simplicity of presentation, we now assume that a (restricted) directed graph $D = (V, A, F, B)$ itself (not its transitive closure) is an instance of the tree-like-restricted All-LU problem.

Theorem 6.2.2. *There is a randomized algorithm that solves the tree-like-restricted* All-LU *problem for $D = (V, A, F, B)$ with high probability in $O^*(144^{|F|})$ time.*

Proof. Keeping in mind Lemma 6.2.2, we enumerate all undirected trees of order η, where $|F| \leq \eta \leq 2|F| - 1$ (and $\eta \leq n$); there are $O(9^{|F|})$ such candidates [131]. For each such a η-vertex candidate tree, we enumerate all orientations of its edges, in order to obtain a directed tree; there are $2^{\eta-1}$ such orientations. Therefore, we have $O(36^{|F|})$ candidates for a directed oriented tree T of order η, where $|F| \leq \eta \leq 2|F| - 1$.

For each candidate $T = (V(T), A(T))$, we determine in $O(\eta)$ time how many (at least) agents, together with their explicit locations at vertices in $V(T)$, are needed to traverse all arcs of T, in an arc-distinct manner. This problem can be solved in linear time just by noting that the number of agents needed at a vertex v is equal to $\max\{0, \deg_{\mathrm{out}}(v) - \deg_{\mathrm{in}}(v)\}$ (since arcs must be traversed in an arc-distinct manner). The locations of agents define a vertex-weight function $L\colon V(T) \to \mathbb{N}$. We then solve the TPE problem with the instance D and $T = (V(T), A(T), L)$ in $O^*(2^\eta)$ time by Theorem 6.2.1.

As already observed, if T admits a positive answer to the TPE problem for D, then D admits a positive answer to the tree-like-restricted All-LU problem. Therefore, by deciding the TPE problem for each of $O(36^{|F|})$ candidates, taking into account the independence of any two tests, we obtain a randomized algorithm for the restricted LU problem with a running time $O^*(144^{|F|})$. □

Taking into account Lemma 6.2.1, we immediately obtain the following corollary.

Corollary 6.2.1. *The* LU *problem admits a fixed-parameter randomized algorithm with respect to the total number l of facilities and bases, running in $2^{O(l)} \cdot \mathrm{poly}(n)$ time, where n is the order of the input graph.* □

Minimizing the number of used agents. The first natural variation on the Link Up problem is its minimization variant, which we shall refer to as the min-LU problem, where for a given input n-vertex digraph $D = (V, A, F, B)$, we wish to determine the minimum number of agents among those available at bases in $\mathcal{B} = B^{-1}(\mathbb{N}^+)$ that are enough to guarantee a positive answer to the (original) Link Up problem in D. We claim that this problem also admits a fixed-parameter algorithm with respect to the total number l of facilities and bases, running in time $2^{O(l)}\mathrm{poly}(n)$, and the solution is concealed in our algorithm for the LU problem. Namely, observe that by enumerating all directed trees of order at most $|F|$, see the proof of Theorem 6.2.2, together with the relevant function L, and checking their embeddability in D, we accidentally solve this minimization problem: the embeddable tree with the minimum sum $\sum_{v \in V} L(v)$ constitutes the answer to min-LU problem.

Corollary 6.2.2. *The* min-LU *problem admits a randomized fixed-parameter algorithm with respect to the total number l of facilities and bases, running in $2^{O(l)} \cdot \mathrm{poly}(n)$ time, where n is the order of the input graph.* □

Maximizing the number of re-connected facilities. In the case when for the input digraph $D = (V, A, F, B)$, not all facilities can be re-connected into one component, that is, D admits a negative answer to the Link Up problem, one can ask about the maximum number of facilities in F that can be re-connected by agents located with respect to the agent-quantity function B [152]; we shall refer to this problem as the max-LU problem. Since we can enumerate all subsets of F in $O^*(2^{|F|})$ time, taking into account Theorem 6.2.2, we obtain the following corollary.

Corollary 6.2.3. *The* max-LU *problem admits a randomized fixed-parameter algorithm with respect to the total number l of facilities and bases, running in $2^{O(l)} \cdot \mathrm{poly}(n)$ time, where n is the order of the input graph.* □

No pre-specified positions of agents. Finally, another natural variant of the Link Up problem is to allow any agents to start at any vertex. Formally, we define the following problem.

The Link Up Problem with Unspecified Bases (LUU)

> Given a subset F of V and an integer $k \geq 1$, do there exist k directed walks in a digraph $D = (V, A)$ whose edges induce a subgraph H of D such that the set F is a subset of the vertex set of H and the underlying graph of H is connected? Again, let us emphasize that these k directed walks may overlap in vertices or edges.

We claim that for the LUU problem, there is also a randomized algorithm with the running time $2^{O(k+l)} \cdot \text{poly}(n)$, where $l = |F|$ is the number of facilities, and n is the order of the input graph. The solution is analogous to that for the LU problem. Namely, one can prove a counterpart of Lemma 6.2.2 which allows us to restrict ourselves to the restricted variant where only order $O(k + l)$ tree-like solutions are allowed. Then, the restricted variant is solved also using the algorithm for the TPE problem as a subroutine: the function B is the constant function $B(v) = n$, and among all directed tree candidates, we check only those with $\sum_{v \in V(T)} L(v) \leq k$. We omit details.

Corollary 6.2.4. *The* LUU *problem admits a randomized fixed-parameter algorithm with respect to the number l of facilities and the number k of agents, running in* $2^{O(k+l)} \cdot \text{poly}(n)$ *time, where n is the order of the input graph.* □

Observe that if the number k of available agents is not a part of the input, that is, we ask about the minimum number of walks whose underlying graph includes a Steiner tree for the set of facilities, then this problem does not seem to be fixed-parameter tractable with respect to only the number of facilities. This follows from the fact that the minimum number of agents is unrelated to the number of facilities in the sense that even for two facilities to be connected, a lot of agents may be required, see Figure 6.3 for an illustration. However, a weakness of this example is its (weak) connectivity, and so, without any not strong evidence, we conjecture that if restricted only to strongly connected digraphs, the aforementioned problem becomes then fixed-parameter tractable with respect to only the number of facilities.

6.2.1 The Tree Pattern Embedding Problem

In this section, we solve the TPE problem by providing a randomized polynomial-time algorithm when the parameter η is fixed. Our algorithm is based on the recent algebraic technique using the concepts of monotone arithmetic circuits and monomials, introduced by Koutis in [103], developed by Williams and Koutis in [104, 150], and adapted to some other graph problems, e.g., [17, 18, 19, 78].

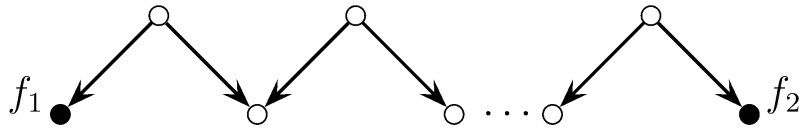


FIGURE 6.3: Two facilities f_1 and f_2 require $n-1$ agents, where n is the order of the digraph.

A (*monotone*) *arithmetic circuit* is a directed acyclic graph where each leaf (i.e., vertex of in-degree 0) is labeled either with a variable or a real non-negative constant (*input gates*), each non-leaf vertex is labeled either with $+$ (an *addition gate* with an unbounded fan-in) or with $\times$ (a *multiplication gate* with fan-in two), and where a single vertex is distinguished (the *output gate*). Each vertex (gate) of the circuit represents (computes) a polynomial — these are naturally defined by induction on the structure of the circuit starting from its input gates — and we say that a polynomial is *represented* (*computed*) *by an arithmetic circuit* if it is represented (computed) by the output gate of the circuit. Finally, a polynomial that is just a product of variables is called a *monomial*, and a monomial in which each variable occurs at most once is termed a *multilinear monomial* [103, 150].

We shall use a slight generalization of the main results of Koutis and Williams in [103, 150], provided by them in Lemma 1 in [104], which, in terms of our notation, can be expressed as follows.

Lemma 6.2.3. ([104]) *Let $P(x_1,\ldots,x_n,z)$ be a polynomial represented by a monotone arithmetic circuit of size $s(n)$ and let t be a non-negative integer. There is a randomized algorithm that for input P runs in $O^*(2^k t^2 s(n))$ time and outputs "YES" with high probability if there is a monomial of the form $z^t Q(x_1,\ldots,x_n)$, where $Q(x_1,\ldots,x_n)$ is a multilinear monomial of degree at most k, in the sum-product expansion of P, and always outputs "NO" if there is no such monomial $z^t Q(x_1,\ldots,x_n)$ in the expansion.* □

Taking into account the above lemma, for the input digraph $D = (V(D), A(D), F, B)$ and directed tree $T = (V(T), A(T), L)$, the idea is to construct an appropriate polynomial $Q(X,z)$ such that $Q(X,z)$ contains a monomial of the form $z^{|\mathcal{S}|}b(X)$, where $b(X)$ is a multilinear polynomial with exactly $|V(T)|$ variables in X and $\mathcal{S} = F \cup B^{-1}(\mathbb{N}^+)$, if and only if the TPE problem has a solution for the input D and T (see Lemma 6.2.4 below). Intuitively, the fact that $b(X)$ is a multilinear polynomial ensures

that no two vertices of T are mapped into the same vertex of D in the corresponding embedding. Moreover, the requirement that $b(X)$ has exactly $|V(T)|$ variables ensures that no vertex of T is missing in the mapping and no vertex is mapped twice. Finally, the variable z 'counts' the number of properly mapped vertices according to the functions L in T and B in D.

Polynomial construction. Let $D = (V(D), A(D), F, B)$ be a directed graph, with a subset F of $V(D)$ and a vertex-weight function $B\colon V(D) \to \mathbb{N}$, and let $T = (V(T), A(T), L)$ be a directed vertex-weighted tree of order η, with a vertex-weight function $L\colon V(T) \to \mathbb{N}$. We consider T to be rooted at a vertex $r \in V(T)$, and we remind that for a non-root vertex v of T, $p(v)$ is a parent of v. Now, for $v \in V(T)$, define two sets $N_T^+(v)$ and $N_T^-(v)$:

$$N_T^+(v) = \left\{u \in V(T) \;\middle|\; (u,v) \in A(T) \text{ and } u \neq p(v)\right\},$$
$$N_T^-(v) = \left\{u \in V(T) \;\middle|\; (v,u) \in A(T) \text{ and } u \neq p(v)\right\}.$$

The idea is to treat T as a 'pattern' that we try to embed into the digraph D, with respect to a subset F and functions B, L. Denote $\mathcal{S} = F \cup B^{-1}(\mathbb{N}^+)$ for brevity. We say that T has an *$\mathcal{S}$-embedding into D* if the following holds (these are the formal conditions that need to be satisfied for the embedding to be correct):

A. There exists an injective function (homomorphism) $f\colon V(T) \to V(D)$ such that if $(u,v) \in A(T)$, then $(f(u), f(v)) \in A(D)$.

B. $\mathcal{S} \subseteq f(V(T))$, where $f(V(T)) = \{f(v) \mid v \in V(T)\}$.

C. $L(v) \leq B(f(v))$ for any $v \in V(T)$.

First, for $\mathcal{S} \subseteq V(D)$, $w \in V(D)$ and $u \in V(T)$, we introduce a particular indicator function, used for fulfilling Conditions B and C:

$$\mathbf{z}_{\mathcal{S}}(u,w) = \begin{cases} z, & \text{if } w \in \mathcal{S} \text{ and } L(u) \leq B(w), \\ 1, & \text{if } w \notin \mathcal{S} \text{ and } L(u) \leq B(w), \\ 0, & \text{otherwise, i.e., if } L(u) > B(w). \end{cases}$$

Next, following [104], we define a polynomial $Q(X,T)$ that we then use to test existence of a desired $\mathcal{S}$-embedding of T in D. Namely, we root T at any vertex $r \in V(T)$. Now, a polynomial $Q_{u,w}(X)$, for a subtree T_u of T rooted at $u \in V(T)$ and for a vertex $w \in V(D)$, is defined inductively (in a bottom up fashion on T) as follows. For each $u \in V(T)$ and for each

$w \in V(D)$: if u is a leaf in T, then

$$Q_{u,w}(X) = \mathbf{z}_{\mathcal{S}}(u,w) \cdot x_w, \tag{6.1}$$

and if u is not a leaf in T, then

$$Q_{u,w}(X) = \begin{cases} \mathbf{z}_{\mathcal{S}}(u,w) \cdot x_w \cdot Q^+_{u,w}(X), & \text{if } N^-_T(u) = \emptyset, \\ \mathbf{z}_{\mathcal{S}}(u,w) \cdot x_w \cdot Q^-_{u,w}(X), & \text{if } N^+_T(u) = \emptyset, \\ \mathbf{z}_{\mathcal{S}}(u,w) \cdot x_w \cdot Q^+_{u,w}(X) \cdot Q^-_{u,w}(X), & \text{otherwise,} \end{cases} \tag{6.2}$$

where

$$Q^+_{u,w}(X) = \prod_{v \in N^+_T(u)} \left(\sum_{(w',w) \in A(D)} Q_{v,w'}(X) \right), \tag{6.3}$$

$$Q^-_{u,w}(X) = \prod_{v \in N^-_T(u)} \left(\sum_{(w,w') \in A(D)} Q_{v,w'}(X) \right). \tag{6.4}$$

Finally, the polynomial $Q(X,z)$ is as follows:

$$Q(X,z) = \sum_{w \in V(D)} Q_{r,w}(X). \tag{6.5}$$

We have the following lemma.

Lemma 6.2.4. *The polynomial $Q(X,z)$ contains a monomial of the form $z^{|\mathcal{S}|}b(X)$, where $b(X)$ is a multilinear polynomial with exactly η variables in X, if and only if the η-vertex tree T has an $\mathcal{S}$-embedding into D.*

Proof. Consider a vertex u of T and assume that the subtree T_u is of order j. By a straightforward induction in the size of a subtree we state the following observation. A monomial, call it $z^q x_{w_1} \cdots x_{w_j}$ for this subtree T_u, where $w_i \in V(D)$ for each $i \in \{1,\ldots,j\}$, is present in $Q_{u,w_1}(X)$ if and only if the three following conditions hold.

A. There exists a homomorphism f_u from the vertices of T_u to $w_1,\ldots,w_j$ such that $f_u(u) = w_1$.

B. $|\mathcal{S} \cap \{w_1,\ldots,w_j\}| \leq q$ and the equality holds if $w_1,\ldots,w_j$ are pairwise different.

C. $L(v) \leq B(f_u(v))$ for any vertex v of T_u.

The fact that f_u is a homomorphism follows from the observation that, during construction of $Q_{u,w_1}(X)$ in (6.3) and (6.4), a neighbor v of u is mapped

to a node w' of D in such a way that if $(v,u) \in A(T)$ then $(w',w) \in A(D)$ (see (6.3)), and if $(u,v) \in A(T)$ then $(w,w') \in A(D)$ (see (6.4)). Conditions B and C are ensured by appropriate usage of the indicator function in (6.1), namely, if u is mapped to w in a homomorphism corresponding to $Q_{u,w}(X)$, then we add the multiplicative factor of z to $Q_{u,w}(X)$ provided that $L(v) \leq B(w)$.

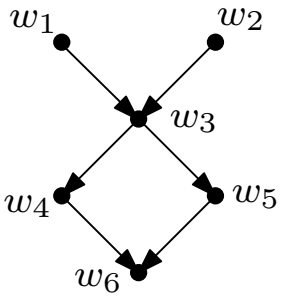
Thus, we obtain that $Q(X,z)$ has a multilinear polynomial $z^{|\mathcal{S}|}x_{w_1}\cdots x_{w_\eta}$ if and only if T has an $\mathcal{S}$-embedding into D. □

Observe that the polynomial $Q(X,z)$ and the auxiliary polynomials $Q^+_{u,w}(X)$, $Q^-_{u,w}(X)$ can be represented by a monotone arithmetic circuit of size polynomial in the order n of the input digraph D. To start with, we need $n+1$ input gates for the variables corresponding to vertices of D, and the auxiliary variable z. With each of the aforementioned polynomials, we associate a gate representing it, which gives in total $O(\eta n)$ such gates. In order to implement the recurrences defining the polynomials, assuming unbounded fan-in of addition gates, we need $O(n)$ auxiliary gates for each recurrence involving large products. Thus, the resulting circuit is of size $O(n^3)$. Hence, by Lemma 6.2.3 combined with Lemma 6.2.4, we conclude that the existence of an $\mathcal{S}$-embedding of the η-vertex tree T into D can be decided in $O^*(2^\eta)$ time. Consequently, we obtain Theorem 6.2.1 by the definition of an $\mathcal{S}$-embedding.

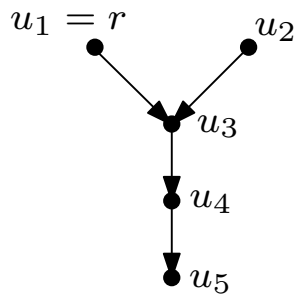
Remark. The above approach can be adapted to the case when we want to embed a directed forest $T = (V, A, F, B)$ of order η into a directed graph. All we need is to build a relevant polynomial for each rooted directed tree-component of T, and then to consider the product $S(X,T)$ of these polynomials, asking about the existence of a monomial of the form $z^{|\mathcal{S}|}b(X)$, where $b(X)$ is a multilinear polynomial with exactly η variables in X. Also, by a similar approach, we may consider and can solve (simpler) variants of our embedding problem without the weight function subset F or without the weight functions B and L; details are omitted.

We finish this section by giving two examples with full computations.

Example 1. Consider digraph $D = (V(D), A(D), F, B)$ and tree $T = (V(T), A(T), L)$ like on the Figure 6.4.



(A) $D = (V(D), A(D), F, B)$ is a directed graph, with a subset F of $V(D)$ and a vertex-weight function $B\colon V(D) \to \mathbb{N}$.



(B) $T = (V(T), A(T), L)$ is a rooted in u_1, directed vertex-weighted tree of order 5.

FIGURE 6.4: Illustration for the first example, where $F = \{w_4, w_6\}$, $B^{-1}(\mathbb{N}^+) = \{w_1\}$, $k = \sum_{v \in V(D)} B(v) = B(w_1) = 3$ and $L^{-1}(0) = V(T)$. From which we conclude the set $\mathcal{S} = \{w_1, w_4, w_6\}$.

Let T be rooted in u_1. For every $v \in V(T)$ sets $N_T^+(v)$ and $N_T^-(v)$ are equal to:

$$\begin{aligned} N_T^+(u_1) &= \emptyset & N_T^-(u_1) &= \{u_3\}; \\ N_T^+(u_2) &= \emptyset & N_T^-(u_2) &= \{u_3\}; \\ N_T^+(u_3) &= \{u_2\} & N_T^-(u_3) &= \{u_4\}; \\ N_T^+(u_4) &= \emptyset & N_T^-(u_4) &= \{u_5\}; \\ N_T^+(u_5) &= \emptyset & N_T^-(u_5) &= \emptyset. \end{aligned}$$

Firstly, we compute the value of appropriate polynomials for leaves. For every leaf $u \in \{u_2, u_5\}$ in T and for every $w \in V(D)$ we have:

$$Q_{u,w}(X) = \mathbf{z}_{\mathcal{S}}(u, w) x_w.$$

Then we construct them for every other vertex from $V(T)$. Because $N_T^+(u_1) = \emptyset$ and $N_T^-(u) = \{u_3\}$, for every $w \in V(D)$ we have

$$\begin{aligned} Q_{u_1,w}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w)x_w Q^-_{u_1,w}(X) = \\ &= \mathbf{z}_{\mathcal{S}}(u_1,w)x_w \prod_{v\in N_T^-(u_1)} \left(\sum_{(w,w')\in A(D)} Q_{v,w'}(X) \right) = \\ &= \mathbf{z}_{\mathcal{S}}(u_1,w)x_w \sum_{(w,w')\in A(D)} Q_{u_3,w'}(X), \end{aligned}$$

which leads to

$$\begin{aligned} Q_{u_1,w_1}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w_1)x_{w_1} Q_{u_3,w_3}(X) \\ Q_{u_1,w_2}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w_2)x_{w_2} Q_{u_3,w_3}(X) \\ Q_{u_1,w_3}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w_3)x_{w_3} \left(Q_{u_3,w_4}(X) + Q_{u_3,w_5}(X)\right) \\ Q_{u_1,w_4}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w_4)x_{w_4} Q_{u_3,w_6}(X) \\ Q_{u_1,w_5}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w_5)x_{w_5} Q_{u_3,w_6}(X) \\ Q_{u_1,w_6}(X) &= 0. \end{aligned}$$

Because $N_T^-(u_3) \neq \emptyset$ and $N_T^+(u_3) \neq \emptyset$, for every $w \in V(D)$ we have

$$Q_{u_3,w}(X) = \mathbf{z}_{\mathcal{S}}(u_3,w)x_w Q^+_{u_3,w}(X) Q^-_{u_3,w}(X),$$

where

$$\begin{aligned} Q^+_{u_3,w}(X) &= \prod_{v\in N_T^+(u_3)} \left(\sum_{(w',w)\in A(D)} Q_{v,w'}(X) \right) = \sum_{(w',w)\in A(D)} Q_{u_2,w'}(X), \\ Q^-_{u_3,w}(X) &= \prod_{v\in N_T^-(u_3)} \left(\sum_{(w',w)\in A(D)} Q_{v,w'}(X) \right) = \sum_{(w',w)\in A(D)} Q_{u_4,w'}(X). \end{aligned}$$

That leads to the following polynomials:

$$\begin{aligned} Q_{u_3,w_1}(X) &= 0 \\ Q_{u_3,w_2}(X) &= 0 \\ Q_{u_3,w_3}(X) &= \mathbf{z}_{\mathcal{S}}(u_3,w_3)x_{w_3}(Q_{u_2,w_1}(X) + Q_{u_2,w_2}(X))\cdot \\ &\qquad \cdot (Q_{u_4,w_4}(X) + Q_{u_4,w_5}(X)) \\ Q_{u_3,w_4}(X) &= \mathbf{z}_{\mathcal{S}}(u_3,w_4)x_{w_4} Q_{u_2,w_3}(X) Q_{u_4,w_6}(X) \\ Q_{u_3,w_5}(X) &= \mathbf{z}_{\mathcal{S}}(u_3,w_5)x_{w_5} Q_{u_2,w_3}(X) Q_{u_4,w_6}(X) \\ Q_{u_3,w_6}(X) &= 0. \end{aligned}$$

The last vertex to analyze is u_4, for which $N_T^+(u_4) = \emptyset$. For every $w \in V(D)$ we have

$$\begin{aligned} Q_{u_4,w}(X) &= \mathbf{z}_{\mathcal{S}}(u_4,w)x_w Q^-_{u_4,w}(X) = \\ &= \mathbf{z}_{\mathcal{S}}(u_4,w)x_w \prod_{v \in N_T^-(u_4)} \left(\sum_{(w,w') \in A(D)} Q_{v,w'}(X) \right) = \\ &= \mathbf{z}_{\mathcal{S}}(u_4,w)x_w \sum_{(w,w') \in A(D)} Q_{u_5,w'}(X), \end{aligned}$$

which leads to

$$\begin{aligned} Q_{u_4,w_1}(X) &= \mathbf{z}_{\mathcal{S}}(u_4,w_1)x_{w_1} Q_{u_5,w_3}(X) = \mathbf{z}_{\mathcal{S}}(u_4,w_1)x_{w_1}\mathbf{z}_{\mathcal{S}}(u_5,w_3)x_{w_3} \\ Q_{u_4,w_2}(X) &= \mathbf{z}_{\mathcal{S}}(u_4,w_2)x_{w_2} Q_{u_5,w_3}(X) = \mathbf{z}_{\mathcal{S}}(u_4,w_2)x_{w_2}\mathbf{z}_{\mathcal{S}}(u_5,w_3)x_{w_3} \\ Q_{u_4,w_3}(X) &= \mathbf{z}_{\mathcal{S}}(u_4,w_3)x_{w_3} \left(Q_{u_5,w_4}(X) + Q_{u_5,w_5}(X)\right) = \\ &= \mathbf{z}_{\mathcal{S}}(u_4,w_3)x_{w_3} \left(\mathbf{z}_{\mathcal{S}}(u_5,w_4)x_{w_4} + \mathbf{z}_{\mathcal{S}}(u_5,w_5)x_{w_5}\right) \\ Q_{u_4,w_4}(X) &= \mathbf{z}_{\mathcal{S}}(u_4,w_4)x_{w_4} Q_{u_5,w_6}(X) = \mathbf{z}_{\mathcal{S}}(u_4,w_4)x_{w_4}\mathbf{z}_{\mathcal{S}}(u_5,w_6)x_{w_6} \\ Q_{u_4,w_5}(X) &= \mathbf{z}_{\mathcal{S}}(u_4,w_5)x_{w_5} Q_{u_5,w_6}(X) = \mathbf{z}_{\mathcal{S}}(u_4,w_5)x_{w_5}\mathbf{z}_{\mathcal{S}}(u_5,w_6)x_{w_6} \\ Q_{u_4,w_6}(X) &= 0. \end{aligned}$$

Going now 'backwards' we can compute the missing values for u_3:

$$\begin{aligned} Q_{u_3,w_3}(X) &= \mathbf{z}_{\mathcal{S}}(u_3,w_3)x_{w_3}\mathbf{z}_{\mathcal{S}}(u_5,w_6)x_{w_6}(\mathbf{z}_{\mathcal{S}}(u_2,w_1)x_{w_1} + \mathbf{z}_{\mathcal{S}}(u_2,w_2)x_{w_2}) \cdot \\ &\quad \cdot (\mathbf{z}_{\mathcal{S}}(u_4,w_4)x_{w_4} + \mathbf{z}_{\mathcal{S}}(u_4,w_5)x_{w_5}) \\ Q_{u_3,w_4}(X) &= \mathbf{z}_{\mathcal{S}}(u_3,w_4)x_{w_4}\mathbf{z}_{\mathcal{S}}(u_2,w_3)x_{w_3} \cdot 0 = 0 \\ Q_{u_3,w_5}(X) &= \mathbf{z}_{\mathcal{S}}(u_3,w_5)x_{w_5}\mathbf{z}_{\mathcal{S}}(u_2,w_3)x_{w_3} \cdot 0 = 0. \end{aligned}$$

And for u_1:

$$\begin{aligned} Q_{u_1,w_1}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w_1)x_{w_1}\mathbf{z}_{\mathcal{S}}(u_3,w_3)x_{w_3}\mathbf{z}_{\mathcal{S}}(u_5,w_6)x_{w_6} \cdot \\ &\quad \cdot (\mathbf{z}_{\mathcal{S}}(u_2,w_1)x_{w_1} + \mathbf{z}_{\mathcal{S}}(u_2,w_2)x_{w_2})(\mathbf{z}_{\mathcal{S}}(u_4,w_4)x_{w_4} + \mathbf{z}_{\mathcal{S}}(u_4,w_5)x_{w_5}) \\ Q_{u_1,w_2}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w_2)x_{w_2}\mathbf{z}_{\mathcal{S}}(u_3,w_3)x_{w_3}\mathbf{z}_{\mathcal{S}}(u_5,w_6)x_{w_6} \cdot \\ &\quad \cdot (\mathbf{z}_{\mathcal{S}}(u_2,w_1)x_{w_1} + \mathbf{z}_{\mathcal{S}}(u_2,w_2)x_{w_2})(\mathbf{z}_{\mathcal{S}}(u_4,w_4)x_{w_4} + \mathbf{z}_{\mathcal{S}}(u_4,w_5)x_{w_5}) \\ Q_{u_1,w_3}(X) &= \mathbf{z}_{\mathcal{S}}(u_1,w_3)x_{w_3} \cdot 0 = 0 \\ Q_{u_1,w_4}(X) &= 0 \\ Q_{u_1,w_5}(X) &= 0. \end{aligned}$$

Finally, for $\mathcal{S} = \{w_1, w_4, w_6\}$, we have

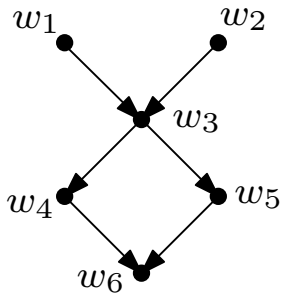
$$Q(X,T) = \sum_{w \in V(D)} Q_{u_1,w}(X) = z^2 x_{w_1} x_{w_3} x_{w_6} (z x_{w_1} + x_{w_2})(z x_{w_4} + x_{w_5}) + \\ + z x_{w_2} x_{w_3} x_{w_6} (z x_{w_1} + x_{w_2})(z x_{w_4} + x_{w_5}). \tag{6.6}$$

We notice that there exist monomials with exactly 5 (the order of T) variables from X and variable z in the power of $|\mathcal{S}| = k = 3$, which means that a tree T can be $\mathcal{S}$-embedded into D. Moreover, the form of $Q(X,T)$ tells us a little bit more about this embedding. After multiplying variables from $Q_{u_1,w_1}(X)$ and $Q_{u_1,w_2}(X)$, we obtain two identical monomials:

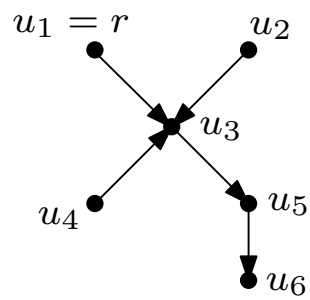
$$z^3 x_{w_1} x_{w_2} x_{w_3} x_{w_4} x_{w_6}.$$

This means that there exists two solutions: (1) u_1 mapped into w_1 and (2) u_1 mapped into w_2. Polynomial $Q(X,T)$ contains no more monomials of the form $z^3 b(X)$, which means that these are the only existing solutions. Indeed, although tree T could be also embedded into D using vertices w_1, w_2, w_3, w_5, w_6 it would not be appropriate solution, while it does not include w_4 from $\mathcal{S}$.

Example 2. Consider digraph $D = (V(D), A(D), F, B)$ and tree $T = (V(T), A(T), L)$ like on the Figure 6.5.



(A) $D = (V(D), A(D), F, B)$ is a directed graph, with a subset F of $V(D)$ and a vertex-weight function $B\colon V(D) \to \mathbb{N}$



(B) $T = (V(T), A(T), L)$ is a rooted in u_1, directed vertex-weighted tree of order 6.

FIGURE 6.5: Illustration for the second example, where $F = \{w_4, w_6\}$, $B^{-1}(\mathbb{N}^+) = \{w_1\}$, $k = \sum_{v \in V(D)} B(v) = B(w_1) = 3$ and $L^{-1}(0) = V(T)$. From which we conclude the set $\mathcal{S} = \{w_1, w_4, w_6\}$.

Let T be rooted in u_1. For every $v \in V(T)$ sets $N_T^+(v)$ and $N_T^-(v)$ are equal to:

$$\begin{aligned} N_T^+(u_1) &= \emptyset & N_T^-(u_1) &= \{u_3\};\\ N_T^+(u_2) &= \emptyset & N_T^-(u_2) &= \{u_3\};\\ N_T^+(u_3) &= \{u_2, u_4\} & N_T^-(u_3) &= \{u_5\};\\ N_T^+(u_4) &= \emptyset & N_T^-(u_4) &= \{u_3\};\\ N_T^+(u_5) &= \emptyset & N_T^-(u_5) &= \{u_6\};\\ N_T^+(u_6) &= \emptyset & N_T^-(u_6) &= \emptyset. \end{aligned}$$

Firstly, we compute the value of appropriate polynomials for leaves. For every leaf $u \in \{u_2, u_4, u_6\}$ in T and for every $w \in V(D)$ we have:

$$Q_{u,w}(X) = \mathbf{z}_S(u, w)x_w.$$

Then we construct them for every other vertex from $V(T)$. Because $N_T^+(u_1) = \emptyset$ and $N_T^-(u_1) = \{u_3\}$, for every $w \in V(D)$ we have

$$\begin{aligned} Q_{u_1,w}(X) &= \mathbf{z}_S(u_1, w)x_w Q^-_{u_1,w}(X) = \\ &= \mathbf{z}_S(u_1, w)x_w \prod_{v \in N_T^-(u_1)} \left(\sum_{(w,w') \in A(D)} Q_{v,w'}(X) \right) = \\ &= \mathbf{z}_S(u_1, w)x_w \sum_{(w,w') \in A(D)} Q_{u_3,w'}(X), \end{aligned}$$

which leads to

$$\begin{aligned} Q_{u_1,w_1}(X) &= \mathbf{z}_S(u_1, w_1)x_{w_1} Q_{u_3,w_3}(X)\\ Q_{u_1,w_2}(X) &= \mathbf{z}_S(u_1, w_2)x_{w_2} Q_{u_3,w_3}(X)\\ Q_{u_1,w_3}(X) &= \mathbf{z}_S(u_1, w_3)x_{w_3} \left(Q_{u_3,w_4}(X) + Q_{u_3,w_5}(X)\right)\\ Q_{u_1,w_4}(X) &= \mathbf{z}_S(u_1, w_4)x_{w_4} Q_{u_3,w_6}(X)\\ Q_{u_1,w_5}(X) &= \mathbf{z}_S(u_1, w_5)x_{w_5} Q_{u_3,w_6}(X)\\ Q_{u_1,w_6}(X) &= 0. \end{aligned}$$

Because $N_T^-(u_3) \neq \emptyset$ and $N_T^+(u_3) \neq \emptyset$ for every $w \in V(D)$ we have

$$Q_{u_3,w}(X) = \mathbf{z}_S(u_3, w)x_w Q^+_{u_3,w}(X) Q^-_{u_3,w}(X),$$

where

$$Q^+_{u_3,w}(X) = \prod_{v\in N^+_T(u_3)} \left(\sum_{(w',w)\in A(D)} Q_{v,w'}(X) \right) = \\ = \sum_{(w',w)\in A(D)} Q_{u_2,w'}(X) \sum_{(w',w)\in A(D)} Q_{u_4,w'}(X),$$

$$Q^-_{u_3,w}(X) = \prod_{v\in N^-_T(u_3)} \left(\sum_{(w',w)\in A(D)} Q_{v,w'}(X) \right) = \sum_{(w',w)\in A(D)} Q_{u_5,w'}(X).$$

That leads to:

$$\begin{aligned}
&Q_{u_3,w_1}(X) = 0\\
&Q_{u_3,w_2}(X) = 0\\
&Q_{u_3,w_3}(X) = \mathbf{z}_S(u_3,w_3)x_{w_3}(Q_{u_2,w_1}(X) + Q_{u_2,w_2}(X))\cdot\\
&\qquad\qquad \cdot (Q_{u_4,w_1}(X) + Q_{u_4,w_2}(X))(Q_{u_5,w_4}(X) + Q_{u_5,w_5}(X))\\
&Q_{u_3,w_4}(X) = \mathbf{z}_S(u_3,w_4)x_{w_4}Q_{u_2,w_3}(X)Q_{u_4,w_3}(X)Q_{u_5,w_6}(X)\\
&Q_{u_3,w_5}(X) = \mathbf{z}_S(u_3,w_5)x_{w_5}Q_{u_2,w_3}(X)Q_{u_4,w_3}(X)Q_{u_5,w_6}(X)\\
&Q_{u_3,w_6}(X) = 0.
\end{aligned}$$

The last vertex to analyze is u_5, for which is $N^+_T(u_5) = \emptyset$. For every $w \in V(D)$ we have

$$\begin{aligned}
Q_{u_5,w}(X) &= \mathbf{z}_S(u_5,w)x_w Q^-_{u_5,w}(X) =\\
&= \mathbf{z}_S(u_5,w)x_w \prod_{v\in N^-_T(u_5)} \left(\sum_{(w,w')\in A(D)} Q_{v,w'}(X) \right) =\\
&= \mathbf{z}_S(u_5,w)x_w \sum_{(w,w')\in A(D)} Q_{u_6,w'}(X),
\end{aligned}$$

which leads to

$$\begin{aligned}
&Q_{u_5,w_1}(X) = \mathbf{z}_S(u_5,w_1)x_{w_1}Q_{u_6,w_3}(X) = \mathbf{z}_S(u_5,w_1)x_{w_1}\mathbf{z}_S(u_6,w_3)x_{w_3}\\
&Q_{u_5,w_2}(X) = \mathbf{z}_S(u_5,w_2)x_{w_2}Q_{u_6,w_3}(X) = \mathbf{z}_S(u_5,w_2)x_{w_2}\mathbf{z}_S(u_6,w_3)x_{w_3}\\
&Q_{u_5,w_3}(X) = \mathbf{z}_S(u_5,w_3)x_{w_3}\left(Q_{u_6,w_4}(X) + Q_{u_6,w_5}(X)\right) =\\
&\qquad = \mathbf{z}_S(u_5,w_3)x_{w_3}\left(\mathbf{z}_S(u_6,w_4)x_{w_4} + \mathbf{z}_S(u_6,w_5)x_{w_5}\right)\\
&Q_{u_5,w_4}(X) = \mathbf{z}_S(u_5,w_4)x_{w_4}Q_{u_6,w_6}(X) = \mathbf{z}_S(u_5,w_4)x_{w_4}\mathbf{z}_S(u_6,w_6)x_{w_6}\\
&Q_{u_5,w_5}(X) = \mathbf{z}_S(u_5,w_5)x_{w_5}Q_{u_6,w_6}(X) = \mathbf{z}_S(u_5,w_5)x_{w_5}\mathbf{z}_S(u_6,w_6)x_{w_6}\\
&Q_{u_5,w_6}(X) = 0.
\end{aligned}$$

Going now 'backwards' we compute the missing values for u_3:

$$\begin{aligned}
Q_{u_3,w_3}(X) &= \mathbf{z}_S(u_3,w_3)x_{w_3}\mathbf{z}_S(u_6,w_6)x_{w_6}(\mathbf{z}_S(u_2,w_1)x_{w_1}+\mathbf{z}_S(u_2,w_2)x_{w_2})\cdot\\
&\quad\cdot(\mathbf{z}_S(u_4,w_1)x_{w_1}+\mathbf{z}_S(u_4,w_2)x_{w_2})(\mathbf{z}_S(u_5,w_4)x_{w_4}+\mathbf{z}_S(u_5,w_5)x_{w_5})\\
Q_{u_3,w_4}(X) &= \mathbf{z}_S(u_3,w_4)x_{w_4}\mathbf{z}_S(z_2,w_3)x_{w_3}\mathbf{z}_S(z_4,w_3)x_{w_3}\cdot 0 = 0\\
Q_{u_3,w_5}(X) &= \mathbf{z}_S(u_3,w_5)x_{w_5}\mathbf{z}_S(z_2,w_3)x_{w_3}\mathbf{z}_S(z_4,w_3)x_{w_3}\cdot 0 = 0.
\end{aligned}$$

And for u_1:

$$\begin{aligned}
Q_{u_1,w_1}(X) &= \mathbf{z}_S(u_1,w_1)x_{w_1}\mathbf{z}_S(u_3,w_3)x_{w_3}\mathbf{z}_S(u_6,w_6)x_{w_6}\cdot\\
&\quad\cdot(\mathbf{z}_S(u_2,w_1)x_{w_1}+\mathbf{z}_S(u_2,w_2)x_{w_2})(\mathbf{z}_S(u_4,w_1)x_{w_1}+\mathbf{z}_S(u_4,w_2)x_{w_2})\cdot\\
&\quad\cdot(\mathbf{z}_S(u_5,w_4)x_{w_4}+\mathbf{z}_S(u_5,w_5)x_{w_5})\\
Q_{u_1,w_2}(X) &= \mathbf{z}_S(u_1,w_2)x_{w_2}\mathbf{z}_S(u_3,w_3)x_{w_3}\mathbf{z}_S(u_6,w_6)x_{w_6}\cdot\\
&\quad\cdot(\mathbf{z}_S(u_2,w_1)x_{w_1}+\mathbf{z}_S(u_2,w_2)x_{w_2})(\mathbf{z}_S(u_4,w_1)x_{w_1}+\mathbf{z}_S(u_4,w_2)x_{w_2})\cdot\\
&\quad\cdot(\mathbf{z}_S(u_5,w_4)x_{w_4}+\mathbf{z}_S(u_5,w_5)x_{w_5})\\
Q_{u_1,w_3}(X) &= \mathbf{z}_S(w_3)x_{w_3}\cdot 0 = 0\\
Q_{u_1,w_4}(X) &= 0\\
Q_{u_1,w_5}(X) &= 0.
\end{aligned}$$

Finally, for $\mathcal{S} = \{w_1, w_4, w_6\}$ we have

$$\begin{aligned}
Q(X,T) = \sum_{w\in V(D)} Q_{u_1,w}(X) &= z^2x_{w_1}x_{w_3}x_{w_6}(zx_{w_1}+x_{w_2})^2(zx_{w_4}+x_{w_5})+\\
&+ zx_{w_2}x_{w_3}x_{w_6}(zx_{w_1}+x_{w_2})^2(zx_{w_4}+x_{w_5}). \qquad (6.7)
\end{aligned}$$

Observe that $Q(X,T)$ does not contain monomials of the form $z^3b(X)$, which means that the given tree T can not be $\mathcal{S}$-embedded into D.

6.3 The Link Up Problem is Hard

In this section, we prove that the Link Up problem is NP-complete by describing a polynomial-time reduction from the Set Cover problem.

Let $\mathcal{U} = \{u_1, \ldots, u_n\}$ be a set of n items and let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a family of m sets containing the items in $\mathcal{U}$, i.e., each $S_t \subseteq \mathcal{U}$, such that each element in $\mathcal{U}$ belongs to at least one set from $\mathcal{S}$. A k-element subset of $\mathcal{S}$, whose union is equal to the whole universe $\mathcal{U}$, is called *a set cover of size k.*

The Set Cover Problem (SC)

Given $\mathcal{U}, \mathcal{S}$ and a positive integer k, does there exist a set cover of a size k?

The Set Cover problem is well known to be NP-complete [87]. We are going to prove that for a given $\mathcal{U} = \{u_1, \ldots, u_n\}$, $\mathcal{S} = \{S_1, \ldots, S_m\}$ and an integer k, there exists a set cover of size k if and only if there is a solution for the LU problem in the appropriately constructed acyclic digraph $D_{SC} = (V, A, F, B)$. Basically, in this graph, for each element $u \in \mathcal{U}$, there is a gadget C_u being the union of the number of 'vertical' paths equal to the number of sets that u appears in. Also, there is one 'spanning' gadget including m 'horizontal' paths PH_t, $t \in \{1, \ldots, m\}$, each of which visits the relevant gadget C_u if the element u belongs to the set S_t. In our construction, all vertices are facilities, and snow teams are located only at source vertices, one team at each source vertex (see details below).

In the following, we assume $\mathcal{U} = \{1, \ldots, n\}$ and that elements in $S_t = \{x_1, \ldots, x_{\ell(t)}\}$ are sorted in the ascending order, where $\ell(t)$ denotes the size of S_t, $t \in \{1, \ldots, m\}$. Also, we denote a solution to the set cover problem with the input $\langle \mathcal{U}, \mathcal{S}, k \rangle$ by $\mathcal{X}^I$ which is encoded as a subset of $\{1, \ldots, m\}$, where $t \in \mathcal{X}^I$ if and only if S_t belongs to the k-element subset of $\mathcal{S}$ forming the set cover. Finally, for simplicity of presentation, we denote a set $\{1, \ldots, l\}$ of indices by $[l]$.

Digraph construction. For $i \in \mathcal{U}$, let I_i be the set of all indices of subsets from $\mathcal{S}$ which contain i: $I_i = \{j \mid i \in S_j\}$. First, for every $i \in \mathcal{U}$, we introduce the following four sets of vertices (see Figure 6.6 for an illustration):

- $U_i = \{u_i\} \cup \{\, u_{i,j} : j \in I_i \,\}$,
- $U'_i = \left\{\, u'_{i,j} : j \in I_i \right\}$,
- $V_i = \{\, v_{i,j} : j \in I_i \,\}$,
- $V'_i = \left\{\, v'_{i,j} : j \in I_i \right\}$.

Next, we introduce an additional set $Z = \{z, z_1, \ldots, z_k\}$ of vertices corresponding to the input integer k. Finally, for every $i \in \mathcal{U}$ and $j \in I_i$, we create a directed path $PV_{i,j} = (u_{i,j}, u_{i,}, u'_{i,j}, v_{i,j}, v'_{i,j})$; we refer to these paths as *vertical*. Observe that for $i \in \mathcal{U}$, the union $\bigcup_{j \in I_i} V(PV_{i,j})$ induces the directed subgraph which we shall refer to as the *i-th element component* C_i. This finishes the first step of our construction.

For the second step (see Figure 6.7 for an illustration), for $t \in [m]$, we consider the set $S_t = \{x_1, \ldots, x_{\ell(t)}\}$ — recall that the elements in S_t

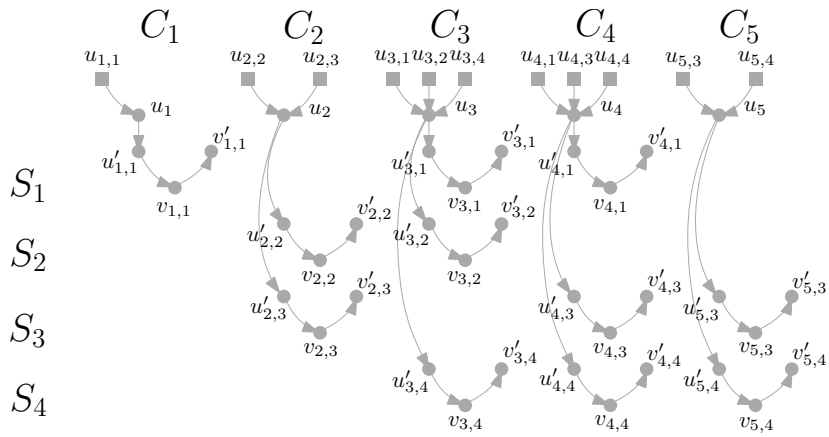


FIGURE 6.6: Step 1: construction of the element components. Here, $\mathcal{U} = \{1,2,3,4,5\}$ and $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$, where $S_1 = \{1,3,4\}$, $S_2 = \{2,3\}$, $S_3 = \{2,4,5\}$ and $S_4 = \{3,4,5\}$. The corresponding sets of indices are: $I_1 = \{1\}$, $I_2 = \{2,3\}$, $I_3 = \{1,2,4\}$, $I_4 = \{1,3,4\}$ and $I_5 = \{3,4\}$. For $k = 2$, there exists a set cover $\mathcal{X}^I = \{1,3\}$ of size 2, consisting of the sets S_1 and S_3.

are sorted in the ascending order — and construct a directed path $PH_t = (z, v_{x_1,t}, v_{x_2,t}, \ldots, v_{x_{\ell(t)},t})$; these paths are called *horizontal*. Next, we add k additional arcs, namely, for each $l \in [k]$, we add the arc (z_l, z).

With our 2-step construction, we have built the directed graph $D_{\mathrm{SC}} = (V, A)$, where

$$V = \bigcup_{i\in[n]} V(C_i) \cup Z$$
$$A = \big\{(z_i, z) \;\big|\; i \in [k]\big\} \cup \bigcup_{i\in[n]} E(C_i) \cup \bigcup_{i\in[m]} E(PH_i).$$

We finalize our construction by defining the subset F of V and the function $B\colon V \to \mathbb{N}$. Specifically, we set $F = V$, and $B(v) = 1$ if and only if $v \in V$ is a source vertex in D_{SC}:

$$B(v) = \begin{cases} 1, & \text{if } v \in \big\{u_{i,j} \;\big|\; i \in \mathcal{U}, j \in I_i\big\} \cup \{z_1, \ldots, z_k\} \\ 0, & \text{otherwise.} \end{cases}$$

Therefore, $F = V$ and $\mathcal{B} = B^{-1}(\mathbb{N}^+)$ is the set of all source vertices in D_{SC}. In particular, there is exactly one agent at each source vertex of D_{SC}, and so $k = \sum_{v\in V} B(v) = |s(D_{\mathrm{SC}})|$, which equals to $k + \sum_{i=1}^{m} |S_i|$ by the

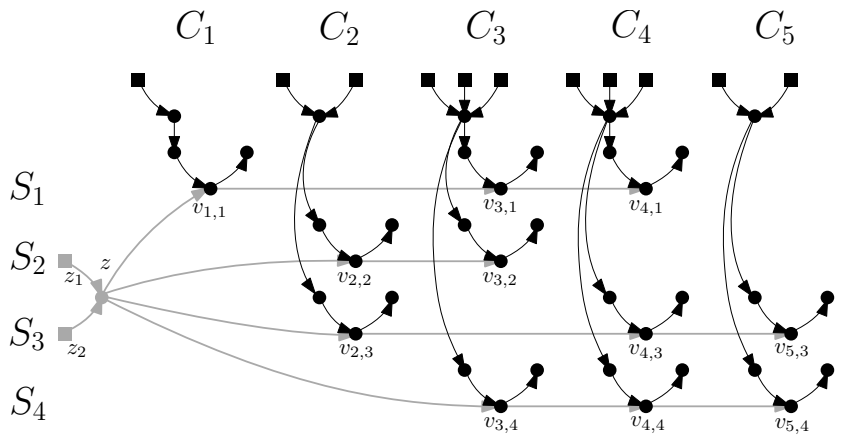


FIGURE 6.7: (Cont. Figure 6.6) Step 2: four horizontal paths connecting all components and two new source vertices z_1, z_2 are added (recall $k = 2$).

construction of D_{SC}.

Clearly, our reduction takes polynomial time. The order of D_{SC} is equal to $4 \cdot \sum_{i=1}^{m} |S_i| + n + k + 1 = O(nm + k)$, its size also equals $O(nm + k)$, and the descriptions of the set F and the function B require $O(nm + k)$ space either. Finally, observe that D_{SC} is acyclic and its underlying graph is connected. The latter observation follows from the fact that $\mathcal{S}$ is a family of sets whose union is $\mathcal{U}$, and each element in $\mathcal{U}$ belongs to at least one set from $\mathcal{S}$.

6.3.1 Direct Implication

First, we are going to prove the direct implication.

Lemma 6.3.1. *Let $\langle \mathcal{U}, \mathcal{S}, k \rangle$ be an instance of the* SC *problem. If there exists a set cover of size k for $\mathcal{U}$ and $\mathcal{S}$, then there exists a solution to the* LU *problem for the digraph $D_{\text{SC}} = (V, A, F, B)$.*

Proof. (See Figure 6.8 for an illustration.) Let $\mathcal{X}^I = \{\xi(1), \ldots, \xi(k)\}$ be a solution to the SC problem for $\langle \mathcal{U}, \mathcal{S}, k \rangle$. We now construct a solution $\mathcal{W}$ to the LU problem to consist of the following paths:

$$PV_{i,j} = (u_{i,j}, u_{i,}, u'_{i,j}, v_{i,j}, v'_{i,j}), \quad \text{for } i \in \mathcal{U},\ j \in I_i,$$

and

$$(z_t, z) \circ PH_{\xi(t)} = (z_t, z, v_{x_1,\xi(t)}, v_{x_2,\xi(t)}, \ldots, v_{x_{\ell(t)},\xi(t)}), \quad \text{for } t \in [k],$$

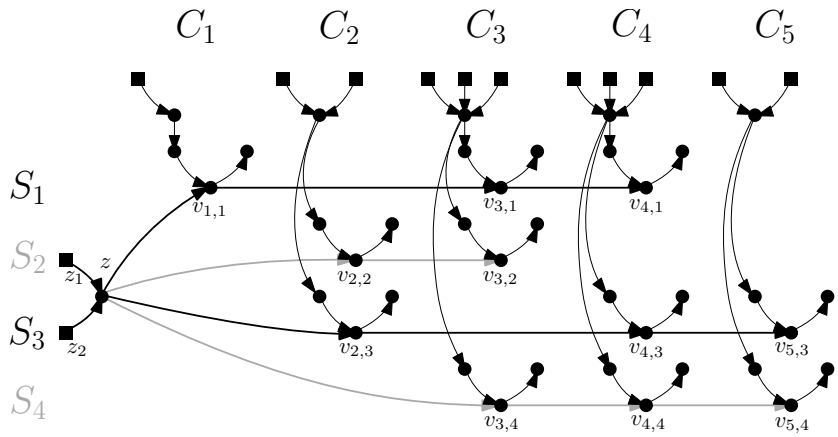


FIGURE 6.8: (Cont. Figure 6.7) The fact that the set cover contains all elements in $\mathcal{U}$ guarantees that the corresponding subgraph H of D is connected. For $k = 2$, there exists a set cover $\mathcal{X}^I = \{1, 3\}$ of size 2, corresponding to sets S_1 and S_3, respectively.

where $x_1, \ldots, x_{\ell(t)}$ are the (ordered) elements of the set $S_t \in \mathcal{S}$, with $|S_t| = \ell(t)$.

Clearly, by the construction, the set $\mathcal{W}$ consists of $k = |s(D_{SC})|$ paths that together cover all vertices of D_{SC} and each of which starts at a distinct vertex in $\mathcal{B} = B^{-1}(\mathbb{N}^+)$. Thus, it remains to prove that these paths induce a subgraph H of D_{SC} whose underlying graph is connected. First, by the definition of the paths $PV_{i,j}$, observe that each element component induces a subgraph whose underlying graph is connected. Hence, it is enough to argue that for each element component C_i, where $i \in \mathcal{U}$, there exists a directed path in H that connects z with a vertex of C_i. Suppose for a contradiction that this is not the case for the i-th element component C_i, for some $i \in \mathcal{U}$. That means that no vertex $v_{i,j}$, for any $j \in I_i$, is lying on any of the chosen horizontal paths in H. But that means, by the choice of the horizontal paths and the fact that $\mathcal{X}^I$ is a solution to the SC problem, that i does not belong to $\bigcup_{j \in \mathcal{X}^I} S_j$, a contradiction. □

6.3.2 Converse implication

In this subsection, to complete the NP-completeness proof of the LU problem, we are going to prove the converse implication, in a sequence of lemmas. Recall that in our digraph $D_{SC} = (V, A, F, B)$, we have

$$s(D_{SC}) = \{z_1, \ldots, z_k\} \cup \bigcup_{i \in \mathcal{U}} \{u_{i,j} \mid j \in I_i\},$$

and we set $B(v) = 1$ for each $v \in s(D_{SC})$ and $B(v) = 0$ otherwise, and so $k = |s(D_{SC})|$. Thus, keeping in mind that D_{SC} is acyclic, any solution $\mathcal{W}$ to the LU problem for D_{SC} consists of k paths that start at distinct source vertices in $s(D_{SC})$; in the following, $\pi(v) \in \mathcal{W}$ denotes the unique path of $\mathcal{W}$ that starts at a source vertex $v \in s(D_{SC})$.

Lemma 6.3.2. *Suppose that the* LU *problem admits a positive answer for the input graph D_{SC}. Then, there exists a solution $\{\pi(v) \mid v \in s(D_{SC})\}$ to the* LU *problem for D_{SC} such that for each $i \in \mathcal{U}$ and $j \in I_i$, $V(\pi(u_{i,j})) \subseteq V(C_i)$.*

Proof. Let $\mathcal{W}$ be a solution to the LU problem for D_{SC} and assume that in $\mathcal{W}$, for some $i \in \mathcal{U}$ and $j \in I_i$, we have $\pi(u_{i,j}) = (u_{i,j}, u_i, u'_{i,j'}, v_{i,j'}, v_{i',j'}) \circ P$ for some (possibly empty) path P, that is, the arc $(v_{i,j'}, v_{i',j'}) \in A(\pi(u_{i,j}))$ and so $\pi(u_{i,j})$ 'leaves' C_i at $v_{i,j'}$ by visiting vertex $v_{i',j'}$, for some $i' > i$. We will argue that we may obtain another solution to the LU problem in which $V(\pi(u_{i,j})) \subseteq V(C_i)$ as required by the lemma. The idea is to modify two paths in $\mathcal{W}$ maintaining the following invariant: each path that is a subgraph of an element component remains a subgraph of this element component. Thus, since this modification can be repeated for any $i \in \mathcal{U}$ and $j \in I_i$, this is sufficient to prove our claim.

Since $F = V$, we have $v'_{i,j'} \in F$ and hence there exists another path $\pi(v) \in \mathcal{W}$ for some $v \in s(D_{SC})$ such that $\pi(v) = P' \circ (v_{i,j'}, v'_{i,j'})$, for some path P' in D_{SC}. We then modify the set $\mathcal{W}$ of paths by removing the two paths $\pi(u_{i,j})$ and $\pi(v)$, and then adding the following two paths: $(u_{i,j}, u_i, u'_{i,j'}, v_{i,j'}, v'_{i,j'})$ and $P' \circ (v_{i,j'}, v_{i',j'}) \circ P$.

Observe that any two paths that start at vertices in $U_i \setminus \{u_i\}$ have only the vertex u_i in common — this is due to the fact that the vertices in U'_i are only reachable with paths that start at vertices in $U_i \setminus \{u_i\}$ and $|U'_i| = |U_i \setminus \{u_i\}| = |B^{-1}(1) \cap V(C_i)|$. Consequently, the vertex set $V(\pi(v))$ of the original path $\pi(v)$ is not a subset of a single element component, and hence, after the modification, each path whose vertex set is a subset of an element component keeps this property as required. Clearly, all vertices of D are covered in the new solution and, since the two new paths also share

the vertex $v_{i,j'}$, the modified set of paths also induces a connected spanning subgraph of the underlying graph of D. □

Lemma 6.3.3. *Suppose that the* LU *problem admits a positive answer for the input graph D_{SC}. Then, there exists a solution $\{\pi(v) \mid v \in s(D_{SC})\}$ to the* LU *problem for D_{SC} such that:*

a) for each $i \in \mathcal{U}$ and $j \in I_i$, we have $\pi(u_{i,j}) = PV_{i,j}$;

b) for each $t \in [k]$, we have $\pi(z_t) = (z_t, z) \circ PH_l$ for some $l \in [m]$.

Before we proceed with the proof of Lemma 6.3.3, note that Property (a) implies that the paths $\pi(u_{i,j})$, where $i \in \mathcal{U}$ and $j \in I_i$, visit all and only vertices of all components C_i, $i \in \mathcal{U}$. Therefore, since these components have no vertices in common, the underlying graph becomes connected only thanks to the paths $\pi(v)$, $v \in Z \setminus \{z\}$, which Property (b) refers to.

Proof. (a) Consider any solution, say $\mathcal{W}$, to the LU problem for D_{SC} and assume that in $\mathcal{W}$, for some $i \in \mathcal{U}$ and $j \in I_i$, we have $\pi(u_{i,j}) \neq PV_{i,j}$; we shall refer to $\pi(u_{i,j})$ as well as to any other path $\pi(u_{i',j'})$ such that $\pi(u_{i',j'}) \neq PV_{i',j'}$ as *inconsistent*. By Lemma 6.3.2, $\pi(u_{i,j}) \subseteq V(C_i)$ and hence there exists $j' \in I_i \setminus \{j\}$ such that $\pi(u_{i,j}) = (u_{i,j}, u_i, u'_{i,j'}, v_{i,j'}, v'_{i,j'})$. Then, again by Lemma 6.3.2 and the fact that each vertex in U'_i (in particular $u'_{i,j}$) has to belong to some path $\pi(v) \in \mathcal{W}$, $v \in U_i \setminus \{u\}$, there exists in $\mathcal{W}$ another inconsistent path $\pi(u_{i,j''}) = (u_{i,j''}, u_i, u'_{i,j}, v_{i,j}, v'_{i,j})$ for some $j'' \in I_i \setminus \{j\}$. Now, we modify the solution by substituting $\pi(u_{i,j}) := PV_{i,j}$ and $\pi(u_{i,j''}) := (u_{i,j''}, u_i, u'_{i,j'}, v_{i,j'}, v'_{i,j'})$. Clearly, the two new paths still share vertex u_i and cover exactly the same vertices as the two original ones. Therefore, the modified set of paths is also a solution to the LU problem, moreover, with less number of inconsistent paths.

By repeating the above replacement argument a finite number of times, if ever needed, we obtain the desired solution satisfying Property (a).

(b) Consider any solution, say $\mathcal{W}$, to the LU problem for D_{SC} satisfying already proved Property (a). Consider any $t \in [k]$. If $\pi(z_t) \in \mathcal{W}$ ends at a vertex of some horizontal path PH_l for some $l \in [m]$ and $\pi(z_t) \neq (z_t, z) \circ PH_l$, then we just extend $\pi(z_t)$ to have $\pi(z_t) = (z_t, z) \circ PH_l$. Otherwise, by the construction of D_{SC}, we must have $\pi(z_t) = (z_t, z) \circ P \circ (v_{i,l}, v'_{i,l})$ for some $i \in \mathcal{U}$, where P is a subpath of PH_l for some $l \in [m]$. By the choice of $\mathcal{W}$, the arc $(v_{i,l}, v'_{i,l})$ belongs to the (consistent) path $\pi(u_{i,j}) = PV_{i,j}$ for some $j \in I_i$, and hence the path $\pi(z_t)$ can be replaced by: $\pi(z_t) := (z_t, z) \circ PH_l$. Since P is a subpath of the new path $\pi(z_t)$, we conclude that the new set

of paths is also a solution to the LU problem for D_{SC}, and moreover, it maintains the property that $\pi(u_{i,j}) = PV_{i,j}$ for each $i \in \mathcal{U}$ and $j \in I_i$.

Therefore, by repeating the above replacement argument a finite number of times, if ever needed, we obtain the desired solution satisfying both Properties (a) and (b) □

Now, we are going to prove our final lemma.

Lemma 6.3.4. *If there is a solution to the* LU *problem for* $D_{SC} = (V, A, F, B)$*, then there exists a set cover of size* k *for the set system* $(\mathcal{U}, \mathcal{S})$*.*

Proof. By Lemma 6.3.3, any solution to the LU problem can be modified to be composed of the following paths: $\pi(u_{i,j}) = PV_{i,j}$ for each $i \in \mathcal{U}$ and $j \in I_i$, and $\pi(z_t) = (z_t, z) \circ PH_{\xi(t)}$ for each $t \in [k]$, where $\xi(t) \in [m]$. Now, we claim that the set $\mathcal{X}^I = \{\xi(1), \ldots, \xi(k)\}$ is a set cover solution for the instance $\langle \mathcal{U}, \mathcal{S}, k \rangle$. Indeed, since our solution to the LU problem is valid, for each $i \in \mathcal{U}$ there exists $t \in [k]$ such that $v_{i,j}$ is a vertex of $\pi(z_t)$, since otherwise, in the underlying simple graph induced by our solution, no vertex in C_i is connected by a path to the vertex z. Thus, $i \in S_{\xi(t)}$ which completes the proof. □

Note that the LU problem is clearly in NP and, as already observed, the construction of D_{SC} is polynomial in the input size to the SC problem. Hence, by combining Lemmas 6.3.1 and 6.3.4, we obtain the following result.

Theorem 6.3.1. *The* LU *problem is strongly NP-complete even for directed acyclic graphs* $D = (V, A, F, B)$ *with* $F = V$ *and* $B(v) = 1$ *if* v *is a source vertex in* D *and* $B(v) = 0$ *otherwise.* □

No pre-specified positions of agents. We claim that the Link Up problem with Unspecified bases is also NP-complete. The reduction is exactly the same as for the LU problem. All we need is to observe that if facilities are located at all vertices of the input digraph, then the number of agents sufficient to solve the LUU problem is bounded from below by the number of source vertices in the digraph, since there must be at least one agent at each of its source vertices. Furthermore, without loss of generality we may assume that in any feasible solution of k walks, all agents are initially located at source vertices. Since in the digraph D_{SC} constructed for the proof of Theorem 6.3.1, we have $B(v) = 1$ if v is a source vertex in D_{SC} and $B(v) = 0$ otherwise, we may conclude with the following corollary.

Corollary 6.3.1. *The* LUU *problem is strongly NP-complete even for directed acyclic graphs $D = (V, A, F)$ with $F = V$ and k being equal to the number of source vertices in D.* □

Since by setting $F = V$, the LUU problem becomes just the Agent Clearing Tree problem (ACT) studied in [110], we immediately obtain the following corollary resolving the open problem of the complexity status of ACT posed in [110].

Corollary 6.3.2. *The* ACT *problem is NP-complete.* □

6.4 Conclusions

In this chapter we have proposed a new group of part-exploration problems on digraphs, called Link Up problems, and shown that they are fixed-parameter tractable. In particular, we prove that the LU problem admits a fixed-parameter randomized algorithm with respect to the total number l of facilities and snow team bases, running in $2^{O(l)} \cdot \text{poly}(n)$ time, where $\text{poly}(n)$ is a polynomial in the order n of the input graph. The proof relies on the algebraic framework introduced by Koutis in [103]. On the other hand, we show that the LU problem (as well as some of its variants) is NP-complete, by a reduction from the Set Cover problem [87]. Our result on NP-completeness of the LU problem implies NP-completeness of the Agent Clearing Tree problem studied in [110], where the complexity status of the latter has been posed as an open problem.

In all of our variants of the Link Up problem, we assumed that a agents can traverse arbitrary number of arcs. However, from a practical point of view, it is more natural to assume that each agent, called an *s-agent*, can traverse and clear only the fixed number s of arcs [123]. Observe that in this case, the key Lemma 6.2.2 does not hold, which immediately makes our algebraic approach unfeasible for the Link Up problem with s-agents, so this variant requires further studies.

Chapter 7

Conclusions

In this thesis we have discussed the distributed monotone contiguous decontamination problem, the on-line collaborative exploration and the partial exploration of digraphs. The most important results derived in this thesis are listed in Table 7.1. The open problems and the directions for further research have been described at the end of every chapter. Apart from the presented results, the author of this thesis currently is investigating the relationship between the domination and searching problems. We present below a short overview of the topic.

Searching and Domination. For any graph $G = (V, E)$ a set of vertices $D \subseteq V$ is a *dominating set* if every $v \in V \setminus D$ is adjacent to at least one vertex from D. The *domination number* $\gamma(G)$ is the size of the smallest dominating set for G. For a given graph G and integer number k the problem of deciding whether $\gamma(G) \leq k$ is NP-complete [89]. In literature can be found many different types of domination. The author of this thesis has recently presented results about the *twin domination* number of tournaments [130] and the relationship between the *connected* and *convex domination* numbers [54].

The *cops and robber* game is a type of a searching game played by two players, one controlling cops (agents) and the other the robber (fugitive). Firstly, players place cops and the robber on the chosen vertices of a graph. Then, cops and the robber alternate in turns, by moving to a distance at most one each. The robber cannot move to a vertex occupied by a cop. The goal of agents is to *catch* the robber, i.e., to occupy the same vertex at the same time. On the other hand, we say that the robber *wins* if it can avoid being caught forever. The problem has been defined in 1983 by Winkler and Nowakowski [122] for one cop. Computing the minimum number of cops for which is possible to catch a robber on a given graph is NP-hard [79].

MAIN RESULTS OF THIS THESIS	
Results	Sec.
SEARCHING UNKNOWN PARTIAL GRIDS OF ORDER n	
Construction of the distributed algorithm, that for any partial grid of order n computes a connected and monotone searching strategy with the use of $O(\sqrt{n})$ searchers.	3
We give a lower bound of the competitive ratio of $\Omega(\sqrt{n}/\log n)$.	3.3
COMPUTING CONNECTED PATHWIDTH	
Verification if a connected pathwidth of a given graph is at most k, for a fixed k, can be done in polynomial time.	4
COST-OPTIMAL EXPLORATION	
Construction of the cost-optimal off-line algorithm for rings.	5.2
Construction of the cost-optimal off-line algorithm for trees.	5.4
Construction of the 2-competitive on-line algorithm for rings.	5.3
We give lower bounds of the competitive ratio of $3/2$ (for rings)	5.3
and 2 (for trees) for any on-line algorithm.	5.5
LINK UP PROBLEMS	
Link Up problem is FPT.	6.2
Link Up problem is NP-hard.	6.3

TABLE 7.1: List of the main results obtained in this thesis.

The choice of cops' starting positions is very often crucial in order for the first player to win. Thus, the cops and robber game (called also sometimes as a *domination* game [106]) is an example of the existing connection between searching and domination problems. In particular, it can be noticed for the *one-tick cops and k-robbers* game, where cops win only if they catch k robbers in their first move [2]. See also, e.g., [49], where authors present the results for a game with infinitely fast robber and *defensive* domination number or [106], where the link between *d-distance* domination and the game, where cops have *d-visibility* is investigated. Currently, the author of this thesis is studying the connection between the twin domination number on digraphs and the one-tick cops and k-robbers game on digraphs.

The contributions of the author of this thesis to the obtained results are as follows:

- A leading role in developing the algorithm `ModGridSearching` together with its analysis (Sections 3.4-3.6) and writing down the work.
- The algorithm from Chapter 4 and Theorem 4.4.1 have been developed in the course of joint discussions, but the author had a leading role in the analysis of the algorithm (Section 4.4), in particular she is the author of Lemmas 4.4.3-4.4.8.
- All results presented in Chapter 5 are authors independent results.
- A proof of NP-completeness of the LU problem (Section 6.3) is a joint work with Dariusz Dereniowski.

At the end, let us notice that the results that have been obtained in this thesis required using several algorithmic techniques. Those in the area of distributed algorithms include construction of lower bounds (Theorem 5.3.1, Theorem 5.5.1) and amortized analysis (Theorem 3.5.1, Theorem 3.6.1, Lemma 5.3.1). The analysis of off-line algorithms involved e.g. theory of NP-completeness (Theorem 6.3.1), theory of parametrized complexity (Theorem 4.4.1) and analysis of the centralized algorithms (Theorem 5.4.1).

Bibliography

[1] Ahmad Abdi et al. "Lehman's Theorem and the Directed Steiner Tree Problem". In: *SIAM Journal on Discrete Mathematics* 30.1 (2016), pp. 141–153.

[2] Brian Alspach et al. "Time constrained graph searching". In: *Theoretical Computer Science* 399.3 (2008), pp. 158–168.

[3] Yaniv Altshuler et al. "Multi-agent cooperative cleaning of expanding domains". In: *The International Journal of Robotics Research* 30.8 (2011), pp. 1037–1071.

[4] Eyal Amir. "Approximation algorithms for treewidth". In: *Algorithmica* 56.4 (2010), pp. 448–479.

[5] Baruch Awerbuch et al. "Piecemeal graph exploration by a mobile robot". In: *Information and Computation* 152.2 (1999), pp. 155–172.

[6] Evangelos Bampas et al. "Maximal exploration of trees with energy-constrained agents". In: *arXiv preprint arXiv:1802.06636* (2018).

[7] Lali Barrière et al. "Capture of an intruder by mobile agents". In: *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2002, pp. 200–209.

[8] Lali Barriere et al. "Connected and internal graph searching". In: *29th Workshop on Graph Theoretic Concepts in Computer Science (WG), Springer-Verlag, LNCS*. Vol. 2880. 2003, pp. 34–45.

[9] Lali Barrière et al. "Connected graph searching". In: *Information and Computation* 219 (2012), pp. 1–16.

[10] Lali Barrière et al. "Searching is not jumping". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 2003, pp. 34–45.

[11] Niko Beerenwinkel et al. "Covering pairs in directed acyclic graphs". In: *The Computer Journal* 58.7 (2014), pp. 1673–1686.

[12] Micah J Best et al. "Contraction obstructions for connected graph searching". In: *Discrete Applied Mathematics* 209 (2016), pp. 27–47.

[13] Margrit Betke, Ronald L Rivest, and Mona Singh. "Piecemeal learning of an unknown environment". In: *Machine Learning* 18.2-3 (1995), pp. 231–254.

[14] Deepak Bhadauria et al. "Capturing an evader in polygonal environments with obstacles: The full visibility case". In: *The International Journal of Robotics Research* 31.10 (2012), pp. 1176–1189.

[15] Therese Biedl et al. "Using ILP/SAT to determine pathwidth, visibility representations, and other grid-based graph drawings". In: *International Symposium on Graph Drawing*. Springer. 2013, pp. 460–471.

[16] Daniel Bienstock and Paul Seymour. "Monotonicity in graph searching". In: *Journal of Algorithms* 12.2 (1991), pp. 239–245.

[17] Andreas Björklund, Thore Husfeldt, and Nina Taslaman. "Shortest cycle through specified elements". In: *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM. 2012, pp. 1747–1753.

[18] Andreas Björklund, Petteri Kaski, and Łukasz Kowalik. "Constrained multilinear detection and generalized graph motifs". In: *Algorithmica* 74.2 (2016), pp. 947–967.

[19] Andreas Björklund et al. "Narrow sieves for parameterized paths and packings". In: *Journal of Computer and System Sciences* 87 (2017), pp. 119–139.

[20] Lélia Blin, Janna Burman, and Nicolas Nisse. "Exclusive graph searching". In: *Algorithmica* 77.3 (2017), pp. 942–969.

[21] Lélia Blin, Janna Burman, and Nicolas Nisse. "Perpetual graph searching". PhD thesis. INRIA, 2012.

[22] Lélia Blin et al. "Distributed chasing of network intruders". In: *Theoretical Computer Science* 399.1-2 (2008), pp. 12–37.

[23] Hans L Bodlaender. "A linear-time algorithm for finding tree-decompositions of small treewidth". In: *SIAM Journal on computing* 25.6 (1996), pp. 1305–1317.

[24] Hans L Bodlaender and Ton Kloks. "Efficient and constructive algorithms for the pathwidth and treewidth of graphs". In: *Journal of Algorithms* 21.2 (1996), pp. 358–402.

[25] Hans L Bodlaender et al. "A cˆkn 5-Approximation Algorithm for Treewidth". In: *SIAM Journal on Computing* 45.2 (2016), pp. 317–378.

[26] Hans L Bodlaender et al. "A note on exact algorithms for vertex ordering problems on graphs". In: *Theory of Computing Systems* 50.3 (2012), pp. 420–432.

[27] Piotr Borowiecki, Dariusz Dereniowski, and Łukasz Kuszner. "Distributed graph searching with a sense of direction". In: *Distributed Computing* 28.3 (2015), pp. 155–170.

[28] Piotr Borowiecki, Dariusz Dereniowski, and Paweł Prałat. "Brushing with additional cleaning restrictions". In: *Theoretical Computer Science* 557 (2014), pp. 76–86.

[29] Vincent Bouchitté and Ioan Todinca. "Treewidth and minimum fill-in: Grouping the minimal separators". In: *SIAM Journal on Computing* 31.1 (2001), pp. 212–232.

[30] Peter Brass, Ivo Vigan, and Ning Xu. "Improved analysis of a multi-robot graph exploration strategy". In: *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*. IEEE. 2014, pp. 1906–1910.

[31] Peter Brass et al. "Multirobot tree and graph exploration". In: *IEEE Transactions on Robotics* 27.4 (2011), pp. 707–717.

[32] Darryn Bryant et al. "Brushing without capacity restrictions". In: *Discrete Applied Mathematics* 170 (2014), pp. 33–45.

[33] Moses Charikar et al. "Approximation algorithms for directed Steiner problems". In: *Journal of Algorithms* 33.1 (1999), pp. 73–91.

[34] Rajesh Chitnis et al. "A tight algorithm for strongly connected steiner subgraph on two terminals with demands". In: *Algorithmica* 77.4 (2017), pp. 1216–1239.

[35] Marek Chrobak and Claire Kenyon-Mathieu. "SIGACT news online algorithms column 10: competitiveness via doubling". In: *ACM SIGACT News* 37.4 (2006), pp. 115–126.

[36] Timothy H Chung, Geoffrey A Hollinger, and Volkan Isler. "Search and pursuit-evasion in mobile robotics". In: *Autonomous robots* 31.4 (2011), p. 299.

[37] David Coudert. "A note on Integer Linear Programming formulations for linear ordering problems on graphs". PhD thesis. Inria; I3S; Universite Nice Sophia Antipolis; CNRS, 2016.

[38] David Coudert, Dorian Mazauric, and Nicolas Nisse. "Experimental evaluation of a branch-and-bound algorithm for computing pathwidth and directed pathwidth". In: *Journal of Experimental Algorithmics (JEA)* 21 (2016), pp. 1–3.

[39] Jerzy Czyzowicz et al. “Energy-optimal broadcast in a tree with mobile agents”. In: *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics*. Springer. 2017, pp. 98–113.

[40] Yassine Daadaa. “Network Decontamination with temporal immunity”. PhD thesis. Ph. D. thesis, University of Ottawa, 2012.

[41] Yassine Daadaa, Asif Jamshed, and Mudassir Shabbir. “Network decontamination with a single agent”. In: *Graphs and Combinatorics* 32.2 (2016), pp. 559–581.

[42] Shantanu Das, Dariusz Dereniowski, and Christina Karousatou. “Collaborative exploration of trees by energy-constrained mobile robots”. In: *Theory of Computing Systems* (2018), pp. 1–18.

[43] Shantanu Das, Dariusz Dereniowski, and Przemyslaw Uznanski. “Brief Announcement: Energy Constrained Depth First Search”. In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[44] Nick D Dendris, Lefteris M Kirousis, and Dimitrios M Thilikos. “Fugitive-search games on graphs and related parameters”. In: *Theoretical Computer Science* 172.1-2 (1997), pp. 233–254.

[45] Dariusz Dereniowski. “Approximate search strategies for weighted trees”. In: *Theoretical Computer Science* 463 (2012), pp. 96–113.

[46] Dariusz Dereniowski. “Connected searching of weighted trees”. In: *Theoretical Computer Science* 412.41 (2011), pp. 5700–5713.

[47] Dariusz Dereniowski. “From pathwidth to connected pathwidth”. In: *SIAM Journal on Discrete Mathematics* 26.4 (2012), pp. 1709–1732.

[48] Dariusz Dereniowski and Danny Dyer. “On minimum cost edge searching”. In: *Theoretical Computer Science* 495 (2013), pp. 37–49.

[49] Dariusz Dereniowski, Tomáš Gavenčiak, and Jan Kratochvíl. “Cops, a fast robber and defensive domination on interval graphs”. In: *Theoretical Computer Science* (2018).

[50] Dariusz Dereniowski, Wieslaw Kubiak, and Yori Zwols. “The complexity of minimum-length path decompositions”. In: *Journal of Computer and System Sciences* 81.8 (2015), pp. 1715–1747.

[51] Dariusz Dereniowski and Dorota Urbańska. “On-line Search in Two-Dimensional Environment”. In: *International Workshop on Approximation and Online Algorithms*. Springer. 2017, pp. 223–237.

[52] Dariusz Dereniowski et al. “Clearing directed subgraphs by mobile agents: Variations on covering with paths”. In: *Journal of Computer and System Sciences* 102 (2019), pp. 57–68.

[53] Dariusz Dereniowski et al. “Fast collaborative graph exploration”. In: *Information and Computation* 243 (2015), pp. 37–49.

[54] Magda Dettlaff et al. “On the connected and weakly convex domination numbers”. In: *arXiv preprint arXiv:1902.07505* (2019).

[55] Yann Disser et al. “A general lower bound for collaborative tree exploration”. In: *Theoretical Computer Science* (2018).

[56] Rodney G Downey and Michael Ralph Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012.

[57] Joseph W Durham, Antonio Franchi, and Francesco Bullo. “Distributed pursuit-evasion without mapping or global localization via local frontiers”. In: *Autonomous Robots* 32.1 (2012), pp. 81–95.

[58] Miroslaw Dynia, Miroslaw Korzeniowski, and Christian Schindelhauer. “Power-aware collective tree exploration”. In: *International Conference on Architecture of Computing Systems*. Springer. 2006, pp. 341–351.

[59] Miroslaw Dynia, Jakub ŁopuszaŃski, and Christian Schindelhauer. “Why robots need maps”. In: *International Colloquium on Structural Information and Communication Complexity*. Springer. 2007, pp. 41–50.

[60] Miroslaw Dynia et al. “Smart robot teams exploring sparse trees”. In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2006, pp. 327–338.

[61] John Ellis and Robert Warren. “Lower bounds on the pathwidth of some grid-like graphs”. In: *Discrete Applied Mathematics* 156.5 (2008), pp. 545–555.

[62] John Arthur Ellis, H Sudborough, and Jonathan S Turner. “Graph separation and search number”. In: (1987).

[63] Jonathan A Ellis, Ivan Hal Sudborough, and Jonathan S Turner. “The vertex separation and search number of a graph”. In: *Information and Computation* 113.1 (1994), pp. 50–79.

[64] Uriel Feige, MohammadTaghi Hajiaghayi, and James R Lee. “Improved approximation algorithms for minimum weight vertex separators”. In: *SIAM Journal on Computing* 38.2 (2008), pp. 629–657.

[65] Andreas Emil Feldmann and Dániel Marx. "The complexity landscape of fixed-parameter directed Steiner network problems". In: *arXiv preprint arXiv:1707.06808* (2017).

[66] Amos Fiat and Gerhard J Woeginger. *Online algorithms: The state of the art*. Vol. 1442. Springer, 1998.

[67] Paola Flocchini, Miao Jun Huang, and Flaminia L Luccio. "Decontaminating chordal rings and tori using mobile agents". In: *International Journal of Foundations of Computer Science* 18.03 (2007), pp. 547–563.

[68] Paola Flocchini, Miao Jun Huang, and Flaminia L Luccio. "Decontamination of hypercubes by mobile agents". In: *Networks: An International Journal* 52.3 (2008), pp. 167–178.

[69] Paola Flocchini, Flaminia L Luccio, and Lisa Xiuli Song. "Size Optimal Strategies for Capturing an Intruder in Mesh Networks". In: (2005).

[70] Paola Flocchini, Bernard Mans, and Nicola Santoro. "Tree decontamination with temporary immunity". In: *International Symposium on Algorithms and Computation*. Springer. 2008, pp. 330–341.

[71] Paola Flocchini, Amiya Nayak, and Arno Schulz. "Cleaning an arbitrary regular network with mobile agents". In: *International Conference on Distributed Computing and Internet Technology*. Springer. 2005, pp. 132–142.

[72] Paola Flocchini, Amiya Nayak, and Arno Schulz. "Decontamination of arbitrary networks using a team of mobile agents with limited visibility". In: *6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)*. IEEE. 2007, pp. 469–474.

[73] Paola Flocchini et al. "Network decontamination under *m* immunity". In: *Discrete Applied Mathematics* 201 (2016), pp. 114–129.

[74] Paola Flocchini et al. "Optimal network decontamination with threshold immunity". In: *International Conference on Algorithms and Complexity*. Springer. 2013, pp. 234–245.

[75] Fedor V Fomin. *Complexity of connected search when the number of searchers is small*. Open problems of GRASTA 2017: the 6th Workshop on GRAph Searching, Theory and Applications. 2017. URL: `http://files.thilikos.info/data/conferences/GRASTA2017/open_problems/GRASTA2017-open_problems.pdf`.

[76] Fedor V Fomin and Dimitrios M Thilikos. "An annotated bibliography on guaranteed graph searching". In: *Theoretical computer science* 399.3 (2008), pp. 236–245.

[77] Fedor V Fomin, Dimitrios M Thilikos, and Ioan Todinca. "Connected graph searching in outerplanar graphs". In: *Electronic Notes in Discrete Mathematics* 22.213-216 (2005), 7th.

[78] Fedor V Fomin et al. "Faster algorithms for finding and counting subgraphs". In: *Journal of Computer and System Sciences* 78.3 (2012), pp. 698–706.

[79] Fedor V Fomin et al. "Pursuing a fast robber on a graph". In: *Theoretical Computer Science* 411.7-9 (2010), pp. 1167–1181.

[80] Pierre Fraigniaud, David Ilcinkas, and Andrzej Pelc. "Oracle size: a new measure of difficulty for communication tasks". In: *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. ACM. 2006, pp. 179–187.

[81] Pierre Fraigniaud, David Ilcinkas, and Andrzej Pelc. "Tree exploration with an oracle". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2006, pp. 24–37.

[82] Pierre Fraigniaud and Nicolas Nisse. "Connected treewidth and connected graph searching". In: *Latin American Symposium on Theoretical Informatics*. Springer. 2006, pp. 479–490.

[83] Pierre Fraigniaud et al. "Collective tree exploration". In: *Networks: An International Journal* 48.3 (2006), pp. 166–177.

[84] Zachary Friggstad et al. "Linear programming hierarchies suffice for directed Steiner tree". In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer. 2014, pp. 285–296.

[85] Martin Fürer. "Faster computation of path-width". In: *International Workshop on Combinatorial Algorithms*. Springer. 2016, pp. 385–396.

[86] Harold N. Gabow, Shachindra N Maheshwari, and Leon J. Osterweil. "On two problems in the generation of program test paths". In: *IEEE Transactions on Software Engineering* 3 (1976), pp. 227–231.

[87] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. wh freeman New York, 2002.

[88] Serge Gaspers et al. "Parallel cleaning of a network with brushes". In: *Discrete Applied Mathematics* 158.5 (2010), pp. 467–478.

[89] Stephen T Hedetniemi and Renu C Laskar. "Bibliography on domination in graphs and some basic definitions of domination parameters". In: *Annals of Discrete Mathematics*. Vol. 48. Elsevier, 1991, pp. 257–277.

[90] Yuya Higashikawa et al. "Online graph exploration algorithms for cycles and trees by multiple searchers". In: *Journal of Combinatorial Optimization* 28.2 (2014), pp. 480–495.

[91] Geoffrey Hollinger et al. "Efficient multi-robot search for a moving target". In: *The International Journal of Robotics Research* 28.2 (2009), pp. 201–219.

[92] Miao Jun Huang. "Contiguous search by mobile agents in cube networks and chordal rings". PhD thesis. University of Ottawa (Canada), 2004.

[93] David Ilcinkas, Nicolas Nisse, and David Soguet. "The cost of monotonicity in distributed graph searching". In: *Distributed Computing* 22.2 (2009), pp. 117–127.

[94] Navid Imani, Hamid Sarbazi-Azad, and Albert Y Zomaya. "Capturing an intruder in product networks". In: *Journal of Parallel and Distributed Computing* 67.9 (2007), pp. 1018–1028.

[95] Navid Imani et al. "Detecting threats in star graphs". In: *IEEE Transactions on Parallel and Distributed Systems* 20.4 (2009), pp. 474–483.

[96] Mark Jones et al. "Parameterized complexity of directed steiner tree on sparse graphs". In: *SIAM Journal on Discrete Mathematics* 31.2 (2017), pp. 1294–1327.

[97] Borislav Karaivanov et al. "Decontaminating Planar Regions by Sweeping with Barrier Curves." In: *CCCG*. 2014.

[98] Nancy G Kinnersley. "The vertex separation number of a graph equals its path-width". In: *Information Processing Letters* 42.6 (1992), pp. 345–350.

[99] Lefteris M Kirousis and Christos H Papadimitriou. "Searching and pebbling". In: *Theoretical Computer Science* 47 (1986), pp. 205–218.

[100] Kenta Kitsunai et al. "Computing Directed Pathwidth in $O(1.89^n)$ Time". In: *Algorithmica* 75.1 (2016), pp. 138–157.

[101] Andreas Kolling and Stefano Carpin. "Multi-robot pursuit-evasion without maps". In: *2010 IEEE International Conference on Robotics and Automation*. IEEE. 2010, pp. 3045–3051.

[102] Petr Kolman and Ondřej Pangrác. "On the complexity of paths avoiding forbidden pairs". In: *Discrete Applied Mathematics* 157.13 (2009), pp. 2871–2876.

[103] Ioannis Koutis. "Faster algebraic algorithms for path and packing problems". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2008, pp. 575–586.

[104] Ioannis Koutis and Ryan Williams. "Limits and applications of group algebras for parameterized problems". In: *ACM Transactions on Algorithms (TALG)* 12.3 (2016), p. 31.

[105] Jakub Kováč. "Complexity of the path avoiding forbidden pairs problem revisited". In: *Discrete Applied Mathematics* 161.10-11 (2013), pp. 1506–1512.

[106] Stephan Kreutzer and Sebastian Ordyniak. "Distance d-domination games". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 2009, pp. 308–319.

[107] Bundit Laekhanukit. "Approximating Directed Steiner Problems via Tree Embedding". In: *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[108] Jens Lagergren. "Efficient parallel algorithms for graphs of bounded tree-width". In: *Journal of Algorithms* 20.1 (1996), pp. 20–44.

[109] Andrea S LaPaugh. "Recontamination does not help to search a graph". In: *Journal of the ACM (JACM)* 40.2 (1993), pp. 224–245.

[110] Christos Levcopoulos et al. "Clearing connections by few agents". In: *International Conference on Fun with Algorithms*. Springer. 2014, pp. 289–300.

[111] Harry R Lewis and Christos H Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, 1997.

[112] Fabrizio Luccio and Linda Pagli. "A general approach to toroidal mesh decontamination with local immunity". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–8.

[113] Fabrizio Luccio, Linda Pagli, and Nicola Santoro. "Network decontamination in presence of local immunity". In: *International Journal of Foundations of Computer Science* 18.03 (2007), pp. 457–474.

[114] Flaminia L Luccio. "Contiguous search problem in Sierpiński graphs". In: *Theory of Computing Systems* 44.2 (2009), pp. 186–204.

[115] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[116] Euripides Markou, Nicolas Nisse, and Stéphane Pérennes. "Exclusive graph searching vs. pathwidth". In: *Information and Computation* 252 (2017), pp. 243–260.

[117] Minko Markov, Vladislav Haralampiev, and Georgi Georgiev. "Lower bounds on the directed sweepwidth of planar shapes". In: *Serdica Journal of Computing* 9.2 (2015), pp. 151–166.

[118] Nimrod Megiddo et al. "The complexity of searching a graph". In: *Journal of the ACM (JACM)* 35.1 (1988), pp. 18–44.

[119] Margaret-Ellen Messinger, Richard J Nowakowski, and P Prałat. "Cleaning a network with brushes". In: *Theoretical Computer Science* 399.3 (2008), pp. 191–205.

[120] Margaret-Ellen Messinger, Richard J Nowakowski, and Paweł Prałat. "Cleaning with brooms". In: *Graphs and Combinatorics* 27.2 (2011), pp. 251–267.

[121] Nicolas Nisse and David Soguet. "Graph searching with advice". In: *Theoretical Computer Science* 410.14 (2009), pp. 1307–1318.

[122] Richard Nowakowski and Peter Winkler. "Vertex-to-vertex pursuit in a graph". In: *Discrete Mathematics* 43.2-3 (1983), pp. 235–239.

[123] Simeon Ntafos and Teofilo Gonzalez. "On the computational complexity of path cover problems". In: *Journal of Computer and System Sciences* 29.2 (1984), pp. 225–242.

[124] Simeon C. Ntafos and S. Louis Hakimi. "On path cover problems in digraphs and applications to program testing". In: *IEEE Transactions on Software Engineering* 5 (1979), pp. 520–529.

[125] Simeon C Ntafos and S Louis Hakimi. "On structured digraphs and program testing". In: *IEEE Transactions on Computers* 100.1 (1981), pp. 67–77.

[126] Christian Ortolf and Christian Schindelhauer. "A recursive approach to multi-robot exploration of trees". In: *International Colloquium on Structural Information and Communication Complexity*. Springer. 2014, pp. 343–354.

[127] Christian Ortolf and Christian Schindelhauer. "Online multi-robot exploration of grid graphs with rectangular obstacles". In: *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2012, pp. 27–36.

[128] Dorota Osula. "Decontaminating Arbitrary Graphs by Mobile Agents: a Survey". In: *Utilitas Mathematica* (to appear). ISSN: 0315-3681.

[129] Dorota Osula. "Minimizing the Cost of Team Exploration". In: *International Conference on Current Trends in Theory and Practice of Informatics*. Springer. 2019, pp. 392–405.

[130] Dorota Osula and Rita Zuazua. "Twin domination number of Tournaments". In: *arXiv preprint arXiv:1702.00646* (2017).

[131] Richard Otter. "The number of trees". In: *Ann. Math* 49.2 (1948), pp. 583–599.

[132] Torrence D Parsons. "Pursuit-evasion in a graph". In: *Theory and applications of graphs*. Springer, 1978, pp. 426–441.

[133] Nikolai Nikolaevich Petrov. "Some extremal search problems on graphs". In: *Differentsial'nye Uravneniya* 18.5 (1982), pp. 821–827.

[134] Jun Qiu. "Best effort decontamination of networks". PhD thesis. University of Ottawa (Canada), 2007.

[135] Eric Raboin, Ugur Kuter, and Dana Nau. "Generating strategies for multi-agent pursuit-evasion games in partially observable Euclidean space". In: *The 11th International Conference on Autonomous Agents and Multiagent Systems*. Vol. 3. International Foundation for Autonomous Agents and Multiagent Systems. 2012, pp. 1201–1202.

[136] Bruce A Reed. "Finding approximate separators and computing tree width quickly". In: *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. ACM. 1992, pp. 221–228.

[137] Neil Robertson and Paul D Seymour. "Graph minors. I. Excluding a forest". In: *Journal of Combinatorial Theory, Series B* 35.1 (1983), pp. 39–61.

[138] Neil Robertson and Paul D Seymour. "Graph minors. XX. Wagner's conjecture". In: *Journal of Combinatorial Theory, Series B* 92.2 (2004), pp. 325–357.

[139] Cyril Robin and Simon Lacroix. "Multi-robot target detection and tracking: taxonomy and survey". In: *Autonomous Robots* 40.4 (2016), pp. 729–760.

[140] Samuel Rodriguez et al. "Toward realistic pursuit-evasion using a roadmap-based approach". In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 1738–1745.

[141] Shai Sachs, Steven M LaValle, and Stjepan Rajko. "Visibility-based pursuit-evasion in an unknown planar environment". In: *The International Journal of Robotics Research* 23.1 (2004), pp. 3–26.

[142] Pooya Shareghi, Navid Imani, and Hamid Sarbazi-Azad. "Capturing an intruder in the pyramid". In: *International Computer Science Symposium in Russia*. Springer. 2006, pp. 580–590.

[143] Nicholas M Stiffler and Jason M O'Kane. "A complete algorithm for visibility-based pursuit-evasion with multiple pursuers". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 1660–1667.

[144] Karol Suchan and Yngve Villanger. "Computing pathwidth faster than 2^n". In: *International Workshop on Parameterized and Exact Computation*. Springer. 2009, pp. 324–335.

[145] Ondřej Suchỳ. "On directed steiner trees with multiple roots". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 2016, pp. 257–268.

[146] Atsushi Takahashi, Shuichi Ueno, and Yoji Kajitani. "Mixed searching and proper-path-width". In: *Theoretical Computer Science* 137.2 (1995), pp. 253–268.

[147] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[148] Dimitri Watel et al. "Directed Steiner tree with branching constraint". In: *International Computing and Combinatorics Conference*. Springer. 2014, pp. 263–275.

[149] Dimitri Watel et al. "Directed Steiner trees with diffusion costs". In: *Journal of Combinatorial Optimization* 32.4 (2016), pp. 1089–1106.

[150] Ryan Williams. "Finding Paths of Length k in $O^*(2^k)$ time". In: *Information Processing Letters* 109.6 (2009), pp. 315–318.

[151] Boting Yang, Danny Dyer, and Brian Alspach. "Sweeping graphs with large clique number". In: *International symposium on algorithms and computation*. Springer. 2004, pp. 908–920.

[152] Marguerite Zientara. Personal communication. 2016.

[153] Leonid Zosin and Samir Khuller. "On directed Steiner trees". In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2002, pp. 59–63.